IBM WebSphere Application Server for IBM i, Version 8.0

Scripting the application serving environment

Note

Before using this information, be sure to read the general information under “Notices” on page 961.

Compilation date: July 31, 2011

Contents

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an e-mail form appears.
  3. Fill out the e-mail form as instructed, and click on **Submit feedback** .
- To send comments on PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com** or fax them to 919-254-5250.

  Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Changes to serve you more quickly

Print sections directly from the information center navigation

PDF books are provided as a convenience format for easy printing, reading, and offline use. The information center is the official delivery format for IBM WebSphere Application Server documentation. If you use the PDF books primarily for convenient printing, it is now easier to print various parts of the information center as needed, quickly and directly from the information center navigation tree.

To print a section of the information center navigation:

1. Hover your cursor over an entry in the information center navigation until the **Open Quick Menu** icon is displayed beside the entry.
2. Right-click the icon to display a menu for printing or searching your selected section of the navigation tree.
3. If you select **Print this topic and subtopics** from the menu, the selected section is launched in a separate browser window as one HTML file. The HTML file includes each of the topics in the section, with a table of contents at the top.
4. Print the HTML file.

For performance reasons, the number of topics you can print at one time is limited. You are notified if your selection contains too many topics. If the current limit is too restrictive, use the feedback link to suggest a preferable limit. The feedback link is available at the end of most information center pages.

Under construction!

The Information Development Team for IBM WebSphere Application Server is changing its PDF book delivery strategy to respond better to user needs. The intention is to deliver the content to you in PDF format more frequently. During a temporary transition phase, you might experience broken links. During the transition phase, expect the following link behavior:

- Links to Web addresses beginning with http:// work
- Links that refer to specific page numbers within the same PDF book work
- The remaining links will *not* work. You receive an error message when you click them

Thanks for your patience, in the short term, to facilitate the transition to more frequent PDF book updates.

Chapter 1. How do I use wsadmin commands to administer applications and their environments?

The wsadmin tool is a command-line interface that provides the ability to automate common tasks using Jacl or Jython scripts. The AdminTask, AdminApp, AdminControl, AdminConfig, and Help objects provide many commands and options that allow you to write and customize scripts to administer your applications, environment, web services, resources, and security configurations. Follow these shortcuts to get started quickly with popular tasks.

Use scripting to configure web services policy sets

Use scripting to create secure sessions between clients and services

Use scripting to configure application servers

Use scripting to manage application servers

Use scripting to update applications

Use scripting to administer communication with web servers (plug-ins)

Use scripting to administer HTTP sessions

Use scripting to provide access to relational databases (JDBC resources)

Use scripting to provide access to messaging resources (default messaging provider)

Use scripting to secure applications and their environments

Chapter 2. Using wsadmin scripting

The WebSphere® administrative (wsadmin) scripting program is a powerful, non-graphical command interpreter environment enabling you to run administrative operations in a scripting language.

About this task

The wsadmin tool is intended for production environments and unattended operations. You can use the wsadmin tool to perform the same tasks that you can perform using the administrative console.

The following list highlights the topics and tasks available with scripting:

Procedure

- Getting started with scripting Provides an introduction to WebSphere Application Server scripting and information about using the wsadmin tool. Topics include information about the scripting languages and the scripting objects, and instructions for starting the wsadmin tool.
- Using the script library to automate the application serving environment Provides a set of Jython script procedures that automate the most common application server administration functions. For example, you can use the script library to easily configure servers, applications, mail settings, resources, nodes, business-level applications, clusters, authorization groups, and more. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.
- Deploying applications Provides instructions for deploying and uninstalling applications. For example, stand-alone Java archive files and web archive files, the administrative console, remote Enterprise Archive (EAR) files, file transfer applications, and so on.
- Managing deployed applications Includes tasks that you perform after the application is deployed. For example, starting and stopping applications, checking status, modifying listener address ports, querying application state, configuring a shared library, and so on.
- Configuring servers Provides instructions for configuring servers, such as creating a server, modifying and restarting the server, configuring the Java virtual machine, disabling a component, disabling a service, and so on.
- Configuring connections to web servers Includes topics such as regenerating the plug-in, creating new virtual host templates, modifying virtual hosts, and so on.
- Managing servers Includes tasks that you use to manage servers. For example, stopping nodes, starting and stopping servers, querying a server state, starting a listener port, and so on.
- Configuring security Includes security tasks, for example, enabling and disabling administrative security, enabling and disabling Java 2 security, and so on.
- Configuring data access Includes topics such as configuring a Java DataBase Connectivity (JDBC) provider, defining a data source, configuring connection pools, and so on.
- Configuring messaging Includes topics about messaging, such as Java Message Service (JMS) connection, JMS provider, WebSphere queue connection factory, MQ topics, and so on.
- Configuring mail, URLs, and resource environment entries Includes topics such as mail providers, mail sessions, protocols, resource environment providers, referenceables, URL providers, URLs, and so on.
- Troubleshooting Provides information about how to troubleshoot using scripting. For example, tracing, thread dumps, profiles, and so on.
- Scripting reference material Includes all of the reference material related to scripting. Topics include the syntax for the wsadmin tool and for the administrative command framework, explanations and examples for all of the scripting object commands, the scripting properties, and so on.

Chapter 3. Scripting concepts

Scripting provides a non-graphical alternative to the administrative console. Using the wsadmin tool, you can run scripts to configure and manage the product. The wsadmin tool supports two scripting languages: Jacl and Jython. Five objects are available with scripts: AdminControl, AdminConfig, AdminApp, AdminTask, and Help. Scripts use these objects to communicate with MBeans that run in product processes. MBeans are Java objects that represent Java Management Extensions (JMX) resources. JMX is a technology that provides a simple and standard way to manage Java objects.

Using wsadmin scripting with Java Management Extensions (JMX)

Java Management Extensions (JMX) is a framework that provides a standard way of exposing Java resources, for example, application servers, to a system management infrastructure. Using the JMX framework, a provider can implement functions, such as listing the configuration settings, and editing the settings. This framework also includes a notification layer that management applications can use to monitor events such as the startup of an application server.

JMX key features

The key features of the WebSphere Application Server implementation of JMX include:

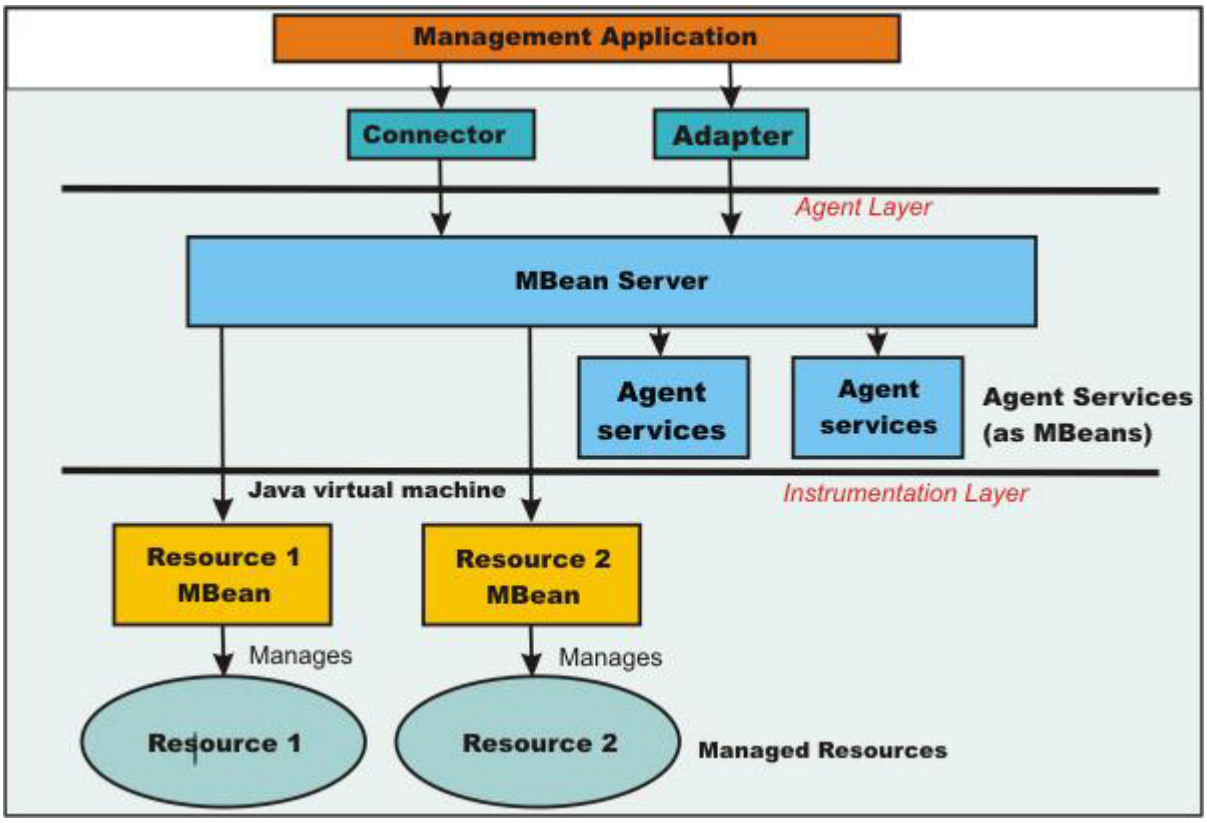
- All processes that run the JMX agent.
- All run-time administration that is performed through JMX operations.
- Connectors that are used to connect a JMX agent to a remote JMX-enabled management application. The following connectors are supported:
  - SOAP JMX Connector
  - JMX Remote application programming interface (JSR 160) Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) JMX Connector, (the JSR160RMI connector)
  - Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) JMX Connector
  - Inter-Process Communications (IPC)
- Protocol adapters that provide a management view of the JMX agent through a given protocol. Management applications that connect to a protocol adapter are usually specific to a given protocol.
- The ability to query and update the configuration settings of a run-time object.
- The ability to load, initialize, change, and monitor application components and resources during run time.

JMX architecture

The JMX architecture is structured into three layers:

- Instrumentation layer - Dictates how resources can be wrapped within special Java beans, called managed beans (MBeans).
- Agent layer - Consists of the MBean server and agents, which provide a management infrastructure. The services that are implemented include:
  - Monitoring
  - Event notification
  - Timers
- Management layer - Defines how external management applications can interact with the underlying layers in terms of protocols, APIs, and so on. This layer uses an implementation of the distributed services specification (JSR-077), which is not yet part of the Java 2 platform, Enterprise Edition (J2EE) specification.

The layered architecture of JMX is summarized in the following figure:



JMX distributed administration

The following figure shows how the JMX architecture fits into the overall distributed administration topology of a WebSphere Application Server, Network Deployment environment:

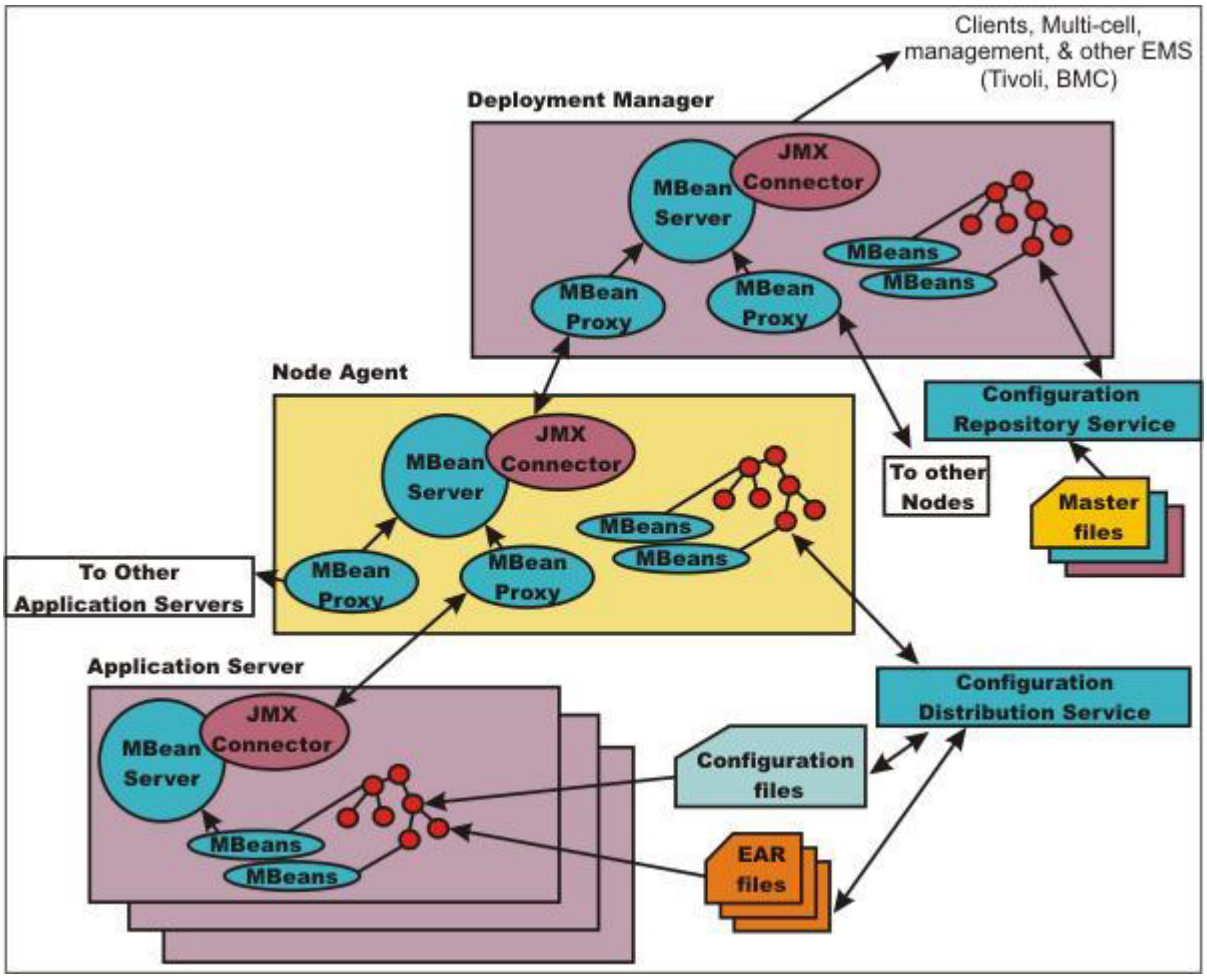


Figure 2: WebSphere Application Server distributed administration of JMX

The key points of this distributed administration architecture include:

- Internal MBeans that are local to the Java virtual machine (JVM) register with the local MBean server.
- External MBeans have a local proxy to their MBean server. The proxy registers with the local MBean server.

JMX Mbeans

WebSphere Application Server provides a number of MBeans, each of which has different functions and operations available. For example, an application server MBean can expose operations such as start and stop. An application MBean can expose operations such as install and uninstall. Some JMX usage scenarios that you can encounter include:

- External programs that are written to control the WebSphere Application Server, Network Deployment run time and its WebSphere resources by programmatically accessing the JMX API.
- Third-party applications that include custom JMX MBeans as part of the deployed code, supporting the JMX API management of application components and resources.

The following example illustrates how to obtain the name of a particular MBean:

Using Jacl:

```
set am [$AdminControl queryNames type=ApplicationManager,process=server1,*]
```

Using Jython:

```
am = AdminControl.queryNames('type=ApplicationManager,process=server1,*')
```

Each WebSphere Application Server runtime MBean can have attributes, operations, and notifications. The complete documentation for each MBean that is supplied with the product is available in this information center at ***information_center*** > **Reference** > **Programming interfaces** > **Mbean interfaces**.

JMX benefits

The use of JMX for management functions in WebSphere Application Server provides the following benefits:

- Enables the management of Java applications without significant investment.
- Relies on a core-managed object server that acts as a management agent.
- Java applications can embed a managed object server and make some of its functionality available as one or several MBeans that are registered with the object server.
- Provides a scalable management architecture.
- Every JMX agent service is an independent module that can be plugged into the management agent.
- The API is extensible, allowing new WebSphere Application Server and custom application features to be easily added and exposed through this management interface.
- Integrates existing management solutions.
- Each process is self-sufficient when it comes to the management of its resources. No central point of control exists. In principle, a JMX-enabled management client can be connected to any managed process and interact with the MBeans that are hosted by that process.
- JMX provides a single, flat, domain-wide approach to system management. Separate processes interact through MBean proxies that support a single management client to seamlessly navigate through a network of managed processes.
- Defines the interfaces that are necessary for management only.
- Provides a standard API for exposing application and administrative resources to management tools.

WebSphere Application Server configuration model using wsadmin scripting

Understanding the relationship between the different configuration objects is essential when creating wsadmin scripts that perform configuration function.

Configuration data is stored in several different XML files which the server run time reads when it starts and responds to the component settings stored there. The configuration data includes the settings for the run time, such as, Java virtual machine (JVM) options, thread pool sizes, container settings, and port numbers the server will use. Other configuration files define Java 2 Platform, Enterprise Edition (J2EE) resources to which the server connects in order to obtain data that is needed by the application logic. Security settings are stored in a separate document from the server and resource configuration. Application-specific configuration, such as, deployment target lists, session configuration, and cache settings, are stored in files under the root directory of each application. When viewing the XML data in the configuration files, you can discern relationship between the configuration objects.

For more information on the WebSphere Application Server configuration objects view the HTML tables in the *installroot*/web/configDocs directory. There are several subdirectories, one for each configuration package in the model. The index.html file ties all of the individual configuration packages together in a top-level navigation tree. Each configuration package lists the supported configuration classes and the configuration class lists all of the supported properties. The properties with names that end with the at (@) character imply that property is a reference to a different configuration object within the configuration data. The properties with names that end with an asterisk (*) character imply that the property is a list of other configuration objects.

Using wsadmin scripting with Jacl

Jacl is an alternate implementation of TCL, and is written entirely in Java code.

The wsadmin tool uses Jacl V1.3.2.

Stabilization of the Jacl syntax in the wsadmin tool

The Jacl language stabilized in Version 7 of the product. IBM® does not currently plan to deprecate or remove this capability in a subsequent release of the product; but future investment will be focused on the Jython language, which is the strategic alternative. You do not need to change any of your existing applications and scripts that use Jacl; but you should consider using the strategic alternative for new applications.

The Jython syntax for the wsadmin tool is the strategic direction for WebSphere Application Server administrative automation. The product continues to provide enhanced administrative functions and tooling that support product automation and the use of the Jython syntax. The following Jython scripting-related enhancements are provided in the product:

- Administrative console command assist - A feature of the administrative console that displays the wsadmin command that is equivalent to the action taken by the user that interacts with the console. The output from the console command assist feature can be transferred directly to the WebSphere Application Server Tool, which simplifies the development of Jython scripts that are based on administrative console actions. You can also save the output after using the console command assist feature in a plain text file for later use.
- Jacl-to-Jython conversion utility - A program that converts Jacl syntax wsadmin scripts into equivalent Jython syntax wsadmin scripts. Dozens of new wsadmin high-level commands that decouple the script from the underlying administrative model through use of simple parameters and smart default logic.

Basic syntax

The basic syntax for a Jacl command is the following:

```
Command arg1 arg2 arg3 ...
```

The command is either the name of a built-in command or a Jacl procedure. For example:

```
puts stdout {Hello, world!}
=> Hello, world!
```

In this example, the command is puts which takes two arguments, an I/O stream identifier and a string. The puts command writes the string to the I/O stream along with a trailing new line character. The arguments are interpreted by the command. In the example, `stdout` is used to identify the standard output stream. The use of `stdout` as a name is a convention employed by the puts command and the other I/O commands. `stderr` identifies the standard error output, and `stdin` identifies the standard input.

Variables

The set command assigns a value to a variable. This command takes two arguments: the name of the variable and the value. Variable names can be any length and are case sensitive. You do not have to declare Jacl variables before you use them. The interpreter will create the variable when it is first assigned a value. For example:

```
set a 5
=> 5

set b $a
=> 5
```

The second example assigns the value of variable `a` to variable `b`. The use of dollar sign ($) indicates variable substitution. You can delete a variable with the unset command, for example:

```
unset varName1 varName2 ...
```

You can pass any number of variables to the unset command. The unset command gives an error if a variable is not already defined. You can delete an entire array or just a single array element with the unset command. Using the unset command on an array is an easy way to clear out a big data structure. The existence of a variable can be tested with the **info exists** command. You might need to test for the existence of the variable because the `incr` parameter requires that a variable exist first, for example:

```
if ![info exists my_info] {set my_info 0} else {incr my_info}
```

Command substitution

The second form of substitution is command substitution. A nested command is delimited by square brackets, [ ]. The Jacl interpreter evaluates everything between the brackets and evaluates it as a command. For example:

```
set len [string length my_string]
=> 6
```

In this example, the nested command is the following: `string length` *my_string*. The string command performs various operations on strings. In this case, the command asks for the length of the string *my_string*. If there are several cases of command substitution within a single command, the interpreter processes them from left bracket to right bracket. For example:

```
set number "1 2 3 4"
=> 1 2 3 4

set one [lindex $number 0]
=> 1

set end [lindex $number end]
=> 4

set another {123  456  789}
=> 123  456  789

set stringLen [string length [lindex  $another  1]]
=> 3

set listLen [llength [lindex $another 1]
=> 1
```

Math expressions

The Jacl interpreter does not evaluate math expressions. Use the expr command to evaluate math expressions. The implementation of the expr command takes all arguments, concatenates them into a single string, and parses the string as a math expression. After the expr command computes the answer, it is formatted into a string and returned. For example:

```
expr 7.2 / 3
=> 2.4
```

Backslash substitution

The final type of substitution done by the Jacl interpreter is backslash substitution. Use backslashes to add quotation characters that have special meaning to the interpreter. For example, you can specify a literal dollar sign, brace, or bracket by quoting it with a backslash. If you are using lots of backslashes, instead you can group things with curly braces to turn off all interpretation of special characters. There are cases where backslashes are required. For example:

```
set dollar "This is a string \$contain dollar char"
=> This is a string $contain dollar char

set x $dollar
=> This is a string $contain dollar char

set  group {$ {} [] { [ } ]}
=> $ {} [] { [ } ]
```

You can also use backslashes to continue long commands on multiple lines. A new line without the backslash terminates a command. A backslash that is the last character on a line convert into a space. For example:

```
set  totalLength  [expr  [string  length  "first string"] + \
[string  length  "second string"]]
=> 25
```

Grouping with braces and double quotation marks

Use double quotation marks and curly braces to group words together. Quotation marks enable substitutions to occur in the group and curly braces prevent substitution. This rule applies to command, variable, and backslash substitutions. For example:

```
set s Hello
=> Hello

puts stdout "The length of $s is [string length $s]."
=> The length of Hello is 5.

puts  stdout  {The length of $s is [string  length  $s].}
=> The length of $s is [string length $s].
```

In the second example, the Jacl interpreter performs variable and command substitution on the second argument from the puts command. In the third command, substitutions are prevented so the string is printed as it is.

Procedures and scope

Jacl uses the proc command to define procedures. The basic syntax to define a procedure is the following:

```
proc  name  arglist  body
```

The first argument is the name of the procedure being defined. The name is case sensitive, and in fact it can contain any characters. Procedure names and variable names do not conflict with each other. The second argument is a list of parameters to the procedures. The third argument is a command, or more typically a group of commands that form the procedure body. Once defined, a Jacl procedure is used just like any of the built-in commands. For example:

```
proc divide {x y} {
set result [expr $x/$y]
puts $result
}
```

Inside the script, this is how to call divide procedure:

```
divide 20 5
```

And it gives a result resembling the following:

```
4
```

It is not necessary to use the variable c in this example. The procedure body might also written as:

```
return  [expr  sqrt($a  *  $a  +  $b  *  $b)]
```

The return command is optional in this example because the Jacl interpreter returns the value of the last command in the body as the value of the procedure. So, the procedure body might be reduced to:

```
expr  sqrt($a  *  $a  +  $b  *  $b)
```

The result of the procedure is the result returned by the last command in the body. The return command can be used to return a specific value.

There is a single, global scope for procedure names. You can define a procedure inside another procedure, but it is visible everywhere. There is a different name space for variables and procedures

therefore you might have a procedure and a variable with the same name without a conflict. Each procedure has a local scope for variables. Variables introduced in the procedures exist only for the duration of the procedure call. After the procedure returns, those variables are undefined. If the same variable name exists in an outer scope, it is unaffected by the use of that variable name inside a procedure. Variables defined outside the procedure are not visible to a procedure, unless the global scope commands are used.

The global scope is the top-level scope. This scope is outside of any procedure. You must make variables defined at the global scope accessible to the commands inside procedure by using the global command. The syntax for the global command is the following:

```
global  varName1  varName2 ...
```

Comments

Use the pound character (#) to make comments.

Command-line arguments

The Jacl shells pass the command-line arguments to the script as the value of the `argv` variable. The number of command-line arguments is given by `argc` variable. The name of the program, or script, is not part of `argv` nor is it counted by `argc`. The `argv` variable is a list. Use the lindex command to extract items from the argument list, for example:

```
set  first  [lindex  $argv  0]
set  second  [lindex  $argv  1]
```

Strings and pattern matching

Strings are the basic data item in the Jacl language. There are multiple commands that you can use to manipulate strings. The general syntax of the string command is the following:

```
string operation stringvalue otherargs
```

The `operation` argument determines the action of the string. The second argument is a string value. There might be additional arguments depending on the operation.

The following table includes a summary of the string command:

Table 1. string command syntax descriptions. Run the string command with one or more arguments.

Command	Description
string compare str1 str2	Compares strings lexicographically. Returns 0 if equal, -1 if str1 sorts before str2, else1.
string first str1 str2	Returns the index in str2 of the first occurrence of str1, or -1 if str1 is not found.
string index string1 index1	Returns the character at the specified index.
string last str1 str2	Returns the index in str2 of the last occurrence of str1, or -1 if str1 is not found.
string length string	Returns the number of characters in the string.
string match pattern str	Returns 1 if str matches the pattern, else 0.
string range str i j	Returns the range of characters in str from i to j
string tolower string	Returns string in lowercase.
string toupper string	Returns string in uppercase.
string trim string ?chars?	Trims the characters in chars from both ends of string. chars defaults to white space.
string trimleft string ?chars?	Trims the characters in chars from the beginning of string. chars defaults to white space.

Table 1. string command syntax descriptions (continued). Run the string command with one or more arguments.

Command	Description
string trimright string ?chars?	Trims the characters in chars from the end of string. chars defaults to white space.
string wordend str ix	Returns the index in str of the character after the word containing the character at index ix.
string wordstart str ix	Returns the index in str of the first character in the word containing the character at index ix.

The append command

The first argument of the append command is a variable name. It concatenates the remaining arguments onto the current value of the named variable. For example:

```
set  my_item  z
=> z

append  my_item a b c
=> zabc
```

The regexp command

The regexp command provides direct access to the regular expression matcher. The syntax is the following:

```
regexp  ?flags?  pattern  string  ?match  sub1  sub2  ...?
```

The return value is `1` if some part of the string matches the pattern. Otherwise, the return value is `0`. The pattern does not have to match the whole string. If you need more control than this, you can anchor the pattern to the beginning of the string by starting the pattern with ^, or to the end of the string by ending the pattern with dollar sign, $. You can force the pattern to match the whole string by using both characters. For example:

```
set  text1  "This is the first string"
=> This is the first string

regexp  "first string" $text1
=> 1

regexp "second string"  $text1
=> 0
```

Jacl data structures

The basic data structure in the Jacl language is a string. There are two higher level data structures: lists and arrays. Lists are implemented as strings and the structure is defined by the syntax of the string. The syntax rules are the same as for commands. Commands are particular instances of lists. Arrays are variables that have an index. The index is a string value so you can think of arrays as maps from one string (the index) to another string (the value of the array element).

Jacl lists

The lists of the Jacl language are strings with a special interpretation. In the Jacl language, a list has the same structure as a command. A list is a string with list elements separated by white space. You can use braces or quotation marks to group together words with white space into a single list element.

The following table includes commands that are related to lists:

Table 2. list command syntax descriptions. Run the list command with one or more arguments.

Command	Description

Table 2. list command syntax descriptions (continued). Run the list command with one or more arguments.

list arg1 arg2	Creates a list out of all its arguments.
lindex list i	Returns the i'th element from list.
llength list	Returns the number of elements in list.
lrange list i j	Returns the i'th through j'th elements from list.
lappend listVar arg arg ...	Appends elements to the value of listVar
linsert list index arg arg ...	Inserts elements into list before the element at position index. Returns a new list.
lreplace list i j arg arg ...	Replaces elements i through j of list with the args. Return a new list.
lsearch mode list value	Returns the index of the element in list that matches the value according to the mode, which is -exact, -glob, or -regexp, -glob is the default. Return -1 if not found.
lsort switches list	Sorts elements of the list according to the switches: -ascii, -integer, -real, -increasing, -decreasing, -command command. Return a new list.
concat arg arg arg ...	Joins multiple lists together into one list.
join list joinString	Merges the elements of a list together by separating them with joinString.
split string splitChars	Splits a string up into list elements, using the characters in splitChars as boundaries between list elements.

Arrays

Arrays are the other primary data structure in the Jacl language. An array is a variable with a string-valued index, so you can think of an array as a mapping from strings to strings. Internally an array is implemented with a hash table. The cost of accessing each element is about the same. The index of an array is delimited by parentheses. The index can have any string value, and it can be the result of variable or command substitution. Array elements are defined with the set command, for example:

```
set arr(index) value
```

Substitute the dollar sign ($) to obtain the value of an array element, for example:

```
set my_item $arr(index)
```

For example:

```
set fruit(best) kiwi
=> kiwi

set fruit(worst) peach
=> peach

set fruit(ok) banana
=> banana

array get fruit
=> ok banana worst peach best kiwi

array exists fruit
=> 1
```

The following table includes array commands:

Table 3. array command syntax descriptions. Run the array command with an argument.

Command	Description
array exists arr	Returns 1 if arr is an array variable.
array get arr	Returns a list that alternates between an index and the corresponding array value.
array names arr ?pattern?	Return the list of all indexes defined for arr, or those that match the string match pattern.

Table 3. array command syntax descriptions (continued). Run the array command with an argument.

Command	Description
array set arr list	Initializes the array arr from list, which need the same form as the list returned by get.
array size arr	Returns the number of indexes defined for arr.
array startsearch arr	Returns a search token for a search through arr.
array nextelement arr id	Returns the value of the next element in array in the search identified by the token id. Returns an empty string if no more elements remain in the search.
array anymore arr id	Returns 1 if more elements remain in the search.
array donesearch arr id	Ends the search identified by `id`.

Control flow commands

The following looping commands exist:

- `while`
- `foreach`
- `for`

The following are conditional commands:

- `if`
- `switch`

The following is an error handling command:

- `catch`

The following commands fine-tune control flow:

- `break`
- `continue`
- `return`
- `error`

if then else

The if command is the basic conditional command. It says that if an expression is true, then run the second line of code, otherwise run a different line of code. The second command body (the else clause) is optional. The syntax of the command is the following:

```
if  boolean  then  body1  else  body2
```

The then and else keywords are optional. For example:

```
if {$x == 0} {
 puts stderr "Divide by zero!"
} else {
 set slope [expr $y/$x]
}
```

switch

Use the switch command to branch to one of many commands depending on the value of an expression. You can choose based on pattern matching as well as simple comparisons. Any number of pattern-body pairs can be specified. If multiple patterns match, only the code body of the first matching pattern is evaluated. The general form of the command is the following:

```
switch  flags  value  pat1  body1  pat2  body2  ...
```

You can also group all the pattern-body pairs into one argument:

```
switch  flags  value  {pat1  body1  pat2  body2  ...}
```

There are four possible flags that determine how value is matched.

- `-exact` Matches the value exactly to one of the patterns.
- `-glob` Uses glob-style pattern matching.
- `-regexp` Uses regular expression pattern matching.
- `--` No flag (or end of flags). Useful when value can begin with a dash (-).

For example:

```
switch  -exact  --  $value  {
 foo  {doFoo;  incr  count(foo)}
 bar {doBar;  return  $count(foo)}
 default   {incr  count(other)}
}
```

If the pattern that is associated with the last body is `default`, then the command body is started if no other patterns match. The `default` keyword works only on the last pattern-body pair. If you use the default pattern on an earlier body, it is treated as a pattern to match the literal string default.

foreach

The foreach command loops over a command body and assigns a loop variable to each of the values in a list. The syntax is the following:

```
foreach loopVar valueList commandBody
```

The first argument is the name of a variable. The command body runs one time for each element in the loop with the loop variable having successive values in the list. For example:

```
set numbers {1 3 5 7 11 13}
foreach num $numbers {
puts $num
}
```

The result from the previous example is the following output, assuming that only one server exists in the environment. If there is more than one server, the information for all servers returns:

```
1
3
5
7
11
13
```

while

The while command takes two arguments; a test and a command body, for example:

```
while  booleanExpr  body
```

The while command repeatedly tests the boolean expression and runs the body if the expression is true (non-zero). For example:

```
set i 0
while {$i < 5} {
puts "i is $i"
incr i}
```

The result from the previous example resembles the following output, assuming that there is only one server. If there is more than one server, it prints all of the servers:

```
i is 0
i is 1
i is 2
i is 3
i is 4
```

for

The for command is similar to the C language for statement. It takes four arguments, for example:

```
for initial test final body
```

The first argument is a command to initialize the loop. The second argument is a boolean expression which determines if the loop body runs. The third argument is a command that runs after the loop body: For example:

```
set numbers {1 3 5 7 11 13}
for {set i 0} {$i < [llength $numbers]} {incr i 1} {
puts "i is $i"
}
```

The result from previous example resembles the following output, assuming that there is only one server in the environment. If there is more than one server, it prints all of the server names:

```
i is 1
i is 3
i is 5
i is 7
i is 11
i is 13
```

break and continue

You can control loop execution with the break and continue commands. The break command causes an immediate exit from a loop. The continue command causes the loop to continue with the next iteration.

catch

An error occurs if you call a command with the wrong number of arguments or if the command detects some error condition particular to its implementation. An uncaught error prevents a script from running. Use the catch command trap such errors. The catch command takes two arguments, for example:

```
catch command ?resultVar?
```

The first argument is a command body. The second argument is the name of a variable that contains the result of the command or an error message if the command raises an error. The catch command returns a value of zero if no error was caught or a value of one if the command catches an error. For example:

```
catch {expr 20 / 5} result
==> 0
puts $result
==> 4
catch {expr text / 5} result
==> 1
puts $result
==> syntax error in expression "text / 5"
```

return

Use the return command to return a value before the end of the procedure body or if a contrast value must be returned.

Namespaces

Jacl tracks named entities such as variables, in namespaces. The wsadmin tool also adds entries to the global namespace for the scripting objects, such as, the AdminApp object

When you run a proc command, a local namespace is created and initialized with the names and the values of the parameters in the proc command. Variables are held in the local namespace while you run the proc command. When you stop the proc command, the local namespace is erased. The local namespace of the proc command implements the semantics of the automatic variables in languages such as C and Java.

While variables in the global namespace are visible to the top-level code, they are not visible by default from within a proc command. To make them visible, declare the variables globally using the global command. For the variable names that you provide, the global command creates entries in the local namespace that point to the global namespace entries that actually define the variables.

If you use a scripting object provided by the wsadmin tool in a proc, you must declare it globally before you can use it, for example:

```
proc { ... } {
  global AdminConfig
  ... [$AdminConfig ...]
}
```

Calling scripts using another script

Use the source command to call a Jacl script from another Jacl script. For example:

Create a script called `test1.jacl`.

Create a script called `testProcedure.jacl`.

```
proc printName {first last} {
 puts "My name is $first $last"
}
```

Pass the following path as a script argument.

You must use forward slashes (/) as your path separator. Backward slashes (\) do not work.

Redirection using the exec command

The following Jacl exec command for redirection does not work on Linux platforms:

```
eval exec ls -l > /tmp/out
```

The exec command of the Jacl scripting language does not fully support redirection therefore it might cause problems on some platforms.

Do not use redirection when using the exec command of the Jacl language. Instead, you can save the exec command for redirection in a variable and write it to a file, for example:

```
open /tmp/out w puts $fileId $result close $fileId
```

In some cases, you can also perform a redirection using shell and a `.sh` command redirection, not a redirection issued by Tcl.

Using wsadmin scripting with Jython

Jython is an alternate implementation of Python, and is written entirely in Java.

The wsadmin tool uses Jython V2.1. The following information is a basic summary of the Jython syntax. In all sample code, the => notation at the beginning of a line represents command or function output.

Basic function

The function is either the name of a built-in function or a Jython function. For example, the following functions return "Hello, World!" as the output:

```
print "Hello, World!"
=>Hello, World!

import sys
sys.stdout.write("Hello World!\n")
=>Hello, World!
```

In the example, `print` identifies the standard output stream. You can use the built-in module by running import statements such as the previous example. The statement import runs the code in a module as part of the importing and returns the module object. `sys` is a built-in module of the Python language. In the Python language, modules are name spaces which are places where names are created. Names that reside in modules are called attributes. Modules correspond to files and the Python language creates a module object to contain all the names defined in the file. In other words, modules are name spaces.

gotcha: When you issue a Jython command in a wsadmin script that invokes a WebSphere Application Server MBean operation, and the MBean method returns a string that includes some NLS translated characters such as the French accent character, Jython automatically converts the string to a python unicode string, and returns the converted string to wsadmin. If you include the Jython print output command in the script that invokes the MBean method, the NLS translated characters are included in the string that the MBean method returns to wsadmin instead of the python unicode values. To avoid the displaying of NLS translated characters, use a variable for the MBean return (for example, `output = AdminControl.invoke(mbean)`) and then use `print output`. Use the Jython `print` command to convert strings that contain NLS translated characters correctly.

Variable

To assign objects to names, the target of an assignment goes on the left side of an equal sign (=) and the object that you are assigning on the right side. The target on the left side can be a name or object component, and the object on the right side can be an arbitrary expression that computes an object. The following rules exist for assigning objects to names:

- Assignments create object references.
- Names are created when you assign them.
- You must assign a name before referencing it.

Variable name rules are like the rules for the C language; for example, variable names can have an underscore character (_) or a letter plus any number of letters, digits, or underscores.

The following reserved words cannot be used for variable names:

- and
- assert
- break
- class
- continue
- def
- del
- elif
- else
- except

- exec
- finally
- for
- from
- global
- if
- import
- in
- is
- lambda
- not
- or
- pass
- print
- raise
- return
- try
- while

For example:

```
a  = 5
print a
=> 5

b =  a
print b
=> 5

text1, text2, text3, text4  = 'good', 'bad', 'pretty', 'ugly'
print text3
=> pretty
```

The second example assigns the value of variable a to variable b.

Types and operators

The following list contains a few of the built-in object types:

- Numbers. For example:

  ```
  8, 3.133,  999L,  3+4j

  num1 = int(10)
  print num1
  => 10
  ```

- Strings. For example:

  ```
  'name',  "name's", ''

  print str(12345)
  => '12345'
  ```

- Lists. For example:

  ```
  x = [1, [2,  'free'], 5]
  y = [0, 1, 2, 3]
  y.append(5)
  print y
  => [0, 1, 2, 3, 5]

  y.reverse()
  print y
  => [5, 3, 2, 1, 0]

  y.sort()
  print y
  ```

```
=> [0, 1, 2, 3, 5]

print list("apple")
=> ['a', 'p', 'p', 'l', 'e']

print list((1,2,3,4,5))
=> [1, 2, 3, 4, 5]

test = "This is a test"
test.index("test")
=> 10

test.index('s')
=> 3
```

The following list contains a few of the operators:

- x or y

  y is evaluated only if x is false. For example:

  ```
  print 0 or 1
  => 1
  ```

- x and y

  y is evaluated only if x is true. For example:

  ```
  print 0 and 1
  =>  0
  ```

- x + y , x - y

  Addition and concatenation, subtraction. For example:

  ```
  print  6 + 7
  => 13

  text1 = 'Something'
  text2 = ' else'
  print text1 + text2
  => Something else

  list1 = [0, 1, 2, 3]
  list2 = [4, 5, 6, 7]
  print list1 + list2
  => [0, 1, 2, 3, 4, 5, 6, 7]

  print  10 - 5
  => 5
  ```

- x * y, x / y, x % y

  Multiplication and repetition, division, remainder and format. For example:

  ```
  print 5 * 6
  => 30

  print 'test' * 3
  => test test test

  print 30 / 6
  => 5

  print 32 % 6
  => 2
  ```

- x[i], x[i:j], x(...)

  Indexing, slicing, function calls. For example:

  ```
  test = "This is a test"
  print  test[3]
  => s

  print test[3:10]
  => s is a

  print test[5:]
  ```

```
=> is a test

print test[:-4]
=> This is a print len(test)
=> 14
```

- <, <=, >, >=, ==, <>, !=, is is not

  Comparison operators, identity tests. For example:

```
L1 = [1, ('a', 3)]
L2 = [1, ('a', 2)]
L1 < L2, L1 == L2, L1 > L2, L1 <> L2, L1 != L2, L1 is L2, L1 is not L2
=> (0, 0, 1, 1, 1, 0, 1)
```

Backslash substitution

If a statement must span multiple lines, you can also add a back slash (\) at the end of the previous line to indicate you are continuing on the next line. Do not use white space characters, specifically tabs or spaces, after the back slash character. For example:

```
text =  "This is a test of a long lines" \
" continuing lines here."
print text
=> This is a test of a long lines continuing lines here.
```

Functions and scope

Jython uses the `def` statement to define functions. Functions related statements include:

- `def`, `return`

  The `def` statement creates a function `object` and assigns it to a name. The `return` statement sends a result object back to the caller. This is optional, and if it is not present, a function exits so that control flow falls off the end of the function body.

- `global`

  The `global` statement declares module-level variables that are to be assigned. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, list functions in a global statement.

The basic syntax to define a function is the following:

```
def name (arg1, arg2, ... ArgN)
   statements
   return value
```

where *name* is the name of the function being defined. It is followed by an open parenthesis, a close parenthesis, and a colon. The arguments inside parenthesis include a list of parameters to the procedures. The next line after the colon is the body of the function. A group of commands that form the body of the function. After you define a Jython function, it is used just like any of the built-in functions. For example:

```
def intersect(seq1, seq2):
   res = []
   try:
      for x in seq1:
         if x in seq2:
            res.append(x)
   except:
      pass
   return res
```

To call this function, use the following command:

```
s1 = "SPAM"
s2 = "SCAM"
intersect(s1, s2)
```

```
=> [S, A, M]

intersect([1,2,3], (1.4))
=> [1]
```

Comments

Make comments in the Jython language with the pound character (#).

Command-line arguments

The Jython shells pass the command-line arguments to the script as the value of the sys.argv. In wsadmin Jython, the name of the program, or script, is not part of sys.argv. Unlike wsadmin Jython, Jython stand-alone takes the script file as the first argument to the script. Since sys.argv is an array, use the index command to extract items from the argument list. For example, `test.py` takes three arguments a, b, and c.

```
wsadmin -f test.py  a  b  c
```

`test.py` content:

```
import sys
first  =  sys.argv[0]
second = sys.argv[1]
third = sys.argv[2]
arglen = len(sys.argv)
```

Basic statements

There are two looping statements: `while` and `for`. The conditional statement is `if`. The error handling statement is `try`. Finally, there are some statements to fine-tune control flow: `break`, `continue`, and `pass`.

if

The `if` statement selects actions to perform. The `if` statement might contain other statements, including other `if` statements. The `if` statement can be followed by one or more optional `elif` statements and ends with an optional `else` block.

The general format of an if looks like the following:

```
if test1
   statements1
elif test2
   statements2
else
   statements3
```

For example:

```
weather = 'sunny'
if weather == 'sunny':
   print "Nice weather"
elif weather == 'raining':
   print "Bad weather"
else:
   print "Uncertain, don't plan anything"
```

while

The `while` statement consists of a header line with a test expression, a body of one or more indented statements, and an optional `else` statement that runs if control exits the loop without running into a break statement. The `while` statement repeatedly runs a block of indented statements as long as a test at the top keeps evaluating a true value. The general format of a while looks like the following:

```
while test1
   statements1
else
   statements2
```

For example:

```
a = 0; b = 10
while a < b:
   print a
   a = a + 1
```

for

The `for` statement begins with a header line that specifies an assignment target or targets, along with an object you want to step through. The header is followed by a block of indented statements which you want to repeat.

The general format of a for statement looks like the following:

```
for target in object:
   statements
else:
   statements
```

It assigns items in the sequence object to the target, one by one, and runs the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as if it were a cursor stepping through the sequence. For example:

```
sum = 0
for x in [1, 2, 3, 4]:
   sum = sum + x
```

break, continue, and pass

You can control loops with the `break`, `continue` and `pass` statements. The `break` statement jumps out of the closest enclosing loop (past the entire loop statement). The `continue` statements jumps to the top of the closest enclosing loop (to the header line of the loop), and the `pass` statement is an empty statement placeholder.

try

A statement will raise an error if it is called with the wrong number of arguments, or if it detects some error condition particular to its implementation. An uncaught error stops the running of a script. The `try` statement is used to trap such errors. Python `try` statements come in two flavors, one that handles exceptions and one that runs finalization code whether exceptions occur or not. The `try`, `except`, `else` statement starts with a try header line followed by a block of indented statements, then one or more optional except clauses that name exceptions to be caught, and an optional `else` clause at the end. The `try`, `finally` statements starts with a try header line followed by a block of indented statements, then the finally clause that always runs on the way out whether an exception occurred while the try block was running or not.

The general format of the try, except, else function looks like the following:

```
try:
   statements
except name:
   statements
except name, data:
   statements
else
   statements
```

For example:

try: myfunction() except: import sys print 'uncaught exception', sys.exc_info() try: myfilereader() except EOFError: break else: process next line here

The general format of a try and finally looks like the following:

```
try
   statements
finally
   statements
```

For example:

```
def divide(x, y):
   return x / y

def tester(y):
   try:
      print divide(8, y)
   finally:
      print 'on the way out...'
```

The following is a list of syntax rules in Python:

- Statements run one after another until you say otherwise. Statements normally end at the end of the line on which they appear. When statements are too long to fit on a single line you can also add a backslash (\) at the end of the prior line to indicate you are continuing on the next line.
- Block and statement boundaries are detected automatically. There are no braces, or begin or end delimiter, around blocks of code. Instead, the Python language uses the indentation of statements under a header in order to group the statements in a nested block. Block boundaries are detected by line indentation. All statements indented the same distance to the right belong to the same block of code until that block is ended by a line less indented.
- Compound statements = header; ':', indented statements. All compound statements in the Python language follow the same pattern: a header line terminated with a colon, followed by one or more nested statements indented under the header. The indented statements are called a block.
- Spaces and comments are usually ignored. Spaces inside statements and expressions are almost always ignored (except in string constants and indentation), so are comments.

Calling scripts using another script

Use the execfile command to call a Jython script from another Jython script. For example:

Create a script called `test1.py` that contains the following:

Create a script called `testFunctions.py` that contains the following:

```
def printName(first, last):
   name = first + ' ' + last
   return name
```

Then pass the following path as a script argument:

You must use forward slashes (/) as your path separator. Backward slashes (\) do not work.

Chapter 4. Getting started with wsadmin scripting

Scripting is a non-graphical alternative that you can use to configure and manage WebSphere Application Server.

About this task

The WebSphere Application Server wsadmin tool provides the ability to run scripts. The wsadmin tool supports a full range of product administrative activities.

The following figure illustrates the major components involved in a wsadmin scripting solution:

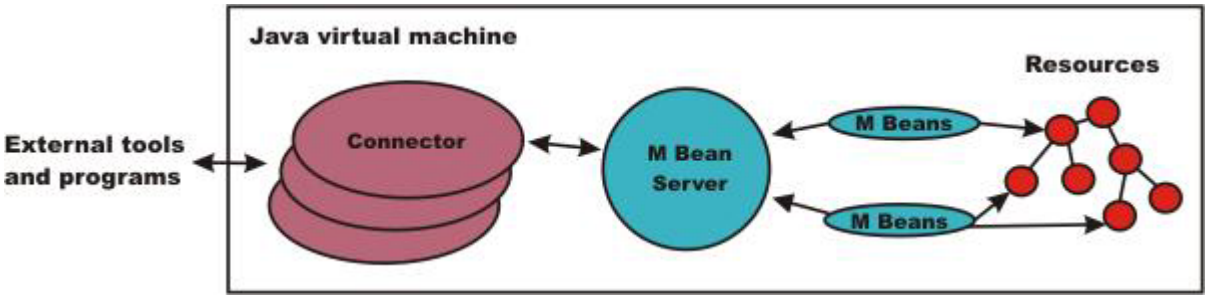


Figure 1: A WebSphere Application Server scripting solution

The wsadmin tool supports two scripting languages: Jacl and Jython. Five objects are available when you use scripts:

- **AdminControl**: Use to run operational commands.
- **AdminConfig**: Use to run configurational commands to create or modify WebSphere Application Server configurational elements.
- **AdminApp**: Use to administer applications.
- **AdminTask**: Use to run administrative commands.
- **Help**: Use to obtain general help.

The scripts use these objects to communicate with MBeans that run in WebSphere Application Server processes. MBeans are Java objects that represent Java Management Extensions (JMX) resources. JMX is an optional package addition to Java 2 Platform Standard Edition (J2SE). JMX is a technology that provides a simple and standard way to manage Java objects.

Important: Some wsadmin scripts, including the AdminApp install, AdminApp update, and some AdminTask commands, require that the user ID under which the server is running must have read permission to the files that are created by the user that is running wsadmin scripting. For example, if the application server is running under user1, but you are running wsadmin scripting under user2, you might encounter exceptions involving a temporary directory. When user2 runs wsadmin scripting to deploy an application, a temporary directory for the enterprise application archive (EAR) file is created. However, when the application server attempts to read and unzip the EAR file as user1, the process fails. It is not recommended that you set the umask value of the user that is running wsadmin scripting to 022 or 023 to work around this issue. This approach makes all of the files that are created by the user readable by other users. To resolve this issue, consider the following approaches based on your administrative policies:

- Run wsadmin scripting with the same user ID as the user that runs the deployment manager or application server. A root user can switch the user ID to complete these actions.

- Set the group ID of the user that is running the deployment manager or application server to be the same group ID as the user that is running wsadmin scripting. Also, set the umask value of the user that is running the wsadmin scripting to be at least a umask 027 value so that files that are created by the wsadmin scripting can be read by members of the group.
- Run wsadmin scripting from a different machine. This approach forces files to be transferred and bypasses the file copy permission issue.

To perform a task using scripting, you must first perform the following steps:

Procedure

1. Choose a scripting language. The wsadmin tool only supports Jacl and Jython scripting languages. Jacl is the language specified by default. If you want to use the Jython scripting language, use the -lang option or specify it in the wsadmin.properties file.
2. Start the wsadmin scripting client interactively, as an individual command, in a script, or in a profile.

What to do next

Before you perform any task using scripting, make sure that you are familiar with the following concepts:

- Java Management Extensions (JMX)
- WebSphere Application Server configuration model
- wsadmin tool
- Jacl syntax or Jython syntax
- Scripting objects

Optionally, you can customize your scripting environment. For more information, see Administrative properties for using wsadmin scripting.

After you become familiar with the scripting concepts, choose a scripting language, and start the scripting client, you are ready to perform tasks using scripting.

What is new for scripted administration (wsadmin)

This topic highlights what is new or changed for users who are going to customize, administer, monitor, and tune production server environments using the wsadmin tool.

The Deprecated, stabilized, and removed features topic describes features that are being replaced or removed in this or future releases.

Improved administrative scripting features

Enhancements to AdminTask command support for managing configurations using properties files

Version 7 introduced system configuration using properties files. Using commands in the PropertiesBasedConfiguration group, you can copy configuration properties from one environment to another, troubleshoot configuration issues, and apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Enhancements to configuration using properties files for Version 8.0 include the following:

- Support for web services endpoint URL fragment properties files

For more information, see Using properties files to manage system configuration and PropertiesBasedConfiguration command group for the AdminTask object using wsadmin scripting.

AdminSDKCmds command group for the AdminTask object

Use the commands and parameters in the AdminSDKCmds group for the AdminTask object to perform the following actions:

- List software development kits that are not used by a node.
- Get or set the default SDK for a node.
- Get or set an SDK for a server.

For more information, see AdminSDKCmds command group for the AdminTask object.

getAvailableSDKsOnNode and getSDKPropertiesOnNode commands in the ManagedObjectMetadata command group for the AdminTask object

Use the getAvailableSDKsOnNode and getSDKPropertiesOnNode commands and parameters in the ManagedObjectMetadata group for the AdminTask object to perform the following actions:

- List software development kits that are installed with the product and available for use by a node.
- List software development kit (SDK) properties.

For more information, see ManagedObjectMetadata command group for the AdminTask object.

listServices command for information about web service references

Use the listServices command to learn about service references. You can use the serviceRef property with the queryProps parameter with the listServices command to query all service references or a specific service reference. This parameter is only applicable for service clients. If you specify an asterisk (*) as a wildcard as the name of the service reference, all of the service references for the matching service client are returned. You can also query a specific service reference name by specifying the name of the service reference that you want. To return detailed service reference information for endpoints and operations, specify the expandResource property.

For more information, see Querying web services using wsadmin scripting.

Security cookies set as HTTPOnly resist cross-site scripting attacks	Use the HttpOnly browser attribute to prevent client side applications (such as Java scripts) from accessing cookies to prevent some cross-site scripting vulnerabilities. The attribute specifies that LTPA and WASReqURL cookies include the HTTPOnly field. For more information, see Single sign-on settings.

Overview and new features for scripting the application serving environment

Use the links provided in this topic to learn about the administrative features.

"What is new for scripted administration (wsadmin)" on page 28

This topic provides an overview of new and changed features for administrative scripting and the wsadmin tool.

Introduction: Administrative scripting (wsadmin)

This topic provides an introduction to administrative scripting and the wsadmin tool.

Chapter 5. Using the wsadmin scripting objects

The wsadmin tool utilizes a set of management objects which allow you to execute commands and command parameters to configure your environment. Use the AdminConfig, AdminControl, AdminApp, AdminTask, and Help objects to perform administrative tasks.

About this task

Each of the management objects have commands that you can use to perform administrative tasks. To use the scripting objects, specify the scripting object, a command, and command parameters. In the following example, `AdminConfig` is the scripting object, `attributes` is the command, and `ApplicationServer` is the command parameter.

Using Jython:

```
print AdminConfig.attributes('ApplicationServer')
```

Using Jacl:

```
$AdminConfig attributes ApplicationServer
```

Administrative functions within the application server are divided into two categories: functions that work with the configuration of application server installations, and functions that work with the currently running objects on application server installations. Scripts work with both categories of objects. For example, an application server is divided into two distinct entities. One entity represents the configuration of the server that resides persistently in a repository on permanent storage.

Procedure

- Use the **AdminConfig object**, the **AdminTask object**, and the **AdminApp object** to handle configuration functionality.

  The **AdminConfig object**, the **AdminTask object**, and the **AdminApp object** are used when you are managing the configuration of the server that resides persistently in a repository on permanent storage. Use these objects to create, query, change, or remove this configuration without starting an application server process. To use the **AdminTask object**, you must be connected to a running server.
- Use the **AdminControl object** to manage running objects on application server installations.

  The **AdminControl object** is used when managing the running instance of an application server by a *Java Management Extensions (JMX) MBean*. This instance can have attributes that you can interrogate and change, and operations that you can invoke. These operational actions that are taken against a running application server do not have an effect on the persistent configuration of the server. The attributes that support manipulation from an MBean differ from the attributes that the corresponding configuration supports. The configuration can include many attributes that you cannot query or set from the running object. The application server scripting support provides functions to locate configuration objects and running objects. The objects in the configuration do not always represent objects that are currently running. The **AdminControl object** manages running objects.
- Use the **Help Object** to obtain information about the AdminConfig, AdminApp, AdminControl, and AdminTask objects, to obtain interface information about running MBeans, and to obtain help for warnings and error messages.

Help object for scripted administration using wsadmin scripting

The Help object provides general help, online information about running MBeans, and help on messages.

Use the Help object to obtain general help for the other objects supplied by the wsadmin tool for scripting: the AdminApp, AdminConfig, AdminTask, and AdminControl objects. For example, using Jacl, `$Help AdminApp` or using Jython, `Help.Adminapp()`, provides information about the AdminApp object and the available commands.

The Help object also to provides interface information about MBeans running in the system. The commands that you use to get online information about the running MBeans include: all, attributes, classname, constructors, description, notification, operations.

You can also use the Help object to obtain information about messages using the message command. The message command provides aid to understand the cause of a warning or error message and find a solution for the problem. For example, you receive a WASX7115E error when running the AdminApp install command to install an application, use the following example:

Using Jacl:

```
$Help message WASX7115E
```

Using Jython:

```
print Help.message('WASX7115E')
```

Example output:

```
Explanation: wsadmin failed to read an ear file when
preparing to copy it to a temporary location for AdminApp
processing.  User action: Examine the wsadmin.traceout
log file to determine the problem; there may be file permission problems.
```

The user action specifies the recommended action to correct the problem. It is important to understand that in some cases the user action may not be able to provide corrective actions to cover all the possible causes of an error. It is an aid to provide you with information to troubleshoot a problem.

To see a list of all available commands for the Help object, see the Commands for the Help object topic or use the **Help** command, for example:

Using Jacl:

```
$Help help
```

Using Jython:

```
print Help.help()
```

Chapter 6. Using the wsadmin scripting AdminApp object for scripted administration

Use the AdminApp object to manage applications.

Before you begin

This object communicates with the run time application management object in WebSphere Application Server to make application inquires and changes, for example:

- Installing and uninstalling applications
- Listing applications
- Editing applications or modules

Because applications are part of configuration data, any changes that you make to an application are kept in the configuration session, similar to other configuration data. Be sure to save your application changes so that the data transfers from the configuration session to the master repository.

About this task

With the application already installed, the AdminApp object can update application metadata, map virtual hosts to web modules, and map servers to modules. You must perform any other changes, such as specifying a library for the application to use or setting session management configuration properties, using the AdminConfig object.

You can run the commands for the AdminApp object in local mode. If a server is running, it is not recommended that you run the scripting client in local mode because any configuration changes that are made in local mode will not be reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

To see a list of all available commands for the AdminApp object:

Procedure

- See the Commands for the AdminApp object topic.
- You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminApp help
  ```

  Using Jython:

  ```
  print AdminApp.help()
  ```

Listing applications using the wsadmin scripting tool

You can list installed applications using the wsadmin tool and scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin scripting client.

Procedure

- Query the configuration and create a list of installed applications, for example:
  - Using Jacl:

```
$AdminApp list
```

– Using Jython:

```
print AdminApp.list()
```

Table 4. AdminApp list command description. Run the list command with no arguments.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`list`	is an AdminApp command

Example output:

```
DefaultApplication
SampleApp
app1serv2
```

- Query the configuration and create a list of installed applications on a given target scope, for example:
  – Using Jacl:

```
$AdminApp list WebSphere:cell=myCell,node=myNode,server=myServer
```

  – Using Jython:

```
print AdminApp.list("WebSphere:cell=myCell,node=myNode,server=myServer")
```

Table 5. AdminApp list command with target description. Run the list command with an optional argument.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`list`	is an AdminApp command
`WebSphere:cell=myCell,node=myNode, server=myServer`	is an optional target scope

Example output:

```
DefaultApplication
PlantsByWebSphere
SamplesGallery
ivtApp
query
```

Editing application configurations using the wsadmin scripting tool

Use the wsadmin tool to configure application settings.

About this task

You can use the AdminApp edit or editInteractive command to change an entire application or a single application module.

You can set or update a configuration value using options in batch mode. To identify which configuration object is to be set or updated, the values of read only fields are used to find the corresponding configuration object. All the values of read only fields have to match with an existing configuration object, otherwise the command fails.

You can use pattern matching to simplify the task of supplying required values for certain complex options. Pattern matching only applies to fields that are required or read only.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Procedure

1. Start the wsadmin scripting tool.
2. Edit the entire application or a single application module. Use one of the following commands:
   - The following command uses the installed application and the command option information to edit the application:
     - Using Jacl:

```
$AdminApp edit appname {options}
```

     - Using Jython list:

```
AdminApp.edit('appname', ['options'])
```

     - Using Jython string:

```
AdminApp.edit('appname', '[options]')
```

Table 6. AdminApp edit command description. Run the edit command with the name of the application or module.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`edit`	is an AdminApp command
appname	is the name of application or application module to edit. For the application module name, use the module name returned from **listModules** command as the value.
`{options}`	is a list of edit options and tasks similar to the ones for the install command

   - The following command changes the application information by prompting you through a series of editing tasks:
     - Using Jacl:

```
$AdminApp editInteractive appname
```

     - Using Jython:

```
AdminApp.editInteractive('appname')
```

Table 7. AdminApp editInteractive command description. Run the editInteractive command with the name of the application or module.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object that supports application object management
`editInteractive`	is an AdminApp command
appname	is the name of application or application module to edit. For the application module name, use the module name returned from **listModules** command as the value.

3. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Chapter 7. Using the wsadmin scripting AdminControl object for scripted administration

The AdminControl scripting object is used for operational control. It communicates with MBeans that represent live objects running a WebSphere server process.

Before you begin

It includes commands to query existing running objects and their attributes and invoke operation on the running objects. In addition to the operational commands, the AdminControl object supports commands to query information on the connected server, convenient commands for client tracing, reconnecting to a server, and start and stop server for network deployment environment.

About this task

Many of the operational commands have two sets of signatures so that they can either invoke using string based parameters or using Java Management Extension (JMX) objects as parameters. Depending on the server process to which a scripting client is connected, the number and type of MBeans available varies. When connected to an application server, only MBeans running in that application server are visible.

The following steps provide a general method to manage the cycle of an application:

- Install the application.
- Edit the application.
- Update the application.
- Uninstall the application.

To see a list of all available commands for the AdminControl object:

Procedure

- See the Commands for the AdminControl object topic.
- You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminControl help
  ```

  Using Jython:

  ```
  print AdminControl.help()
  ```

ObjectName, Attribute, and AttributeList classes using wsadmin scripting

WebSphere Application Server scripting commands use the underlying Java Management Extensions (JMX) classes, ObjectName, Attribute, and AttributeList, to manipulate object names, attributes and attribute lists respectively.

The ObjectName class uniquely identifies running objects. The ObjectName class consists of the following elements:

- The domain name `WebSphere`.
- Several key properties, for example:
  - `type` indicates the type of object that is accessible through the MBean, for example, ApplicationServer, and EJBContainer.
  - `name` represents the display name of the particular object, for example, MyServer.
  - `node` represents the name of the node on which the object runs.

– process represents the name of the server process in which the object runs.
– mbeanIdentifier correlates the MBean instance with corresponding configuration data.

When ObjectName classes are represented by strings, they have the following pattern:

```
[domainName]:property=value[,property=value]*
```

For example, you can specify WebSphere:name="My Server",type=ApplicationServer,node=n1,* to specify an application server named My Server on node n1. (The asterisk (*) is a wildcard character, used so that you do not have to specify the entire set of key properties.) The AdminControl commands that take strings as parameters expect strings that look like this example when specifying running objects (MBeans). You can obtain the object name for a running object with the getObjectName command.

Attributes of these objects consist of a name and a value. You can extract the name and value with the getName and the getValue methods that are available in the javax.management.Attribute class. You can also extract a list of attributes.

Example: Collecting arguments for the AdminControl object using wsadmin scripting

This example shows how to use multiple arguments with the AdminControl object.

Verify that the arguments parameter is a single string. Each individual argument in the string can contain spaces. Collect each argument that contains spaces in some way.

- An example of how to obtain an MBean follows:

  Using Jacl:

  ```
  set am [$AdminControl queryNames type=ApplicationManager,process=server1,*]
  ```

  Using Jython:

  ```
  am = AdminControl.queryNames('type=ApplicationManager,process=server1,*')
  ```

- Multiple ways exist to collect arguments that contain spaces. Choose one of the following alternatives:

  Using Jacl:
  – `$AdminControl invoke $am startApplication {"JavaMail Sample"}`
  – `$AdminControl invoke $am startApplication {{JavaMail Sample}}`
  – `$AdminControl invoke $am startApplication "\"JavaMail Sample\""`

  Using Jython:
  – `AdminControl.invoke(am, 'startApplication', '[JavaMail Sample]')`
  – `AdminControl.invoke(am, 'startApplication', '\"JavaMail Sample\"')`

Example: Identifying running objects using wsadmin scripting

Use the AdminControl object to interact with running MBeans.

In the WebSphere Application Server, MBeans represent running objects. You can interrogate the MBean server to see the objects it contains.

- Use the **queryNames** command to see running MBean objects. For example:

  Using Jacl:

  ```
  $AdminControl queryNames *
  ```

  Using Jython:

  ```
  print AdminControl.queryNames('*')
  ```

  This command returns a list of all MBean types. Depending on the server to which your scripting client attaches, this list can contain MBeans that run on different servers:
  – If the client attaches to a stand-alone WebSphere Application Server, the list contains MBeans that run on that server.

- The list that the `queryNames` command returns is a string representation of JMX `ObjectName` objects. For example:

  ```
  WebSphere:cell=MyCell,name=TraceService,mbeanIdentifier=TraceService,
  type=TraceService,node=MyNode,process=server1
  ```

  This example represents a `TraceServer` object that runs in *server1* on *MyNode*.
- The single queryNames argument represents the `ObjectName` object for which you are searching. The asterisk ("*") in the example means return all objects, but it is possible to be more specific. As shown in the example, `ObjectName` has two parts: a domain, and a list of key properties. For MBeans created by the WebSphere Application Server, the domain is WebSphere. If you do not specify a domain when you invoke queryNames, the scripting client assumes the domain is WebSphere. This means that the first example query above is equivalent to:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:*
  ```

  Using Jython:

  ```
  AdminControl.queryNames('WebSphere:*')
  ```
- WebSphere Application Server includes the following key properties for the `ObjectName` object:
  - name
  - type
  - cell
  - node
  - process
  - mbeanIdentifier

  These key properties are common. There are other key properties that exist. You can use any of these key properties to narrow the scope of the **queryNames** command. For example:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:type=Server,node=myNode,*
  ```

  Using Jython:

  ```
  AdminControl.queryNames('WebSphere:type=Server,node=myNode,*')
  ```

  This example returns a list of all MBeans that represent server objects running the node *myNode*. The, `*` at the end of the `ObjectName` object is a JMX wildcard designation. For example, if you enter the following:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:type=Server,node=myNode
  ```

  Using Jython:

  ```
  print AdminControl.queryNames('WebSphere:type=Server,node=myNode')
  ```

  you get an empty list back because the argument to queryNames is not a wildcard. There is no Server MBean running that has exactly these key properties and no others.
- If you want to see all the MBeans representing applications running on a particular node, invoke the following example:

  Using Jacl:

  ```
  $AdminControl queryNames WebSphere:type=Application,node=myNode,*
  ```

  Using Jython:

  ```
  print AdminControl.queryNames('WebSphere:type=Application,node=myNode,*')
  ```

Specifying running objects using the wsadmin scripting tool

Use scripting and the wsadmin tool to specify running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin scripting client.

About this task

You can run wsadmin commands that obtain object names and specify running objects.

Procedure

1. Obtain the configuration ID with one of the following ways:
   - Obtain the object name with the **completeObjectName** command, for example:
     – Using Jacl:

       ```
       set var [$AdminControl completeObjectName template]
       ```

     – Using Jython:

       ```
       var = AdminControl.completeObjectName(template)
       ```

Table 8. AdminConfig completeObjectName command description. Run the completeObjectName command with the template.

`set`	is a Jacl command
`var`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`completeObjectName`	is an AdminControl command
`template`	is a string containing a segment of the object name to be matched. The template has the same format as an object name with the following pattern: `[domainName]:property=value[,property=value]*`. See Object name, Attribute, Attribute list for more information.

If there are several MBeans that match the template, the **completeObjectName** command only returns the first match. The matching MBean object name is then assigned to a variable.

To look for *server1* MBean in *mynode*, use the following example:

– Using Jacl:

```
set server1 [$AdminControl completeObjectName node=mynode,type=Server,name=server1,*]
```

– Using Jython:

```
server1 = AdminControl.completeObjectName('node=mynode,type=Server,name=server1,*')
```

- Obtain the object name with the **queryNames** command, for example:
  – Using Jacl:

    ```
    set var [$AdminControl queryNames template]
    ```

  – Using Jython:

    ```
    var = AdminControl.queryNames(template)
    ```

Table 9. AdminControl queryNames command description. Run the queryNames command with the template.

`set`	is a Jacl command
`var`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application server process.
`queryNames`	is an AdminControl command

Table 9. AdminControl queryNames command description (continued). Run the queryNames command with the template.

`template`	is a string containing a segment of the object name to be matched. The template has the same format as an object name with the following pattern: `[domainName]:property=value[,property=value]*`

2. If there are more than one running objects returned from the **queryNames** command, the objects are returned in a list syntax. One simple way to retrieve a single element from the list is to use the **lindex** command in Jacl and **split** command in Jython. The following example retrieves the first running object from the server list:
   - Using Jacl:

     ```
     set allServers [$AdminControl queryNames type=Server,*]
     set aServer [lindex $allServers 0]
     ```

   - Using Jython:

     ```
     allServers = AdminControl.queryNames('type=Server,*')

     # get line separator
     import  java
     lineSeparator = java.lang.System.getProperty('line.separator')

     aServer = allServers.split(lineSeparator)[0]
     ```

   For other ways to manipulate the list and then perform pattern matching to look for a specified configuration object, refer to the topic on Jacl syntax.

Results

You can now use the running object in with other AdminControl commands that require an object name as a parameter.

Identifying attributes and operations for running objects using the wsadmin scripting tool

You can use scripting to identify attributes and operations for running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin scripting client.

About this task

Use the **attributes** or **operations** commands of the Help object to find information on a running MBean in the server.

Procedure

1. Specify a running object.
2. Use the **attributes** command to display the attributes of the running object:
   - Using Jacl:

     ```
     $Help attributes MBeanObjectName
     ```

   - Using Jython:

     ```
     Help.attributes(MBeanObjectName)
     ```

Table 10. Help attributes command description. Run the attributes command with an object name.

`$`	is a Jacl operator for substituting a variable name with its value

Table 10. Help attributes command description (continued). Run the attributes command with an object name.

`Help`	is the object that provides general help and information for running MBeans in the connected server process
`attributes`	is a Help command
`MBeanObjectName`	is the string representation of the MBean object name that is obtained in step 2

3. Use the **operations** command to find out the operations that are supported by the MBean:
   - Using Jacl:

     ```
     $Help operations MBeanObjectname
     ```

     or

     ```
     $Help operations MBeanObjectname operationName
     ```
   - Using Jython:

     ```
     Help.operations(MBeanObjectname)
     ```

     or

     ```
     Help.operations(MBeanObjectname, operationName)
     ```

Table 11. Help operations command description. Run the operations command with an object name and, optionally, an operation name.

`$`	is a Jacl operator for substituting a variable name with its value
`Help`	is the object that provides general help and information for running MBeans in the connected server process
`operations`	is a Help command
`MBeanObjectname`	is the string representation of the MBean object name that is obtained in step number 2
`operationName`	(optional) is the specified operation from which you want to obtain detailed information

If you do not provide the operationName value, all the operations that are supported by the MBean return with the signature for each operation. If you specify the operationName value, only the operation that you specify returns and it contains details which include the input parameters and the return value.

To display the operations for the server MBean, use the following example:

- Using Jacl:

  ```
  set server [$AdminControl completeObjectName type=Server,name=server1,*]
  $Help operations $server
  ```
- Using Jython:

  ```
  server = AdminControl.completeObjectName('type=Server,name=server1,*')
  print Help.operations(server)
  ```

To display detailed information about the stop operation, use the following example:

- Using Jacl:

  ```
  $Help operations $server stop
  ```
- Using Jython:

  ```
  print Help.operations(server, 'stop')
  ```

Performing operations on running objects using the wsadmin scripting tool

You can use scripting to invoke operations on running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin scripting client.

About this task

You can run wsadmin commands that obtain the object names of running objects and perform operations:

Procedure

1. Obtain the object name of the running object. For example:
   - Using Jacl:

     ```
     $AdminControl completeObjectName name
     ```
   - Using Jython:

     ```
     AdminControl.completeObjectName(name)
     ```

Table 12. AdminControl completeObjectName command description. Run the completeObjectName command with an object name.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
name	is a fragment of the object name. It is used to find the matching object name. For example: `type=Server,name=serv1,*`. It can be any valid combination of domain and key properties. For example, type, name, cell, node, process, etc.

2. Set the s1 variable to the running object, for example:
   - Using Jacl:

     ```
     set s1 [$AdminControl completeObjectName type=Server,name=server1,*]
     ```
   - Using Jython:

     ```
     s1 = AdminControl.completeObjectName('type=Server,name=server1,*')
     ```

Table 13. AdminControl completeObjectName with type command description. Run the completeObjectName command with an object type and name.

`set`	is a Jacl command
`s1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
`type`	is the object name property key
Server	is the name of the object
`name`	is the object name property key
server1	is the name of the server where the operation is invoked

3. Invoke the operation. For example:
   - Using Jacl:

     ```
     $AdminControl invoke $s1 stop
     ```
   - Using Jython:

     ```
     AdminControl.invoke(s1, 'stop')
     ```

Table 14. AdminControl invoke command description. Run the invoke command with the server identifier and stop operation.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`invoke`	is an AdminControl command
`s1`	is the ID of the server that is specified in step number 3
`stop`	is an operation to invoke on the server

Example

The following example is for operations that require parameters:

- Using Jacl:

```
set traceServ [$AdminControl completeObjectName type=TraceService,process=server1,*]
$AdminControl invoke $traceServ appendTraceString "com.ibm.ws.management.*=all=enabled"
```

- Using Jython:

```
traceServ = AdminControl.completeObjectName('type=TraceService,process=server1,*')
AdminControl.invoke(traceServ, 'appendTraceString',  "com.ibm.ws.management.*=all=enabled")
```

Modifying attributes on running objects using the wsadmin scripting tool

Use scripting and the wsadmin tool to modify attributes on running objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin scripting client.

About this task

You can run a script that modifies attributes on running objects.

Procedure

1. Obtain the name of the running object.

   Run the completeObjectName command with the `name` parameter.

   - Using Jacl:

     `$AdminControl completeObjectName` *`name`*

   - Using Jython:

     `AdminControl.completeObjectName(`*`name`*`)`

Table 15. AdminControl completeObjectName command description. Run the completeObjectName command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans that run in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command

Table 15. AdminControl completeObjectName command description (continued). Run the completeObjectName command from a wsadmin command line.

name	is a fragment of the object name that is used to find the matching object name. For example: `type=TraceService,node=mynode,*`. This value can be any valid combination of domain and key properties, for example, type, name, cell, node, process, and so on.

2. Set the ts1 variable to the running object.

   The following scripts set the ts1 variable to the result of the completeObjectName commands.

   - Using Jacl:

     ```
     set ts1 [$AdminControl completeObjectName name]
     ```

   - Using Jython:

     ```
     ts1 = AdminControl.completeObjectName(name)
     ```

Table 16. AdminControl completeObjectName command description. Set the result of a completeObjectName command to a variable.

`set`	is a Jacl command
`ts1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
name	is a fragment of the object name. It is used to find the matching object name. For example: `type=TraceService,node=mynode,*`. It can be any valid combination of domain and key properties, for example, type, name, cell, node, process, and so on.

3. Modify the running object.

   The following scripts use the setAttribute command to set the ts1 variable ring buffer size to 10.

   - Using Jacl:

     ```
     $AdminControl setAttribute $ts1 ringBufferSize 10
     ```

   - Using Jython:

     ```
     AdminControl.setAttribute(ts1, 'ringBufferSize', 10)
     ```

Table 17. AdminControl setAttribute command description. Run the setAttribute command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`setAttribute`	is an AdminControl command
`ts1`	evaluates to the ID of the server specified in step number 3
`ringBufferSize`	is an attribute of modify objects
10	is the value of the ringBufferSize attribute

   You can also modify multiple attribute name and value pairs, for example:

   - Using Jacl:

     ```
     set ts1 [$AdminControl completeObjectName type=TraceService,process=server1,*]
     $AdminControl setAttributes $ts1 {{ringBufferSize 10}
      {traceSpecification com.ibm.*=all=disabled}}
     ```

   - Using Jython list:

```
ts1 = AdminControl.completeObjectName('type=TraceService,process=server1,*')
AdminControl.setAttributes(ts1, [['ringBufferSize', 10],
 ['traceSpecification',  'com.ibm.*=all=disabled']])
```

- Using Jython string:

```
ts1 =AdminControl.completeObjectName('type=TraceService,process=server1,*')
AdminControl.setAttributes(ts1, '[[ringBufferSize 10]
 [traceSpecification  com.ibm.*=all=disabled]]')
```

The new attribute values are returned to the command line.

Chapter 8. Using the wsadmin scripting AdminConfig object for scripted administration

Use the AdminConfig object to manage the configuration information that is stored in the repository.

Before you begin

This object communicates with the WebSphere Application Server configuration service component to make configuration inquires and changes. You can use it to query existing configuration objects, create configuration objects, modify existing objects, remove configuration objects, and obtain help.

Updates to the configuration through a scripting client are kept in a private temporary area called a workspace and are not copied to the master configuration repository until you run a **save** command. The workspace is a temporary repository of configuration information that administrative clients including the administrative console use. The workspace is kept in the `wstemp` subdirectory of your WebSphere Application Server installation. The use of the workspace allows multiple clients to access the master configuration. If the same update is made by more than one client, it is possible that updates made by a scripting client will not save because there is a conflict. If this occurs, the updates will not be saved in the configuration unless you change the default save policy with the **setSaveMode** command.

About this task

The AdminConfig commands are available in both connected and local modes. If a server is currently running, it is not recommended that you run the scripting client in local mode because the configuration changes made in the local mode is not reflected in the running server configuration and vice versa. In connected mode, the availability of the AdminConfig commands depend on the type of server to which a scripting client is connected in a WebSphere Application Server, Network Deployment installation.

Procedure

- Query and update a configuration object.
  1. Identify the configuration type and the corresponding attributes.
  2. Query an existing configuration object to obtain a configuration ID to use.
  3. Modify the existing configuration object or create a one.
  4. Save the configuration.
- See the Commands for the AdminConfig object topic. You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminConfig help
  ```

  Using Jython:

  ```
  print AdminConfig.help()
  ```

Creating configuration objects using the wsadmin scripting tool

You can use scripting and the wsadmin tool to create configuration objects.

About this task

Perform this task if you want to create an object. To create new objects from the default template, use the **create** command. Alternatively, you can create objects using an existing object as a template with the **createUsingTemplate** command. You can only use the **createUsingTemplate** command for creation of a server with `APPLICATION_SERVER` type. If you want to create a server with a type other than `APPLICATION_SERVER`, use the **createGenericServer** or the **createWebServer** command.

Procedure

1. Start the wsadmin scripting tool.
2. Use the AdminConfig object **listTemplates** command to list available templates:
   - Using Jacl:

```
$AdminConfig listTemplates JDBCProvider
```

   - Using Jython:

```
AdminConfig.listTemplates('JDBCProvider')
```

Table 18. AdminConfig listTemplates command description. Run a command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`listTemplates`	is an AdminConfig command
`JDBCProvider`	is an object type

3. Assign the ID string that identifies the existing object to which the new object is added. You can add the new object under any valid object type. The following example uses a node as the valid object type:
   - Using Jacl:

```
set n1 [$AdminConfig getid /Node:mynode/]
```

   - Using Jython:

```
n1 =  AdminConfig.getid('/Node:mynode/')
```

Table 19. AdminConfig getid command description. Run a command from a wsadmin command line.

`set`	is a Jacl command
`$`	is a Jacl operator for substituting a variable name with its value
`n1`	is a variable name
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is an object type
`mynode`	is the name of the node where the new object is added

4. Specify the template that you want to use:
   - Using Jacl:
   - Using Jython:

Table 20. AdminConfig listTemplates command description. Run a command from a wsadmin command line.

`set`	is a Jacl command
`$`	is a Jacl operator for substituting a variable name with its value
`t1`	is a variable name
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`listTemplates`	is an AdminConfig command
`JDBCProvider`	is an object type
`DB2® JDBC Provider (XA)`	is the name of the template to use for the new object

If you supply a string after the name of a type, you get back a list of templates with display names that contain the string you supplied. In this example, the AdminConfig **listTemplates** command returns the JDBCProvider template whose name matches *DB2 JDBC Provider (XA)*. This example assumes that the variable that you specify here only holds one template configuration ID. If the environment contains multiple templates with the same string, for example, *DB2 JDBC Provider (XA)*, the variable will hold the configuration IDs of all of the templates. Be sure to identify the specific template that you want to

use before you perform the next step, creating an object using a template.

5. Create the object with the following command:
   - Using Jacl:

```
$AdminConfig createUsingTemplate JDBCProvider $n1 {{name newdriver}} $tl
```

   - Using Jython:

```
AdminConfig.createUsingTemplate('JDBCProvider', n1, [['name', 'newdriver']], t1)
```

Table 21. AdminConfig createUsingTemplate command description. Run a command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
createUsingTemplate	is an AdminConfig command
JDBCProvider	is an object type
n1	evaluates the ID of the host node that is specified in step number 3
name	is an attribute of JDBCProvider objects
newdriver	is the value of the name attribute
t1	evaluates the ID of the template that is specified in step number 4

   All **create** commands use a template unless there are no templates to use. If a default template exists, the command creates the object.

6. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Interpreting the output of the AdminConfig attributes command using wsadmin scripting

Use scripting to interpret the output of the AdminConfig attributes command.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin scripting client using wsadmin scripting.

About this task

The **attributes** command is a wsadmin tool on-line help feature. When you issue the **attributes** command, the information that displays does not represent a particular configuration object. It represents information about configuration object types, or object metadata. This article discusses how to interpret the attribute type display.

Procedure

- Simple attributes

  Using Jacl:

```
$AdminConfig attributes ExampleType1
"attr1 String"
```

  Using Jython:

```
print AdminConfig.attributes('ExampleType1')
attr1 String
```

  Types do not display as fully qualified names. For example, `String` is used for `java.lang.String`. There are no ambiguous type names in the model. For example, `x.y.ztype` and `a.b.ztype`. Using only the final portion of the name is possible, and it makes the output easier to read.

- Multiple attributes

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType2
  "attr1 String" "attr2 Boolean" "attr3 Integer"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType2')
  attr1 String attr2 Boolean attr3 Integer
  ```

  All input and output for the scripting client takes place with strings, but `attr2 Boolean` indicates that `true` or `false` are appropriate values. The `attr3 Integer` indicates that string representations of integers ("42") are needed. Some attributes have string values that can take only one of a small number of predefined values. The wsadmin tool distinguishes these values in the output by the special type name ENUM, for example:

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType3
  "attr4 ENUM(ALL, SOME, NONE)"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType3')
  attr4 ENUM(ALL, SOME, NONE)
  ```

  where: `attr4` is an ENUM type. When you query or set the attribute, one of the values is `ALL`, `SOME`, or `NONE`. The value `A_FEW` results in an error.

- Nested attributes

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType4
  "attr5 String" "ex5 ExampleType5"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType4')
  attr5 String ex5 ExampleType5
  ```

  The `ExampleType4` object has two attributes: a string, and an `ExampleType5` object. If you do not know what is contained in the ExampleType5 object, you can use another **attributes** command to find out. The **attributes** command displays only the attributes that the type contains directly. It does not recursively display the attributes of nested types.

- Attributes that represent lists

  The values of these attributes are object lists of different types. The `*` character distinguishes these attributes, for example:

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType5
  "ex6 ExampleType6*"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType5')
  ex6 ExampleType6*
  ```

  In this example, objects of the `ExampleType5` type contain a single attribute, `ex6`. The value of this attribute is a list of `ExampleType6` type objects.

- Reference attributes

  An attribute value that references another object. You cannot change these references using modify commands, but these references display because they are part of the complete representation of the type. Distinguish reference attributes using the `@` sign, for example:

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType6
  "attr7 Boolean" "ex7 ExampleType7@"
  ```

  Using Jython:

```
print AdminConfig.attributes('ExampleType6')
attr7 Boolean ex7 ExampleType7@
```

ExampleType6 objects contain references to ExampleType7 type objects.

- Generic attributes

  These attributes have generic types. The values of these attributes are not necessarily this generic type. These attributes can take values of several different specific types. When you use the AdminConfig attributes command to display the attributes of this object, the various possibilities for specific types are shown in parentheses, for example:

  Using Jacl:

  ```
  $AdminConfig attributes ExampleType8
  "name String" "beast AnimalType(HorseType, FishType, ButterflyType)"
  ```

  Using Jython:

  ```
  print AdminConfig.attributes('ExampleType8')
  name String beast AnimalType(HorseType, FishType, ButterflyType)
  ```

  In this example, the beast attribute represents an object of the generic AnimalType. This generic type is associated with three specific subtypes. The wsadmin tool gives these subtypes in parentheses after the name of the base type. In any particular instance of ExampleType8, the beast attribute can have a value of HorseType, FishType, or ButterflyType. When you specify an attribute in this way, using a modify or create command, specify the type of AnimalType. If you do not specify the AnimalType, a generic AnimalType object is assumed (specifying the generic type is possible and legitimate). This is done by specifying beast:HorseType instead of beast.

Specifying configuration objects using the wsadmin scripting tool

Specify configuration objects with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Starting the wsadmin scripting client topic.

About this task

To manage an existing configuration object, identify the configuration object and obtain a configuration ID of the object to use for subsequent manipulation.

Procedure

1. Obtain the configuration ID in one of the following ways:
   - Obtain the ID of the configuration object with the **getid** command, for example:
     - Using Jacl:

```
set var [$AdminConfig getid /type:name/]
```

     - Using Jython:

```
var = AdminConfig.getid('/type:name/')
```

Table 22. AdminConfig getid command description. Run a command from a wsadmin command line.

set	is a Jacl command
var	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
getid	is an AdminConfig command
/type:*name*/	is the hierarchical containment path of the configuration object

Table 22. AdminConfig getid command description (continued). Run a command from a wsadmin command line.

`type`	is the object type. The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that is displayed in the administrative console.
name	is the optional name of the object

You can specify multiple `/type:name/` value pairs in the string, for example, `/type:name/type:name/type:name/`. If you specify the type in the containment path without the name, include the colon, for example, `/type:/`. The containment path must be a path that contains the correct hierarchical order. For example, if you specify `/Server:server1/Node:node/` as the containment path, you do not receive a valid configuration ID because `Node` is a parent of `Server` and comes before `Server` in the hierarchy.

This command returns all the configuration IDs that match the representation of the containment and assigns them to a variable.

To look for all the server configuration IDs that reside in the `mynode` node, use the code in the following example:

– Using Jacl:

```
set nodeServers [$AdminConfig getid /Node:mynode/Server:/]
```

– Using Jython:

```
nodeServers = AdminConfig.getid('/Node:mynode/Server:/')
```

To look for the server1 configuration ID that resides in mynode, use the code in the following example:

– Using Jacl:

```
set server1 [$AdminConfig getid /Node:mynode/Server:server1/]
```

– Using Jython:

```
server1 = AdminConfig.getid('/Node:mynode/Server:server1/')
```

To look for all the server configuration IDs, use the code in the following example:

– Using Jacl:

```
set servers [$AdminConfig getid /Server:/]
```

– Using Jython:

```
servers = AdminConfig.getid('/Server:/')
```

- Obtain the ID of the configuration object with the **list** command, for example:

  – Using Jacl:

```
set var [$AdminConfig list type]
```

or

```
set var [$AdminConfig list type scopeId]
```

  – Using Jython:

```
var = AdminConfig.list('type')
```

or

```
var = AdminConfig.list('type', 'scopeId')
```

Table 23. AdminConfig list command description. Run a command from a wsadmin command line.

`set`	is a Jacl command
`var`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`list`	is an AdminConfig command
`type`	is the object type. The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that is displayed in the administrative console.

Table 23. AdminConfig list command description (continued). Run a command from a wsadmin command line.

scopeId	is the configuration ID of a cell, a node, or a server object

This command returns a list of configuration object IDs of a given type. If you specify the *scopeId* value, the list of objects is returned within the specified scope. The returned list is assigned to a variable.

To look for all the server configuration IDs, use the following example:

– Using Jacl:

```
set servers [$AdminConfig list Server]
```

– Using Jython:

```
servers = AdminConfig.list('Server')
```

To look for all the server configuration IDs in the *mynode* node, use the code in the following example:

– Using Jacl:

```
set scopeid [$AdminConfig getid /Node:mynode/]
set nodeServers [$AdminConfig list Server $scopeid]
```

– Using Jython:

```
scopeid = AdminConfig.getid('/Node:mynode/')
nodeServers = AdminConfig.list('Server', scopeid)
```

2. If more than one configuration ID is returned from the **getid** or the **list** command, the IDs are returned in a list syntax. One way to retrieve a single element from the list is to use the **lindex** command. The following example retrieves the first configuration ID from the server object list:

   • Using Jacl:

```
set allServers [$AdminConfig getid /Server:/]
set aServer [lindex $allServers 0]
```

   • Using Jython:

```
allServers = AdminConfig.getid('/Server:/')

 # get line separator
 import  java
 lineSeparator = java.lang.System.getProperty('line.separator')

 arrayAllServers = allServers.split(lineSeparator)
 aServer = arrayAllServers[0]
```

   For other ways to manipulate the list and perform pattern matching to look for a specified configuration object, refer to the topic on Jacl syntax.

Results

You can now use the configuration ID in any subsequent AdminConfig commands that require a configuration ID as a parameter.

Listing attributes of configuration objects using the wsadmin scripting tool

You can use scripting to generate a list of attributes of configuration objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting wsadmin.

About this task

Run an AdminConfig command to create a list of attributes of configuration objects.

Procedure

1. List the attributes of a given configuration object type, using the **attributes** command, for example:
   - Using Jacl:

```
$AdminConfig attributes type
```

   - Using Jython:

```
AdminConfig.attributes('type')
```

Table 24. AdminConfig attributes command description. Run the attributes command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`attributes`	is an AdminConfig command
`type`	is an object type

   This command returns a list of attributes and its data type.

   To get a list of attributes for the JDBCProvider type, use the following example command:
   - Using Jacl:

```
$AdminConfig attributes JDBCProvider
```

   - Using Jython:

```
AdminConfig.attributes('JDBCProvider')
```

2. List the required attributes of a given configuration object type, using the **required** command, for example:
   - Using Jacl:

```
$AdminConfig required type
```

   - Using Jython:

```
AdminConfig.required('type')
```

Table 25. AdminConfig required command description. Run the command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`required`	is an AdminConfig command
`type`	is an object type

   This command returns a list of required attributes.

   To get a list of required attributes for the JDBCProvider type, use the following example command:
   - Using Jacl:

```
$AdminConfig required JDBCProvider
```

   - Using Jython:

```
AdminConfig.required('JDBCProvider')
```

3. List attributes with defaults of a given configuration object type, using the **defaults** command, for example:
   - Using Jacl:

```
$AdminConfig defaults type
```

   - Using Jython:

```
AdminConfig.defaults('type')
```

Table 26. AdminConfig defaults command description. Run the command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`defaults`	is an AdminConfig command
`type`	is an object type

This command returns a list of all the attributes, types, and defaults.

To get a list of attributes with the defaults displayed for the JDBCProvider type, use the following example command:

- Using Jacl:

```
$AdminConfig defaults JDBCProvider
```

- Using Jython:

```
AdminConfig.defaults('JDBCProvider')
```

Modifying configuration objects using the wsadmin scripting tool

Modifying configuration objects using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Starting the wsadmin scripting client topic for more information.

About this task

When using the **modify** command for the AdminConfig object, use the configuration object ID to modify the attribute you want to change. If you use the parent object ID to modify the attribute, the command resets all other attributes that are not specified to the default values. For example, you use the **modify** command to change the monitoring policy settings through its parent object, the process definition object. All attributes for the process definition object that were not modified with the command, such as the **pingInterval** and **pingTimeout** attributes, are reset to their default values.

Perform the following steps to modify a configuration object:

Procedure

1. Retrieve the configuration ID of the objects that you want to modify, for example:
   - Using Jacl:

```
set jdbcProvider1 [$AdminConfig getid /JDBCProvider:myJdbcProvider/]
```

   - Using Jython:

```
jdbcProvider1 = AdminConfig.getid('/JDBCProvider:myJdbcProvider/')
```

Table 27. AdminConfig getid command description. Invoke a command from a wsadmin command line.

`set`	is a Jacl command
`jdbcProvider1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`/JDBCProvider:myJdbcProvider/`	is the hierarchical containment path of the configuration object
`JDBCProvider`	is the object type
`myJdbcProvider`	is the optional name of the object

2. Show the current attribute values of the configuration object with the **show** command, for example:
   - Using Jacl:

```
$AdminConfig show $jdbcProvider1
```

   - Using Jython:

```
AdminConfig.show(jdbcProvider1)
```

Table 28. AdminConfig show command description. Invoke a command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
show	is an AdminConfig command
jdbcProvider1	evaluates to the ID of the host node that is specified in step number 1

3. Modify the attributes of the configuration object.

   Examples:

   - Using Jacl:

```
$AdminConfig modify $jdbcProvider1 {{description "This is my new description"}}
$AdminConfig modify $outPort {{retargettedURI "endpoint address"}}
```

   - Using Jython list:

```
AdminConfig.modify(jdbcProvider1, [['description', "This is my new description"]])
AdminConfig.modify(outPort, [['retargettedURI', "endpoint address"]])
```

   - Using Jython string:

```
AdminConfig.modify(jdbcProvider1, '[[description "This is my new description"]]')
AdminConfig.modify(outPort, '[[retargettedURI "endpoint address"]]')
```

   where:

Table 29. AdminConfig modify command description. Invoke a command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
modify	is an AdminConfig command
jdbcProvider1	evaluates to the ID of the host node that is specified in step number 1
description	is an attribute of server objects
This is my new description	is the value of the description attribute
outPort	is the name of the SIBWSOutboundPort created using the addSIBWSOutboundPort command. The AdminConfig command can also be used to modify the other SIBWSOutboundPort command attributes.
retargettedURI	is the attribute of outport objects. This particular attribute is equivalent to changing the value specified for the endpoint address property on the administrative console.
endpoint address	is the value of the retargettedURI attribute

   You can also modify several attributes at the same time. For example:

   - Using Jacl:

```
{{name1 val1} {name2 val2} {name3 val3}}
```

   - Using Jython list:

```
[['name1', 'val1'], ['name2', 'val2'], ['name3', 'val3']]
```

   - Using Jython string:

```
'[[name1 val1] [name2 val2] [name3 val3]]'
```

4. List all of the attributes that can be modified:

   - Using Jacl:

```
$AdminConfig attributes JDBCProvider
```

   - Using Jython:

```
print AdminConfig.attributes('JDBCProvider')
```

   Example output:

```
$AdminConfig attributes JDBCProvider
"classpath String*"
"description String"
"implementationClassName String"
"name String"
"nativepath String*"
"propertySet J2EEResourcePropertySet"
"providerType String"
"xa boolean"
```

5. Modify an attribute that has a type of list and collection. By default, if you try to modify an attribute that has a type of list and collection, and the attribute has an existing value in the list, it will append the new value to the existing values. An attribute that has a type of list and collection will have a star (*). In the following example, the attribute classpath has an type of list and collection and the value is String. If you want to replace the existing value, you must change the classpath to be an empty list before you modify the new value. For example:
   - Using Jacl:

```
$AdminConfig modify $jdbcProvider1 {{classpath {}}}

$AdminConfig modify $jdbcProvider1 [list [list classpath /temp/db2j.jar]]
```

   - Using Jython list:

```
AdminConfig.modify(jdbcProvider1, [['description', []]])

AdminConfig.modify(jdbcProvider1, [['description', '/temp/db2j.jar']]
```

   - Using Jython string:

```
AdminConfig.modify(jdbcProvider1, '[]')

AdminConfig.modify(jdbcProvider1, '[[description /temp/db2j.jar]]')
```

6. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Removing configuration objects with the wsadmin tool

Use this task to delete a configuration object from the configuration repository. This action only affects the configuration.

About this task

If a running instance of a configuration object exists when you remove the configuration, the change has no effect on the running instance.

Procedure

1. Start the wsadmin scripting tool.
2. Assign the ID string that identifies the server that you want to remove:

   Using Jacl:

   ```
   set s1 [$AdminConfig getid /Node:mynode/Server:myserver/]
   ```

   Using Jython:

   ```
   s1 = AdminConfig.getid('/Node:mynode/Server:myserver/')
   ```

Table 30. AdminConfig getid command description. The following table describes the AdminConfig getid command.

`set`	is a Jacl command
`s1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is an object type

Table 30. AdminConfig getid command description (continued). The following table describes the AdminConfig getid command.

mynode	is the host name of the node from which the server is removed
Server	is an object type
myserver	is the name of the server to remove

3. Remove the configuration object. For example:
   - Using Jacl:
     ```
     $AdminConfig remove $s1
     ```
   - Using Jython:
     ```
     AdminConfig.remove(s1)
     ```

Table 31. AdminConfig remove command description. The following table describes the AdminConfig remove command.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
remove	is an AdminConfig command
s1	evaluates the ID of the server that is specified in step number 2

4. Save the configuration changes.
   Use the following command example to save your configuration changes:
   ```
   AdminConfig.save()
   ```

Results

The application server configuration no longer contains a specific server object. Running servers are not affected.

Removing the trust association interceptor class using scripting

Use the wsadmin tool to remove the trust association interceptor class.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 article for more information.

About this task

Use the following example as a Jacl script file and run it with the "-f" option:

Procedure

Using Jacl:

```
set variableName "com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus"
set cellName $env(local.cell)

foreach taiEntry [$AdminConfig list TAInterceptor] {
   set interceptorClass [lindex [$AdminConfig showAttribute $taiEntry interceptorClassName] 0]
   if { [string compare $interceptorClass $variableName] == 0 } {
       puts "found $interceptorClass"
       puts "Removing the TAIntercepter class '$interceptorClass'"
       set tai taiEntry
       #set t [$AdminConfig getid /Cell:$cellName/TAInterceptor:/]
       #$AdminConfig remove $t
       $AdminConfig remove $taiEntry
       puts "'$interceptorClass' is removed."
```

```
      break
   }
}

if { ![info exists tai] } {
    puts "The class '$variableName' does not exist."
}

$AdminConfig save
```

Results

Example output:

```
[root@svtaix23] /tmp
==>/usr/6*/A*/profiles/D*/bin/wsadmin.sh -f tai.jacl

WASX7209I: Connected to process "dmgr" on node svtaix23CellManager01 using SOAP connector;
The type of process is: DeploymentManager
found com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus
Removing the TAIntercepter class 'com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus'
'com.ibm.ws.security.web.TAMTrustAssociationInterceptorPlus' is removed.
```

Changing the application server configuration using the wsadmin tool

You can use the wsadmin AdminConfig and AdminApp objects to make changes to the application server configuration.

About this task

The purpose of this article is to illustrate the relationship between the commands that are used to change the configuration and the files that are used to hold configuration data. This discussion assumes that you have a network deployment installation, but the concepts are very similar for a application server installation.

Procedure

1. Start the wsadmin scripting tool.
2. Set a variable for creating a server:
   - Using Jacl:

     `set n1 [$AdminConfig getid /Node:`*`mynode`*`/]`
   - Using Jython:

     `n1 = AdminConfig.getid('/Node:`*`mynode`*`/')`

Table 32. AdminConfig getid command description. The following table describes the AdminConfig getid command.

`set`	is a Jacl command
`n1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Node`	is the object type
`mynode`	is the name of the object to modify

3. Create a server with the following command:
   - Using Jacl:

     `set serv1 [$AdminConfig create Server $n1 {{name `*`myserv`*`}}]`
   - Using Jython list:

     `serv1 = AdminConfig.create('Server', n1, [['name', '`*`myserv`*`']])`
   - Using Jython string:

```
serv1 = AdminConfig.create('Server', n1, '[[name myserv]]')
```

Table 33. AdminConfig create command description. The following table describes the AdminConfig create command.

`set`	is a Jacl command
`serv1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Server`	is an AdminConfig object
`n1`	evaluates to the ID of the host node that is specified in step number 1
`name`	is an attribute
`myserv`	is the value of the name attribute

After this command completes, some new files can be seen in a workspace used by the deployment manager server on behalf of this scripting client. A workspace is a temporary repository of configuration information that administrative clients use. Any changes made to the configuration by an administrative client are first made to this temporary workspace. For scripting, when a **save** command is invoked on the AdminConfig object, these changes are transferred to the real configuration repository. Workspaces are kept in the `wstemp` subdirectory of a WebSphere Application Server installation.

4. Make a configuration change to the server with the following command:
   - Using Jacl:
     ```
     $AdminConfig modify $serv1 {{stateManagement {{initialState STOP}}}}
     ```
   - Using Jython list:
     ```
     AdminConfig.modify(serv1, [['stateManagement', [['initialState', 'STOP']]]])
     ```
   - Using Jython string:
     ```
     AdminConfig.modify(serv1, '[[stateManagement  [[initialState  STOP]]]]')
     ```

Table 34. AdminConfig modify command description. The following table describes the AdminConfig modify command.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`serv1`	evaluates to the ID of the host node that is specified in step number 2
`stateManagement`	is an attribute
`initialState`	is a nested attribute within the stateManagement attribute
`STOP`	is the value of the initialState attribute

This command changes the initial state of the new server. After this command completes, one of the files in the workspace is changed.

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Modifying nested attributes using the wsadmin scripting tool

You can modify nested attributes for a configuration object using scripting and the wsadmin tool.

About this task

The attributes for a WebSphere Application Server configuration object are often deeply nested. For example, a JDBCProvider object has an attribute factory, which is a list of the J2EEResourceFactory type objects. These objects can be DataSource objects that contain a connectionPool attribute with a ConnectionPool type that contains a variety of primitive attributes.

Procedure

1. Invoke the AdminConfig object commands interactively, or in a script, from an operating system command prompt.

   See the topic on starting the wsadmin scripting client.

2. Obtain the configuration ID of the object, for example:

   Using Jacl:

```
set t1 [$AdminConfig getid /DataSource:TechSamp/]
```

   Using Jython:

```
t1=AdminConfig.getid('/DataSource:TechSamp/')
```

Table 35. AdminConfig getid command description. Run a command from a wsadmin command line.

`set`	is a Jacl command
`t1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`DataSource`	is the object type
TechSamp	is the name of the object that will be modified

3. Modify one of the object parents and specify the location of the nested attribute within the parent, for example:

   Using Jacl:

```
$AdminConfig modify $t1 {{connectionPool {{reapTime 2003}}}}
```

   Using Jython list:

```
AdminConfig.modify(t1, [["connectionPool", [["reapTime", 2003]]]])
```

   Using Jython string:

```
AdminConfig.modify(t1, '[[connectionPool [[reapTime 2003]]]]')
```

Table 36. AdminConfig modify command description. Run a command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`t1`	evaluates to the configuration ID of the datasource in step number 2
`connectionPool`	is an attribute
`reapTime`	is a nested attribute within the connectionPool attribute
2003	is the value of the reapTime attribute

4. Save the configuration by issuing an AdminConfig **save** command. For example:

   Using Jacl:

```
$AdminConfig save
```

   Using Jython:

```
AdminConfig.save()
```

Use the **reset** command of the AdminConfig object to undo changes that you made to your workspace since your last save.

Example

An alternative way to modify nested attributes is to modify the nested attribute directly.

Using Jacl:

```
set techsamp [$AdminConfig getid /DataSource:TechSamp/]
set pool [$AdminConfig showAttribute $techsamp connectionPool]
$AdminConfig modify $pool {{reapTime 2003}}
```

Using Jython list:

```
techsamp=AdminConfig.getid('/DataSource:TechSamp/')
pool=AdminConfig.showAttribute(techsamp,'connectionPool')
AdminConfig.modify(pool,[['reapTime',2003]])
```

Using Jython string:

```
techsamp=AdminConfig.getid('/DataSource:TechSamp/')
pool=AdminConfig.showAttribute(techsamp,'connectionPool')
AdminConfig.modify(pool,'[[reapTime 2003]]')
```

In this example, the first command gets the configuration id of the DataSource, and the second command gets the connectionPool attribute. The third command sets the reapTime attribute on the ConnectionPool object directly.

Saving configuration changes with the wsadmin tool

Use the wsadmin tool and scripting to save configuration changes to the master configuration repository.

About this task

The wsadmin tool uses the workspace to hold configuration changes. You must save your changes to transfer the updates to the master configuration repository. If a scripting process ends and you have not saved your changes, the changes are discarded.

Procedure

Use the following commands to save the configuration changes:

1. Using Jacl:

```
$AdminConfig save
```

2. Using Jython:

```
AdminConfig.save()
```

Table 37. AdminConfig save command description. Run a command from a wsadmin command line.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`save`	is an AdminConfig command

If you are using interactive mode with the wsadmin tool, you will be prompted to save your changes before they are discarded.
If you are using the -c option with the wsadmin tool, changes are automatically saved. On a Unix operating system, if you invoke a command that includes a dollar sign character ($) using the wsadmin -c option, the command line attempts to substitute a variable. To avoid this problem, escape the dollar sign character with a backslash character (\). For example: `wsadmin -c "\$AdminConfig save"`.
If a scripting process ends and no save has been performed, any configuration changes made since the last save are discarded. If there are multiple clients (scripts or browser clients) updating the configuration at the same time, it is possible that the changes requested by a script may not be saved. If this happens,

you will receive an exception and you must make the updates again. If the save fails, the updates will not be saved to the configuration. If it succeeds, all updates are saved. To avoid save failures, you can invoke the **save** command after every configuration update.
You can use the **reset** command of the AdminConfig object to undo changes that you made to your configuration since your last save.

Chapter 9. Using the wsadmin scripting AdminTask object for scripted administration

Use the AdminTask object to access a set of administrative commands that provide an alternative way to access the configuration commands and the running object management commands.

Before you begin

The administrative commands run simple and complex commands. They provide more user friendly and task-oriented commands. The administrative commands are discovered dynamically when you start a scripting client. The set of available administrative commands depends on the edition of WebSphere Application Server that you installed. You can use the AdminTask object commands to access these commands.

About this task

Administrative commands are grouped based on their function. You can use administrative command groups to find related commands. For example, the administrative commands that are related to server management are grouped into a server management command group. The administrative commands that are related to the security management are grouped into a security management command group. An administrative command can be associated with multiple command groups because it can be useful for multiple areas of system management. Both administrative commands and administrative command groups are uniquely identified by their name.

Two run modes are always available for each administrative command, namely the *batch* and *interactive mode*. When you use an administrative command in interactive mode, you go through a series of steps to collect your input interactively. This process provides users a text-based wizard and a similar user experience to the wizard in the administrative console. You can also use the **help** command to obtain help for any administrative command and the AdminTask object.

The administrative commands do not replace any existing configuration commands or running object management commands but provide a way to access these commands and organize the inputs. The administrative commands can be available in connected or local mode. The set of available administrative commands is determined when you start a scripting client in connected or local mode. If a server is running, it is not recommended that you run the scripting client in local mode because any configuration changes made in local mode are not reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

Use parameter name and parameter value pairs to specify the parameters of a step in any order. You do not have to specify option parameters. This applies to all commands for the AdminTask object. For example:

```
AdminTask.createCluster('[-clusterConfig [-clusterName cluster1 -preferLocal true]]')
```

To determinethe names of the step parameters, use the following command: `AdminTask.help('command_name', 'step_name')`, as the following example demonstrates::

```
AdminTask.help('createCluster', 'clusterConfig')
```

Procedure

- Read “Invoking an administrative command in batch mode using wsadmin scripting” on page 70 to use administrative commands in batch mode.
- Read “Invoking an administrative command in interactive mode using wsadmin scripting” on page 75 to use administrative commands in interactive mode.

- Read “Obtaining online help using wsadmin scripting” to learn how to use scripting for online help.

Obtaining online help using wsadmin scripting

You can select from three levels of online help for administrative commands.

About this task

The top-level help provides general information for the AdminTask object and associated commands. The second-level help provides information about all of the available administrative commands and command groups. The third-level help provides specific help on a command group, a command, or a step. Command group-specific help provides descriptions for the command group that you specify and the commands that belong to the associated group. Command-specific help provides description for the specified command, and associated parameters and steps. Step-specific help provides a description for the specified step and the associated parameters. For command and step-specific help, required parameters are marked with an asterisk (*) in the help output.

Procedure

- To obtain general help, use the following examples:

  Using Jacl:

  ```
  $AdminTask help
  ```

  Using Jython:

  ```
  print AdminTask.help()
  ```

  Example output:

  ```
  WASX8001I: The AdminTask object enables the execution of available
             admin commands. AdminTask commands operate in two modes:
             the default mode is one which AdminTask communicates with the
             WebSphere server to accomplish its task. A local mode is also
             available in which no server communication takes place. The local
             mode of operation is invoked by bringing up the scripting client
             using the command line "-conntype NONE" option or setting the
             "com.ibm.ws.scripting.connectiontype=NONE" property in
             wsadmin.properties file.

  The number of admin commands varies and depends on your WebSphere install.
  Use the following help commands to obtain a list of supported commands
  and their parameters:

  help -commands
               list all the admin commands
  help -commandGroups
               list all the admin command groups
  help commandName
               display detailed information for
               the specified command
  help commandName stepName
               display detailed information for
               the specified step belonging to
               the specified command
  help commandGroupName
               display detailed information for
               the specified command group

  There are various flavors to invoke an admin command:

  commandName
               invokes an admin command that does
               not require any argument.
  ```

```
commandName targetObject
             invokes an admin command with the specified
             target object string, for example, the
             configuration object name of a resource
             adapter. The expected target object varies
             with the admin command invoked. Use help
             command to get information on the target
             object of an admin command.

commandName options
             invokes an admin command with the specified
             option strings. This invocation syntax is
             used to invoke an admin command that does
             not require a target object. It is also used
             to enter interactive mode if "-interactive"
             mode is included in the options string.

commandName targetObject options
             invokes an admin command with the specified
             target object and options strings. If
             "-interactive" is included in the options
             string, then interactive mode is entered.
             The target object and options strings vary
             depending on the admin command invoked.
             Use help command to get information on the
             target object and options.
```

- To list the available command groups, use the following examples:

  Using Jacl:

  ```
  $AdminTask help -commandGroups
  ```

  Using Jython:

  ```
  print AdminTask.help('-commandGroups')
  ```

  Example output:

  ```
  WASX8005I: Available admin command groups:

  ClusterConfigCommands - Commands for configuring application
  server clusters and cluster members.
  JCAManagement - A group of admin commands that helps to configure
  Java2 Connector Architecture(J2C) related resources.
  ```

- To list the available commands, use the code in the following examples:

  Using Jacl:

  ```
  $AdminTask help -commands
  ```

  Using Jython:

  ```
  print AdminTask.help('-commands')
  ```

  Example output:

  ```
  WASX8004I: Available administrative commands:

  copyResourceAdapter - copy the specified J2C resource adapter to the specified scope
  createCluster - Creates a new application server cluster.
  createClusterMember - Creates a new member of an application server cluster.
  createJ2CConnectionFactory - Create a J2C connection factory
  deleteCluster - Delete the configuration of an application server cluster.
  deleteClusterMember - Deletes a member from an application server cluster.
  listConnectionFactoryInterfaces - list all of the
  defined connection factory interfaces on the
  specified J2C resource adapter.
  listJ2CConnectionFactories - List J2C connection factories that have a specified
  connection factory interface defined in the specified J2C resouce adapter
  createJ2CAdminObject - Create a J2C administrative object.
  listAdminObjectInterfaces - List all the defined administrative object interfaces
  on the specified J2C resource adapter.
  interface on the specified J2C resource adapter.
  ```

```
listJ2CAdminObjects - List the J2C administrative objects that have a specified
administrative object interface defined in the specified J2C resource adapter.
createJ2CActivationSpec - Create a J2C activation specification.
listMessageListenerTypes - list all of the defined messageListener
type on the specified J2C resource adapter.
listJ2CActivationSpecs - List the J2C activation specifications that have a
specified message listener type defined in the specified J2C resource adapter.
```

- To obtain help about a command group, use the following examples:

  Using Jacl:

  ```
  $AdminTask help JCAManagement
  ```

  Using Jython:

  ```
  print AdminTask.help('JCAManagement')
  ```

  Example output:

  ```
  WASX8007I: Detailed help for command group: JCAManagement

  Description: A group of administrative commands that help to
  configure Java 2 Connector Architecture (J2C)-related resources.

  Commands:
  createJ2CConnectionFactory - Create a J2C connection factory
  listConnectionFactoryInterfaces - list all of the defined connection
  factory interfaces on the specified J2C resource adapter.
  listJ2CConnectionFactories - List J2C connection factories that have
  a specified connection factory interface defined in the
  specified J2C resouce adapter.
  createJ2CAdminObject - Create a J2C administrative object.
  listAdminObjectInterfaces - List all the defined administrative
  object interfaces on the specified J2C resource adapter.
  listJ2CAdminObjects - List the J2C administrative objects that have a
  specified administrative object interface defined in the
  specified J2C resource adapter.
  createJ2CActivationSpec - Create a J2C activation specification.
  listMessageListenerTypes - list all of the defined
  message listener types on the specified J2C resource adapter.
  listJ2CActivationSpecs - List the J2C activation specifications that
  have a specified message listener type defined in the
  specified J2C resource adapter.
  copyResourceAdapter - copy the specified J2C resource
  adapter to the specified scope.
  ```

- To obtain help about an administrative command, use the following examples:

  Using Jacl:

  ```
  $AdminTask help createJ2CConnectionFactory
  ```

  Using Jython:

  ```
  print AdminTask.help('createJ2CConnectionFactory')
  ```

  Example output:

  ```
  WASX8006I: Detailed help for command: createJ2CConnectionFactory

  Description: Create a J2C connection factory

  *Target object: The parent J2C resource adapter of the created J2C connection factory.

  Arguments:
  *connectionFactoryInterface - A connection factory interface that is defined in the deployment
    description of the parent J2C resource adapter.
  *name - The name of the J2C connection factory.
  *jndiName - The JNDI name of the created J2C connection factory.
  description - The description for the created J2C connection factory.
  authDataAlias - the authentication data alias of the created J2C connection factory.

  Steps:
  None
  ```

In the command-specific help output that is previously listed, an administrative command is divided into three input areas: target object, arguments, and steps. Each area can require input depending on the administrative command. If an area requires input, each input is described by its name and a description; except for the target object area, which contains the description of the target object only. When you use an administrative command in batch mode, you can use any input name that resides in the argument area as the argument name.

If an input is required, an asterisk (*) is located before the name. If an area does not require an input, it is marked `None`. The following example uses the help output for the **createJ2CConnectionFactory** command:

- The target object area requires the configuration object name of a J2CResourceAdapter.
- In the arguments area, there are five inputs with three being required inputs. The argument names are connectionFactoryInterface, name, jndiName, description, and authDataAlias. These names are used as the parameter names in the option string to run an administrative command in batch mode, for example:

  ```
  -connectionFactoryInterface javax.resource.cci.ConnectionFactory -name newConnectionFactory
  -jndiName CF/newConnectionFactory
  ```

  See Administrative command invocation syntax using wsadmin scripting for more information about specifying argument options.
- No step is associated with this administrative command.

- To obtain help on a command step, use the step-specific help.

  Step-specific help provides the following data:
  - A description for the command step.
  - Information indicating if this step supports collection. A collection includes objects of the same type. In a command step, a collection contains objects that have the same set of parameters.
  - Information regarding each step parameter with its name and description. If a step parameter is required, an asterisk (*) is located in front of the name.

  The following example obtains help on a command step:

  Using Jacl:

  ```
  $AdminTask help createCluster clusterConfig
  ```

  Using Jython:

  ```
  print AdminTask.help('createCluster', 'clusterConfig')
  ```

  Example output:

  ```
  WASX8013I: Detailed help for step: clusterConfig

  Description: Specifies the configuration of the new server cluster.

  Collection: No

  Arguments:
    *clusterName - Name of server cluster.
    preferLocal - Enables node-scoped routing optimization for the cluster.
  ```

  This example indicates the following information about the clusterConfig step:
  - This step does not support collection. Only one set of parameter values for the clusterName and perferLocal parameters is supported.
  - This step contains two input arguments with one argument that is indicated as required. The required arguments is clusterName and the non-required parameter is preferLocal. The syntax to provide step parameter values is different from the command argument values. You have to provide all argument values of a step and provide them in the exact order as displayed in the step specific help. For any optional argument that you do not want to specify a value, put double quotes ("") in place of a value. If a command step is a collection type, for example, it can contain multiple objects where each object

has the same set of arguments, you can specify multiple objects with each object enclosed by its own pair of braces. To run an administrative command in batch mode and to include this step in the option string, use the following syntax:

Using Jacl:

```
-clusterConfig {{newCluster false}}
```

Using Jython:

```
-clusterConfig [[newCluster false]]
```

See Administrative command invocation syntax using wsadmin scripting for more information about specifying parameter options.

- Use a wildcard character to search for help for a specific command. You can use a regular Java expression pattern or a wildcard pattern to specify command name for AdminTask.help('–commands') and AdminConfig list, types and listTemplates functions.
  - To use a regular Java expression pattern to search for the administrative command names that start with `create`, specify:

    ```
    print AdminTask.help("-commands", "create.*")
    ```
  - To use a wildcard search pattern to search for the administrative command names that start with `create`, specify:

    ```
    print AdminTask.help("-commands", "create*")
    ```
  - To use a Java expression pattern to search for the administrative command names that contain `SSLConfig`, specify:

    ```
    print AdminTask.help("-commands", ".*SSLConfig.*")
    ```
  - To use a wildcard search pattern to search for the administrative command names that contain `SSLConfig`, specify:

    ```
    print AdminTask.help("-commands", "*SSLConfig*")
    ```

Invoking an administrative command in batch mode using wsadmin scripting

Use AdminTask commands to invoke an administrative command in batch mode.

About this task

This topic describes how to invoke an administrative command in batch mode using wsadmin scripting.

To invoke an administrative command in interactive mode, see the topic on invoking a command in interactive mode.

Procedure

1. Invoke the AdminTask object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.

   See the topic on starting the wsadmin scripting client.
2. Issue one of the following commands:
   - If an administrative command does not have a target object and an argument, use the following command:

     Using Jacl:

```
$AdminTask commandName
```

     Using Jython:

```
AdminTask.commandName()
```

Table 38. AdminTask description. Invoke an AdminTask command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object allowing administrative command management
commandName	is the name of the administrative command to invoke

- If an administrative command includes a target object but does not include any arguments or steps, use the following command:

  Using Jacl:

```
$AdminTask commandName targetObject
```

  Using Jython:

```
AdminTask.commandName(targetObject)
```

Table 39. AdminTask targetObject description. Invoke an AdminTask command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
targetObject	is the target object string for the invoked administrative command. The expect target object varies with each administrative command. View the online help for the invoked administrative command to learn more about what you should specify as the target object.

- If an administrative command includes an argument or a step but does not include a target object, use the following command:

  Using Jacl:

```
$AdminTask commandName options
```

  Using Jython:

```
AdminTask.commandName(options)
```

Table 40. AdminTask options description. Invoke an AdminTask command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke

Table 40. AdminTask options description (continued). Invoke an AdminTask command from a wsadmin command line.

options	is the option string for the invoked administrative command. Depending on which administrative command you are invoking, the administrative command can have required or optional option values. The options string is different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps listed on the online administrative command help are specified as options in the option string. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. If the invoked administrative command includes target objects, arguments, or steps, then the –interactive option is available to enter interactive mode. For example, using the output of the following online help for the listDataSource command: `WASX8006I: Detailed help for command: exportServer` `Description: export the configuration of a server to a config archive.` `Target object: None` `Arguments: *serverName - the name of a server *nodeName - the name of a node. This parameter becomes optional if the specified server name is unique across the cell. *archive - the fully qualified file path of a config archive.` `Steps: None` Option names are specified with a dash before the names. Three options are required for this administrative command. The required options are -serverName, -nodename, and -archive. In addition, the -interactive option is available. Options are specified in the option string, which is enclosed by a pair of braces ({}) in Jacl and a pair of brackets ([]) in Jython.

- If an administrative command includes a target object, and arguments or steps:

  Using Jacl:

```
$AdminTask commandName targetObject options
```

  Using Jython:

```
AdminTask.commandName(targetObject, options)
```

Table 41. AdminTask targetObject with options description. Invoke an AdminTask command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke

Table 41. AdminTask targetObject with options description (continued). Invoke an AdminTask command from a wsadmin command line.

targetObject	is the target object string for the invoked administrative command. The expected target object varies with each administrative command. View the online help for the invoked administrative command to obtain information about what to specify as a target object. For example, using the output of the following online help for createJ2CConnectionFactory: `WASX8006I: Detailed help for command:` `createJ2CConnectionFactory` `Description: Create a J2C connection factory` `*Target object: The parent J2C resource adapter of` `the created J2C connection factory.` `Arguments:` `*connectionFactoryInterface - A connection factory` `interface that is defined in the deployment` `description of the parent J2C resource adapter.` `*name - The name of the J2C connection factory.` `*jndiName - The JNDI name of the created J2C` `connection factory.` `description - The description for the created` `J2C connection factory.` `authDataAlias - the authentication data alias of` `the created J2C connection factory.` `Steps:` `None` The target object is a configuration object name of a J2C resource adapter.

Table 41. AdminTask targetObject with options description (continued). Invoke an AdminTask command from a wsadmin command line.

options	is the option string for the invoked administrative command. Depending on which administrative command you are invoking, the administrative command can have required or optional option values. The options string is different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps that are listed on the online administrative command help are specified as options in the option string. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. If the invoked administrative command includes target objects, arguments, or steps, then the –interactive option is available to enter interactive mode. For example, using the output of the following online help for listDataSource: `WASX8006I: Detailed help for command:` `createJ2CConnectionFactory` `Description: Create a J2C connection factory` `*Target object: The parent J2C resource adapter of` `the created J2C connection factory.` `Arguments:` `*connectionFactoryInterface - A connection factory` `interface that is defined in the deployment` `description of the parent J2C resource adapter.` `*name - The name of the J2C connection factory.` `*jndiName - The JNDI name of the created J2C` `connection factory.` `description - The description for the created J2C` `connection factory.` `authDataAlias - the authentication data alias of` `the created J2C connection factory.` `Steps:` `None` Option names are specified with a dash before the names. The required options for this administrative command include: `-connectionFactoryInterface`, `-name`, and `-jndiName`. The optional options include: `-description` and `-authDataAlias`. In addition, you can also use the `-interactive` option. Options are specified in the option string, which is enclosed by a pair of braces ({}) in Jacl and a pair of brackets ([]) in Jython.

Example

- The following example invokes an administrative command with no target object, argument, or step:

  Using Jacl:

```
$AdminTask listNodes
```

  Using Jython:

```
print AdminTask.listNodes()
```

  Example output:

```
myNode
```

- The following example invokes an administrative command with a target object string:

  Using Jacl:

```
set s1 [$AdminConfig getid /Server:server1/]
$AdminTask showServerInfo $s1
```

  Using Jython:

```
s1 = AdminConfig.getid('/Server:server1/')
print AdminTask.showServerInfo(s1)
```

Example output:

```
{cell myCell}
{serverType APPLICATION_SERVER}
{com.ibm.websphere.baseProductVersion 6.0.0.0}
{node myNode}
{server server1}
```

- The following example invokes an administrative command with an option string:

  Using Jacl:

```
$AdminTask getNodeMajorVersion {-nodeName myNode}
```

  Using Jython:

```
print AdminTask.getNodeMajorVersion('[-nodeName myNode]')
```

  Example output:

```
6
```

- The following example invokes an administrative command with a target object and non-step option strings:

  Using Jacl:

```
set ra [$AdminConfig getid /J2CResourceAdapter:myResourceAdapter/]
$AdminTask createJ2CConnectionFactory $ra {-name myJ2CCF -jndiName j2c/cf -connectionFactoryInterface
javax.resource.cci.ConnectionFactory}
```

  Using Jython:

```
ra = AdminConfig.getid('/J2CResourceAdapter:myResourceAdapter/')
AdminTask.createJ2CConnectionFactory(ra, '[-name myJ2CCF -jndiName j2c/cf -connectionFactoryInterface
javax.resource.cci.ConnectionFactory]')
```

  Example output:

```
myJ2CCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_1069690568269)
```

- The following example invokes an administrative command with a target object and a step option:

  Using Jacl:

```
set serverCluster [$AdminConfig getid /ServerCluster:myCluster/]
$AdminTask createClusterMember $serverCluster {-memberConfig {{myNode myClusterMember "" "" false false}}}
```

  Using Jython:

```
serverCluster = AdminConfig.getid('/ServerCluster:myCluster/')
AdminTask.createClusterMember(serverCluster, '[-memberConfig [[myNode myClusterMember "" "" false false]]]')
```

  Example output:

```
myClusterMember(cells/myCell/nodes/myNode|cluster.xml#ClusterMember_3673839301876)
```

Invoking an administrative command in interactive mode using wsadmin scripting

These steps demonstrate how to invoke an administrative command in interactive mode.

About this task

This topic describes how to invoking an administrative command in interactive mode.

To invoke an administrative command in batch mode, see the topic on invoking a command batch mode.

Procedure

1. Invoke the AdminTask object commands interactively, in a script, or use the **wsadmin -c** command from an operating system command prompt.

   See the topic on starting the wsadmin scripting client.
2. Invoke an administrative command in interactive mode by issuing one of the following commands:
   - Use the following command invocation to enter interactive mode without providing another input in the command invocation:

     Using Jacl:

```
$AdminTask commandName {-interactive}
```

Using Jython:

```
AdminTask.commandName('[-interactive]')
```

Table 42. AdminTask command syntax. Invoke an AdminTask command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
-interactive	is the interactive option

- Use the following command invocation to enter interactive mode using an administrative command that takes a target object. You do not have to provide a target object to enter interactive mode. Target objects provided in the command invocation will be applied to the command and displayed as the current target object value during interactive prompting.

  Using Jacl:

```
$AdminTask commandName targetObject {-interactive}
```

  Using Jython:

```
AdminTask.commandName(targetObject, '[-interactive]')
```

Table 43. AdminTask commandName command description. Invoke a command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
targetObject	is the target object string for the invoked administrative command. The target object is different for each administrative command. View the online help for the invoked administrative command to learn more about what to specify as a target object.
-interactive	is the interactive option

- Use the following command invocation to enter interactive mode for an administrative command that takes options. You do not have to provide other options to enter interactive mode. Options provided in the command invocation are applied to the command and the option values will be displayed as the current values during interactive prompting.

  Using Jacl:

```
$AdminTask commandName {-interactive commandOptions}
```

  Using Jython:

```
AdminTask.commandName('[-interactive commandOptions]')
```

Table 44. AdminTask -interactive option description. Invoke a command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke
-interactive	is the interactive option

Table 44. AdminTask -interactive option description (continued). Invoke a command from a wsadmin command line.

commandOptions	is the command option that is available for the associated administrative command. Available command options are different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps that are listed on the online administrative command help are specified as command options. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. For example, using the output of the following online help for the createJ2CConnectionFactory command: `WASX8006I: Detailed help for command:` `createJ2CConnectionFactory` `Description: Create a J2C connection` `factory` `*Target object: The parent J2C resource` `adapter of the created J2C connection` `factory.` `Arguments:` `*connectionFactoryInterface - A connection` `factory interface that is` `defined in the deployment description of` `the parent J2C resource adapter.` `*name - The name of the J2C connection factory.` `*jndiName - The JNDI name of the created` `J2C connection factory.` `description - The description for the created` `J2C connection factory.` `authDataAlias - the authentication data alias` `of the created J2C connection factory.` `Steps:` `None` In this example, five options are available: • -connectionFactoryInterface • -name • -jndiName • -description • -authDataAlias Each option requires a value. Three of the options are required and are denoted with a star (*).

- Use the following command invocation to enter interactive mode for an administrative command that has a target object and options. You do not have to specify a target object to enter interactive mode. The values specified are applied to the command before the command data is displayed. As a result, the values specified will be displayed as the current values during interactive prompting.

  Using Jacl:

```
$AdminTask commandName targetObject {-interactive commandOptions}
```

  Using Jython:

```
AdminTask.commandName(targetObject, '[-interactive commandOptions]')
```

Table 45. AdminTask -interactive targetObject option description. Invoke a command from a wsadmin command line.

$	is a Jacl operator for substituting a variable name with its value
AdminTask	is an object that supports administrative command management
commandName	is the name of the administrative command to invoke

Table 45. AdminTask -interactive targetObject option description (continued). Invoke a command from a wsadmin command line.

targetObject	is the target object string for the invoked administrative command. The expect target object varies with each admin command. Consult the online help on the invoked administrative command to learn more about what to specify as target object.
-interactive	is the interactive option
commandOptions	is the command option that is available for the associated administrative command. Available command options are different for each administrative command. View the online help for the invoked administrative command to obtain more information about which options are available. Arguments and steps that are listed on the online administrative command help are specified as command options. Each option consists of a dash followed immediately by an option name, and then followed by an option value if the option requires a value. For example, using the output of the following online help for the createJ2CConnectionFactory command: `WASX8006I: Detailed help for command:` `createJ2CConnectionFactory` `Description: Create a J2C connection factory` `*Target object: The parent J2C resource adapter of` `the created J2C connection factory.` `Arguments:` `*connectionFactoryInterface - A connection factory` `interface that is` `defined in the deployment description of the` `parent J2C resource adapter.` `*name - The name of the J2C connection factory.` `*jndiName - The JNDI name of the created J2C` `connection factory.` `description - The description for the created` `J2C connection factory.` `authDataAlias - the authentication data alias` `of the created J2C connection factory.` `Steps:` `None` In this example, five options are available: • -connectionFactoryInterface • -name • -jndiName • -description • -authDataAlias Each option requires a value. Three of the options are required and are denoted with a star (*).

Example

- The following example invokes an administrative command in interactive mode by specifying the -interactive option:

  Using Jacl:

```
$AdminTask createJ2CConnectionFactory {-interactive}
```

  Using Jython:

```
AdminTask.createJ2CConnectionFactory('[-interactive]')
```

Example output:

```
Create a J2C connection factory

*The J2C resource adapter: "WebSphere Relational ResourceAdapter
(cells/myCell/nodes/myNode|resources.xml#builtin_rra)"

A connection factory
interface (connectionFactoryInterface):javax.resource.cci.ConnectionFactory
*Name (name): myJ2CCF
*The JNDI name (jndiName): j2c/cf
Description (description):
authentication data alias (authDataAlias):

create J2C connection factory

F (Finish)
C (Cancel)

Select [F, C]: [F]

myJ2CCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_1069690568269)
```

- The following example invokes an administrative command using the –interactive option with a target object that is specified in the command invocation:

  Using Jacl:

```
set ra [$AdminConfig getid /J2CResourceAdapter:myResourceAdapter/]
$AdminTask createJ2CConnectionFactory $ra {-interactive}
```

  Using Jython:

```
ra = AdminConfig.getid('/J2CResourceAdapter:myResourceAdapter/')
AdminTask.createJ2CConnectionFactory(ra, '[-interactive]')
```

  Example output:

```
Create a J2C connection factory

*The J2C resource adapter: ["WebSphere Relational ResourceAdapter
(cells/myCell/nodes/myNode|resources.xml#builtin_rra)"]

A connection factory interface (connectionFactoryInterface):
javax.resource.cci.ConnectionFactory
*Name (name): myJ2CCF
*The JNDI name (jndiName): j2c/cf
Description (description):
authentication data alias (authDataAlias):

create J2C Connection Factory

F (Finish)
C (Cancel)

Select [F, C]: [F]

myJ2CCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_1069690568269)
```

- The following example invokes an administrative command using the –interactive option where both the target object and the additional command options are specified in the command invocation:

  Using Jacl:

```
set ra [$AdminConfig getid /J2CResourceAdapter:myResourceAdapter/]
$AdminTask createJ2CConnectionFactory $ra {-name myNewCF -interactive}
```

  Using Jython:

```
ra = AdminConfig.getid('/J2CResourceAdapter:myResourceAdapter/')
AdminTask.createJ2CConnectionFactory(ra, '[-name myNewCF -interactive]')
```

  Example output:

```
Create a J2C connection factory

*The J2C resource adapter: ["WebSphere Relational ResourceAdapter
(cells/myCell/nodes/myNode|resources.xml#builtin_rra)"]

A connection factory interface (connectionFactoryInterface):javax.resource.cci.ConnectionFactory
*Name (name): [myNewCF]
*The JNDI name (jndiName): j2c/cf
Description (description):
authentication data alias (authDataAlias):

create J2C Connection Factory

F (Finish)
C (Cancel)
```

```
Select [F, C]: [F]

myNewCF(cells/myCell/nodes/myNode|resources.xml#J2CConnectionFactory_3839439380269)
```

Administrative command interactive mode environment using wsadmin scripting

An administrative command can be run in interactive mode by providing the -interactive option in the options string when invoking the command.

You can still provide other options, even when using the interactive option. The options values that are specified are applied to the command before the command data is displayed. Whether or not other options are specified, the wsadmin tool steps the user through the command to collect command information.

The general interactive flow sequence is:

1. Collect user inputs for target object and parameters
2. If the command does not include a step, the command execution menu displays to run or cancel the command.
3. If the command includes a step, the menu to select the step displays. When all the required inputs are entered, the menu includes command execution.
4. When a step is selected, if the step supports collection, then the menu to select an object in the collection displays and you can exit the step. If you exit the step, repeat steps 1-3.
5. Collect user inputs for the selected step or for an object in the collection
6. Repeat steps 4 and 5 if from the collection step menu
7. Repeat steps 3-5 if from step selection menu

Depending on what input area is enabled by an administrative command, you can go through part or all of the interactive flow sequence. If an administrative command is run in interactive mode, the syntax to run the command except for the deletion of collection object in batch mode is generated and logged as a WASX7278I message in both the interactive session and in the wsadmin trace file.

Collect user inputs for target object and parameters

The following interactive prompt is used to collect inputs for the Target object and Arguments input areas that are displayed in the command-specific help:

```
Command title

Command Description

*target object title [current or default value]:
*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...
```

Display command execution menu

If an administrative command does not contain a step, you are presented with the following menu after collecting values for target object and parameters:

```
Command title

F (Finish)
C (Cancel)

Select [F, C]: F
```

The Finish option runs the command and the Cancel option cancels the command. The default selection is F (Finish). This menu is the last menu that is displayed for a non-step command to exit interactive mode by either canceling or running the command.

Display command step selection and execution menu

If an administrative command contains a step, the following menu is displayed after collecting values for target object and parameters:

```
Command title
Command description
 -> *1. step1 title (step1 name)
     2. step2 title (step2 name)
    *3. step3 title (step3 name)
    (4. step4 title (step4 name))
     ...
     n. stepn title (stepn name)

S (Select)
N (Next)
P (Previous)
F (Finish)
C (Cancel)
H (Help)

Select [S, N, P, F, C, H]: S
```

The number of steps that is displayed in the menu depends on the administrative command. The step name is displayed for information and is the name that is used to set data in this step in batch mode. The following notations are used to describe a step:

- A "->" before the step indicates the current step position.
- A "*" before the step indicates a required step.
- A ( ) enclosing the entire step indicates a disabled step. You cannot navigate to this step by using the Next or Previous options.

Using the menu, you can navigate through steps sequentially by selecting Previous or Next. Select selects the current step, Finish runs the command, Cancel cancels the command, and Help provides online help for the command. Not all menu choices are available. Previous is not available if the current step is the first step. Next is not available if the current step is the last step. Finish is not available if still steps are still missing required inputs. The default selection is `S` (Select) if the current step is a valid step and steps are missing required inputs. Default selection is `F` (Finish) if all the required input is provided for the steps.

For commands with steps, you can exit interactive mode on this menu by either canceling or running the command.

Display collection step menu

A step might or might not support collection. A collection refers to objects of the same type. In an administrative command, a collection contains objects that have the same set of parameters. If a step that supports collection is selected, the wsadmin tool displays the following menu to add and select an object in the collection:

```
Step title (step name)
    | key param1 title (key param1 name), key param2 title (key param2 name), ...
    ------------------------------------------------------------------------
->  | object1 key param1 value, key param2 value, ...
   *| object2 key param1 value, key param2 value, ...
    ...
key param1 title, key param2 title, ... must be provided to specify a row in batch row.

S (Select Row)
N (Next)
P (Previous)
A (Add Row or Add Row Before)
D (Delete Row)
```

```
F (Finish)
H (Help)

Select [S, N, P, A, D, F, H]: F
```

The number of objects that display in the menu depends on the command step. Key parameters are identified by the step to use to uniquely identify an object in a collection. Key parameter values are displayed to identify an object to select. As with the command step selection menu, an arrow (->) is used to indicate the current object position, and a asterisk (*) is used to indicate that required input is missing in the object.

Use the menu to navigate through objects sequentially by selecting Previous or Next. Select Row selects the current object, Add Row adds a new object, Add Row Before adds a new object before the current object, Delete Row deletes the current object, Finish returns control back to the step selection and execution menu, and Help provides on-line help for the step. Not all menu choices are available. Previous is not available if there is no object in the collection or the first object is the current object. Next is not available if there is no object in the collection or the last object is the current object. Select Row is available only if there is a current object. Add Row is provided only if there is no object in the collection and the step supports new object to be added. Add Row Before is provided if the step supports new object to be added and there are existing objects in the collection. Delete Row is provided only if there is a current object and the step supports an object to be deleted. Finish is not available if there are still objects missing required inputs. Default selection is A (Add Row) when there is no object in the collection and the step supports objects to be added. Default selection is S (Select Row) if there is a current object and there are still objects missing required inputs. Default selection is F (Finish) if there is no required input missing in any object.

Collect user inputs for parameters of a collection object

After a collection object is selected, the parameter value for each parameter is prompted sequentially as shown in the following example:

```
*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...
```

The number of parameters depends on the number of arguments in the Argument area of the command step-specific help. The same asterisk (*) notation is used to denote a required parameter. If a parameter value is restricted to a set of values, then the valid choices are displayed. If the current or default value is available, it is displayed. For each writable parameter, you can accept the existing value by pressing Enter. To add or change an existing value, enter a new value and press Enter. For a read-only parameter, the parameter and its value are displayed. You will not be given the prompt to modify its value. After you go through all of the parameters, the wsadmin tool returns to the collection step menu.

Collect user inputs for non-collection step

This step has two parts. The first part displays the current or default parameter values for the selected step, as shown in the following example:

```
Step title (step name)

*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...

Select [C (Cancel), E (Edit)]: [E]
```

No prompting is included in this part. Instead, this part is more like a help function providing parameter information on the selected step. The number of parameters depends on the number of arguments in the `argument` area of the command step specific help. The asterisk (*) notation denotes a required parameter. If a parameter value is restricted to a set of values, then the valid choices will be displayed. If the current

or default value is available, it is displayed. You can choose to cancel the step or continue to the next part to provide parameter inputs. The default selection is `Edit`. Because it is possible that you are seeing default values assigned to a new piece of data that is not yet set in the step, you can accept the default selection to continue to the next part. Otherwise, if no data exists in the selected step, selecting Cancel does not result in creating the data.

If you accept the default `Edit` selection, collect user inputs for parameters sequentially just like Collect user inputs for parameters of a collection object.

```
*param1 title (param1 name) [choice1, choice2, ...]: [current/default value]
param2 title (param2 name) [choice1, choice2, ...]: [current/default value]
...
```

For each writable parameter, you can accept the existing value by pressing Enter. To add or change an existing value, enter a new value and then press Enter. For a read-only parameter, the parameter and its value are displayed. You will not be given the prompt to modify the value of the parameter. As soon as you step through all the parameters, the wsadmin tool will lead you back to the command step selection and execution menu.

Data types for the AdminTask object using wsadmin scripting

The parameters for the AdminTask object accept various data types for different commands. This topic provides examples of valid data type syntax.

The following table lists the primitive and Java data types that the AdminTask object accepts for different commands, including the following data types:

- String
- Boolean
- Character
- Integer
- Long
- Byte
- Short
- Float
- Double
- Javax.management.ObjectName
- Java.util.Properties
- String[]
- Integer[]
- Jaca.net.URL
- Javax.management.Attribute
- Javax.management.AttributeList
- Java.util.ArrayList
- Java.util.List
- Java.util.Hashtable

The following example command specifies various data types for parameter values that are commonly used with the AdminTask object:

```
wsadmin>AdminTask.helloWorld('[-personName John -personalInfo [ [cellPhone 123-456-7890] [workPhone 123-456-7892]
 [homePhone 123-456-7891] ] -pets [dog cat] -personID WebSphere:John(organization=ibm,country=usa,state=texas,city=austin)
 –personAttrs [ [gender male] [age 29] [citizership USA] ] -hobbyList [swim tennis baseball]
 -favorFoodTable [ [juice orange] [fruit apple] ] ]')
```

where:

Table 46. AdminTask parameter descriptions. Run a command from a wsadmin command line.

Parameter	Data type	Example value
personName	String	`John`
personalInfo	java.util.Properties	`[ [cellPhone 123-456-7890] [workPhone 123-456-7892] [homePhone 123-456-7891] ]`
pets	String[]	`[dog cat]`
personID	javax.management.ObjectName	`WebSphere:John(organization=ibm,country=usa,state=texas,city=austin)`
personAttrs	javax.management.AttributeList	`[ [gender male] [age 29] [citizenship USA] ]`
hobbyList	java.util.ArrayList	`[swim tennis baseball]`
favorFoodTable	java.util.Hashtable	`[ [juice orange] [fruit apple] ]`

Chapter 10. Starting the wsadmin scripting client using wsadmin scripting

You can use the wsadmin tool to configure and administer application servers, application deployment, and server runtime operations.

About this task

The wsadmin tool provides the ability to automate configuration tasks for your environment by running scripts. However, there are some limitations for using the wsadmin tool, including:

- The wsadmin tool only supports the Jython and Jacl scripting languages.

  The Version 6.1 release of WebSphere Application Server represented the start of the deprecation process for the Jacl syntax that is associated with the wsadmin tool. The Jacl syntax for the wsadmin tool continues to remain in the product and is supported for at least two major product releases. After that time, the Jacl language support might be removed from the wsadmin tool. The Jython syntax for the wsadmin tool is the strategic direction for WebSphere Application Server administrative automation. The application server provides significantly enhanced administrative functions and tooling that support product automation and the use of the Jython syntax.

  gotcha: Not all of the WebSphere Application Server component classes are packaged in the same `.jar` file. If you are going to be using the wsadmin tool to run Jython scripts, include the jython.package.path system property on your wsadmin command to ensure that all of the required JAR files are set to the jython package path during wsadmin startup.

  ```
  ./wsadmin.sh -lang jython -javaoption
  "-Djython.package.path=/usr/WebSphere70/AppServer/plugins/com.ibm.ws.wlm.jar"
  ```

  If you want to invoke WebSphere Application Server functions from different WebSphere Application Server classes that are packaged in `.jar` files other than `runtime.jar` and `admin.jar`, you can include multiple jar files in the path specified for the jython.package.path system property, and separate them with semicolon (;).

  ```
  ./wsadmin.sh -lang jython -javaoption
  "-Djython.package.path=/usr/WebSphere70/AppServer/plugins/com.ibm.ws.wlm.jar;com.ibm.ws.wccm.jar"
  ```

  If you want to invoke WebSphere Application Server functions in a jython script using ws_ant, you can create a `.prop` file text file, and include the following line in this file:

  ```
  jython.package.path=/usr/WebSphere70/AppServer/plugins/com.ibm.ws.wlm.jar
  ```

  Then include the property file in the ant script xml file. For example:

  ```
  <taskdef name="wsadmin" classname="com.ibm.websphere.ant.tasks.WsAdmin"/>
  <target name="main" >
      <wsadmin conntype="NONE" lang="jython" failonerror="true" properties="/tmp/jython.prop"
               script="/home/fsgapp/MSTWasBuild/project/scripts/socr/socr/jython/configure.py">
      </wsadmin>
  </target>
  ```

- The wsadmin tool manages the installation, configuration, deployment, and runtime operations for application servers that run the same version or a higher version of the product. The wsadmin tool cannot connect to an application server that runs a product version which is older than the version of the wsadmin tool. For example, a Version 6.x wsadmin client cannot connect to a Version 5.x application server. However, a Version 5.x wsadmin client can connect to a Version 6.x application server. This limitation exists because new functionality is added to the wsadmin tool in each product release. You cannot use new command functionality on application servers running previous product versions.

Procedure

1. Locate the command that starts the wsadmin scripting client.

   Choose one of the following:

   - Invoke the scripting process using a specific profile. The QShell command for invoking a scripting process is located in the *profile_root*/bin directory. The name of the QShell script is wsadmin. If you use this option, you do not need to specify the -profileName *profilename* parameter.
   - Invoke the scripting process using the default profile. The wsadmin Qshell command is located in the *app_server_root*/bin directory. If you do not want to connect to the default profile, you must specify the -profileName *profilename* parameter to indicate the profile that you want to use.

2. Review additional connection options for the wsadmin tool.

   You can start the wsadmin scripting client in several different ways. To specify the method for running scripts, perform one of the following wsadmin tool options:

   Run scripting commands interactively

   Run wsadmin with an option other than -f or -c or without an option. The wsadmin tool starts and displays an interactive shell with a wsadmin prompt. From the wsadmin prompt, enter any Jacl or Jython command. You can also invoke commands using the AdminControl, AdminApp, AdminConfig, AdminTask, or Help wsadmin objects. To leave an interactive scripting session, use the quit or exit commands. These commands do not take any arguments.

   The following examples launch the wsadmin tool:

   - Launch the wsadmin tool using Jython:

     ```
     wsadmin -lang jython
     ```

   - Launch the wsadmin tool using Jython when security is enabled:

     ```
     wsadmin -lang jython -user wsadmin -password wsadmin
     ```

   - Launch the wsadmin tool using Jacl with no options:

     ```
     wsadmin -lang jacl
     ```

   Run scripting commands as individual commands

   Run the wsadmin tool with the -c option.

   The following examples run commands individually:

   - Run the list command for the AdminApp object using Jython:

     ```
     wsadmin -lang jython -c "AdminApp.list()"
     ```

   - Run the list command for the AdminApp object using Jacl:

     ```
     wsadmin -c "$AdminApp list"
     ```

   Run scripting commands in a script

   Run the wsadmin tool with the -f option, and place the commands that you want to run into the file.

   The following examples run scripts:

   - Run the a1.py script using Jython:

     ```
     wsadmin -lang jython -f  a1.py
     ```

   where the a1.py file contains the following commands:

   ```
   apps = AdminApp.list()
   print apps
   ```

   Run scripting commands in a profile script

   A *profile script* is a script that runs before the main script, or before entering interactive mode. You can use profile scripts to set up a scripting environment that is customized for the user or the installation.

By default, the following profile script files might be configured for the `com.ibm.ws.scripting.profiles` profiles property in the *app_server_root*`/properties/wsadmin.properties` file:

- *app_server_root*`/bin/securityProcs.jacl`
- *app_server_root*`/bin/LTPA_LDAPSecurityProcs.jacl`

By default, these files are in ASCII. If you use the `profile.encoding` option to run EBCDIC encoded profile script files, change the encoding of the files to EBCDIC.

To run scripting commands in a profile script, run the wsadmin tool with the -profile option, and include the commands that you want to run into the profile script.

To customize the script environment, specify one or more profile scripts to run.

Do not use parenthesis in node names when creating profiles.

The following examples run profile scripts:

- Run the `a1prof.py` script using Jython:

```
wsadmin -lang jython -profile alprof.py
```

where the `a1prof.py` file contains the following commands:

```
apps = AdminApp.list()
print "Applications currently installed:\n " + apps
```

- Run the `a1prof.py` script using Jacl:

```
wsadmin -profile alprof.jacl
```

where the `a1prof.py` file contains the following commands:

```
set apps [$AdminApp list]
puts "Applications currently installed:\n$apps"
```

Results

The wsadmin returns the following output when it establishes a connection to the server processs:

Jython example output:

```
WASX7209I: Connected to process server1
on node myhost using SOAP connector;
The type of process is: UnManagedProcess
WASX7029I: For help, enter: "$Help help"
wsadmin>
```

Jacl example output:

```
WASX7209I: Connected to process server1
on node myhost using SOAP connector;
The type of process is: UnManagedProcess
WASX7029I: For help, enter: "$Help help"
wsadmin>
```

Chapter 11. Using the script library to automate the application serving environment using wsadmin scripting

The script library provides Jython script procedures to assist in automating your environment. Use the sample scripts to manage applications, resources, servers, nodes, and clusters. You can also use the script procedures as examples to learn the Jython syntax.

About this task

The Jython script library provides a set of procedures to automate the most common application server administration functions. For example, you can use the script library to easily configure servers, applications, mail settings, resources, nodes, business-level applications, clusters, authorization groups, and more. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
      "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication",
      "..\installableApps\DefaultApplication.ear", "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

Each script from the script library directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory, and save existing automation scripts in the *app_server_root*/scriptLibraries directory. Each script library name must be unique and cannot be duplicated.

Note: Do not edit the script procedures in the script library. To customize script library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library.

To automatically load Jython scripts (*.py) that are not located in the *app_server_root*/scriptLibraries directory when the wsadmin tool starts, set the `wsadmin.script.libraries` system property to the script location. For example, if your script libraries are saved in the temp directory on a Windows operating system, the following example sets the script path in the wsadmin command line tool:

```
bin>wsadmin -lang jython -javaoption "-Dwsadmin.script.libraries=c:/myJythonScripts"
```

To load multiple directories, specify each directory in the system property separated by a semicolon (;), as the following example demonstrates:

```
bin>wsadmin -lang jython -javaoption "-Dwsadmin.script.libraries=c:/myJythonScripts;c:/AdminScripts;c:/configScripts"
```

The script library provides automation scripts for the following application server administration functions:

Procedure

- Manage application servers. You can use the AdminServerManagement scripts to configure classloaders, Java virtual machine (JVM) settings, Enterprise JavaBeans (EJB) containers, performance monitoring, dynamic cache, and so on.
- Manage server and system architecture. You can use the AdminServerManagement script library to manage server settings.
- Manage applications. You can use the AdminApplication scripts to install, uninstall, and update your applications with various options.
- Manage data access resources. You can use the AdminJDBC and AdminJ2C script libraries to manage data sources and Java Database Connectivity (JDBC) providers, and to create and configure Java 2 Connector (J2C) resource adapters.
- Manage messaging resources. You can use the AdminJMS script library to configure and manage your Java Messaging Service (JMS) configurations.
- Manage mail resources. You can use the AdminResources scripts in the script library to configure mail, URL, and resource settings.
- Managing authorization groups. You can use the AdminAuthorizations scripts to configure authorization groups.
- Monitor performance and troubleshoot configurations. You can use the AdminUtilities scripts to configure trace, debugging, logs, and performance monitoring. See the Utility scripts topic.
- Get script library help using wsadmin You can use the AdminLibHelp script library to list each available script library, display information for specific script libraries, and to display information for specific script procedures.

What to do next

Determine which scripts to use to automate your environment, or create custom scripts using assembly tools.

Automating server administration using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the server management scripts to configure servers, the server runtime environment, Web containers, performance monitoring, and logs. You can also use the scripts to administer your servers.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

```
#
# My Custom Jython Script - file.py
#
AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
    "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The AdminServerManagement procedures in scripting library are located in the *app_server_root*/scriptLibraries/servers/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminServerManagement.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. Use the following steps to create an application server, connect the application server to the AdminService interface, configure Java virtual machine (JVM) settings, add the application server to a cluster, and propagate the changes to the node.

Procedure

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, job manager, or administrative agent profile, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Create an application server.

Run the createApplicationServer script procedure from the AdminServerManagement script library, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminServerManagement.createApplicationServer("myNode", "myServer", "default")"
```

You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
```

3. Connect the application server of interest to the AdminService interface.

   The AdminService interface is the server interface to the application server administration functions. To connect the application server to the AdminService interface, run the configureAdminService script procedure from the AdminServerManagement script library, specifying the node name, server name, and connector type arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminServerManagement.configureAdminService("myNode", "myServer",
"IPC", "JSR160RMI")
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminServerManagement.configureAdminService("myNode", "myServer", "IPC", "JSR160RMI")
```

4. Configure the Java virtual machine (JVM).

   As part of configuring an application server, you might define settings that enhance the way your operating system uses of the JVM. The JVM is an interpretive computing engine responsible for running the byte codes in a compiled Java program. The JVM translates the Java byte codes into the native instructions of the host machine. The application server, being a Java process, requires a JVM in order to run, and to support the Java applications running on it.

   Run the configureJavaVirtualMachine script procedure from the AdminServerManagement script library, specifying the node name, server name, whether to run the JVM in debug mode, and any debug arguments to pass to the JVM process. You can optionally specify additional configuration attributes with an attribute list. Use the following example to configure the JVM:

```
bin>wsadmin -lang jython -c "AdminServerManagement.configureJavaVirtualMachine("myNode",
"myServer", "true", "mydebug", [["internalClassAccessMode", "RESTRICT"],
     ["disableJIT", "false"], ["verboseModeJNI", "false"]])"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminServerManagement.configureJavaVirtualMachine("myNode", "myServer", "true",
"mydebug", [["internalClassAccessMode", "RESTRICT"],
   ["disableJIT", "false"], ["verboseModeJNI", "false"]])
```

5. Create a cluster, and add the application server as a cluster member.

   Run the createClusterWithFirstMember script procedure from the AdminClusterManagement script library, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminClusterManagement.createClusterWithFirstMember("myCluster",
"APPLICATION_SERVER", "myNode", "myServer")"

wsadmin>AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
"myNode", "myServer")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Server settings configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the application server scripts to configure class loaders, Java Virtual Machine (JVM) settings, Enterprise JavaBeans (EJB) containers, performance monitoring, dynamic cache, and so on. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/`scriptLibraries/servers/V70` directory.

Use the following script procedures to administer your application server:

Use the following script procedures to configure your application server runtime environment:

Use the following script procedures to configure web containers for your application server:

Use the following script procedures to configure logs and monitor performance for your application server:

configureAdminService

This script configures settings for the AdminService interface. The AdminService interface is the server-side interface to the application server administration functions.

Table 47. configureAdminService argument descriptions. Run the script with the node name, server name, local connection protocol, and remote connection protocol.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
localAdminProtocol	Specifies the type of connector to use to connect the AdminService interface to the application server for local connection.
remoteAdminProtocol	Specifies the type of connector to use to connect the AdminService interface to the application server for remote connection.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["enabled", "true"], ["pluginConfigService", "(cells/timmieNode02Cell/nodes/timmieNode01/servers/server1|server.xml#PluginConfigService_1183122130078)"]]`

Syntax

```
AdminServerManagement.configureAdminService(nodeName, serverName, localAdminProtocol, remoteAdminProtocol, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureAdminService("myNode", "myServer", "IPC", "SOAP",
    [["enabled", "true"], ["pluginConfigService",
       "(cells/timmieNode02Cell/nodes/timmieNode01/servers/server1|server.xml#PluginConfigService_1183122130078)"]])
```

configureApplicationServerClassloader

This script configures a class loader for the application server. Class loaders enable applications that are deployed on the application server to access repositories of available classes and resources.

Table 48. configureApplicationServerClassloader argument descriptions. Run the script with the node name, server name, policy, mode, and library name arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
policy	Specifies the application class loader policy as `SINGLE` or `MULTIPLE`. Specify the `SINGLE` value to prevent the isolation applications, and to configure the application server to use a single application class loader to load all of the EJB modules, shared libraries, and dependency Java archive (JAR) files in the system. Specify the `MULTIPLE` value to isolate applications and provide each application with its own class loader to load EJB modules, shared libraries, and dependency JAR files.
mode	Specifies the class loader mode as `PARENT_FIRST` or `APPLICATION_FIRST`. The `PARENT_FIRST` option causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. The `APPLICATION_FIRST` option causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.
libraryName	Specifies the name of the shared library of interest.

Syntax

```
AdminServerManagement.configureApplicationServerClassloader(nodeName, serverName,
 policy, mode, libraryName)
```

Example usage

```
AdminServerManagement.configureApplicationServerClassloader("myNode", "MULTIPLE", "PARENT_FIRST",
"myLibraryReference")
```

configureDynamicCache

This script configures the dynamic cache service in your server configuration. The dynamic cache service works within an application server JVM, intercepting calls to cacheable objects. For example, the dynamic cache service intercepts calls through a servlet service method or a command execute method, and either stores the output of the object to the cache or serves the content of the object from the dynamic cache.

Table 49. configureDynamicCache argument descriptions. Run the script with the node name, server name, default priority, cache size, external cache group name, and external cache group type arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
defaultPriority	Specifies the default priority for cache entries, determining how long an entry stays in a full cache. Specify an integer between 1 and 255.
cacheSize	Specifies a positive integer as the value for the maximum number of entries that the cache holds. Enter a cache size value in this field that is between the range of 100 through 200000.
externalCacheGroupName	The external cache group name needs to match the ExternalCache property as defined in the servlet or JavaServer Pages (JSP) file cachespec.xml file. When external caching is enabled, the cache matches pages with its Universal Resource Identifiers (URI) and pushes matching pages to the external cache. The entries can then be served from the external cache, instead of from the application server.
externalCacheGroupType	Specifies the external cache group type.
otherAttributeList	Optionally specifies additional configuration options for the dynamic cache service in the following format: `[["cacheProvider", "myProvider"], ["diskCacheCleanupFrequency", 2], ["flushToDiskOnStop", "true"]]`

Syntax

```
AdminServerManagement.configureDynamicCache(nodeName, serverName, defaultPriority,
    cacheSize, externalCacheGroupName, externalCacheGroupType,
    otherAttributeList)
```

Example usage

```
AdminServerManagement.configureDynamicCache("myNode", "myServer", 2, 5000, "EsiInvalidator",
   "SHARED", [["cacheProvider", "myProvider"], ["diskCacheCleanupFrequency", 2], ["flushToDiskOnStop","true"]])
```

configureEJBContainer

This script configures an Enterprise JavaBeans (EJB) container in your server configuration. An EJB container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean's operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

Table 50. configureEJBContainer argument descriptions. Run the script with the node name, server name, passivation directory, and default datasource Java Naming and Directory Interface (JNDI) name arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
passivationDirectory	Specifies the directory into which the container saves the persistent state of passivated stateful session beans. This directory must already exist. It is not automatically created.
defaultDatasourceJNDIName	Specifies the JNDI name of a data source to use if no data source is specified during application deployment. This setting is not applicable for EJB 2.x-compliant container-managed persistence beans.

Syntax

```
AdminServerManagement.configureEJBContainer(nodeName, serverName,
passivationDir, defaultDatasourceJNDIName)
```

Example usage

```
AdminServerManagement.configureEJBContainer("myNode", "myServer", "/temp/myDir", "jndi1")
```

configureFileTransferService

This script configures the file transfer service for the application server. The file transfer service transfers files from the deployment manager to individual remote nodes.

Table 51. configureFileTransferService argument descriptions. Run the script with the node name, server name, number of times to retry the file transfer, and the time to wait before retrying the file transfer.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
retriesCount	Specifies the number of times you want the file transfer service to retry sending or receiving a file after a communication failure occurs. The default value is `3`.
retryWaitTime	Specifies the number of seconds that the file transfer service waits before it retries a failed file transfer. The default value is `10`.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["enable", "true"]]`

Syntax

```
AdminServerManagement.configureFileTransferService(nodeName, serverName, retriesCount, retryWaitTime, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureFileTransferService("myNode", "myServer", 5, 600, [["enable", "true"]])
```

configureListenerPortForMessageListenerService

This script configures the listener port for the message listener service in your server configuration. The message listener service is an extension to the Java Messaging Service (JMS) functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

Table 52. configureListenerPortForMessageListenerService argument descriptions. Run the script with the node name, server name, listener port name, connection factory JNDI name, destination JNDI name, maximum number of messages, maximum number of retries, and the maximum session arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
listenerPortName	The name by which the listener port is known for administrative purposes.
connectionFactoryJNDIName	The JNDI name for the JMS connection factory to be used by the listener port; for example, `jms/connFactory1`.
destinationJNDIName	The JNDI name for the destination to be used by the listener port; for example, `jms/destn1`
maxMessages	The maximum number of messages that the listener can process in one transaction. If the queue is empty, the listener processes each message when it arrives. Each message is processed within a separate transaction.
maxRetries	The maximum number of times that the listener tries to deliver a message before the listener is stopped, in the range 0 through 2147483647. The maximum number of times that the listener tries to deliver a message to a message-driven bean instance before the listener is stopped.
maxSession	Specifies the maximum number of concurrent sessions that a listener can have with the JMS server to process messages. Each session corresponds to a separate listener thread and therefore controls the number of concurrently processed messages. Adjust this parameter when the server does not fully use the available capacity of the machine and if you do not need to process messages in a specific message order.

Syntax

```
AdminServerManagement.configureListenerPortForMessageListener(nodeName, serverName,
 listenerPortName, connectionFactoryJNDIName,
     destinationJNDIName, maxMessages, maxRetries, maxSession)
```

Example usage

```
AdminServerManagement.configureListenerPortForMessageListener("myNode", "myServer", "myListenerPort",
 "connJNDI", "destJNDI", 5, 2, 3)
```

configureMessageListenerService

This script configures the message listener service in your server configuration. The message listener service is an extension to the Java Messaging Service (JMS) functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

Table 53. configureMessageListenerService argument descriptions. Run the script with the node name, server name, maximum number of message listener retries, listener recovery interval, pooling threshold, and pooling timeout arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
maxListenerRetry	Specifies the maximum number of times that a listener port managed by this service tries to recover from a failure before giving up and stopping. When stopped the associated listener port is changed to the stop state.
listenerRecoveryInterval	Specifies the time in seconds between retry attempts by a listener port to recover from a failure.
poolingThreshold	Specifies the maximum number of unused connections in the pool. The default value is `10`.
poolingTimeout	Specifies the number of milliseconds after which a connection in the pool is destroyed if it has not been used. An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes.
otherAttributeList	Optionally specifies additional message listener attributes in the following format: `[["description", "test message listener"], ["isGrowable", "true"], ["maximumSize", 100], ["minimumSize", 5]]`

Syntax

```
AdminServerManagement.configureMessageListenerService(nodeName, serverName,
 maxListenerRetry, listenerRecoveryInterval,
     poolingThreshold, poolingTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureMessageListenerService("myNode", "myServer", 5, 120,
 20, 600000, "myProp", "myValue",
      [["description", "test message listener"], ["isGrowable", "true"], ["maximumSize", 100], ["minimumSize", 5]])
```

configureStateManageable

This script configures the initial state of the application server. The initial state refers to the desired state of the component when the server process starts.

Table 54. configureStateManageable argument descriptions. Run the script with the node name, server name, parent type, and initial state arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
parentType	Specifies the type of component to modify.
initialState	Specifies the desired state of the component when the server process starts. Valid values are `START` and `STOP`.

Syntax

```
AdminServerManagement.configureStateManageable(nodeName, serverName,
 parentType, initialState)
```

Example usage

```
AdminServerManagement.configureStateManageable("myNode", "myServer", "Server", "START")
```

configureCustomProperty

This script configures custom properties in your application server configuration. You can use custom properties for configuring internal system properties which some components use, for example, to pass information to a web container.

Table 55. configureCustomProperty argument descriptions. Run the script with the node name, server name, parent type, property name, and property value arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
parentType	Specifies the type of component to configure.
propertyName	Specifies the custom property to configure.
propertyValue	Specifies the value of the custom property to configure.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["commTraceEnabled", "true"], ["enable", "true"]]`

Syntax

```
AdminServerManagement.configureCustomProperty(nodeName, serverName, parentType, propertyName, propertyValue, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureCustomProperty("myNode", "myServer", "ThreadPool", "myProp1", "myPropValue",
    [["description", "my property test"], ["required", "false"]])
```

configureCustomService

This script configures a custom service in your application server configuration. Each custom services defines a class that is loaded and initialized whenever the server starts and shuts down. Each of these classes must implement the com.ibm.websphere.runtime.CustomService interface. After you create a custom service, use the administrative console to configure that custom service for your application servers.

Table 56. configureCustomService argument descriptions. Run the script with the node name, server name, and preferred connector type.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
classname	Specifies the class name of the service implementation. This class must implement the Custom Service interface.
displayname	Specifies the name of the service.
classpath	Specifies the class path used to locate the classes and JAR files for this service.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["description", "test custom service"], ["enable", "true"]]`

Syntax

```
AdminServerManagement.configureCustomService(nodeName, serverName, classname, displayname, classpath, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureCustomService("myNode", "myServer", "myClass", "myName", "/temp/boo.jar",
     [["description", "test custom service"], ["enable", "true"]])
```

configureEndPointsHost

Table 57. configureEndPointsHost argument descriptions. Run the script to configure the host name of the server endpoints. Specify the node name, server name, and host name arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
hostName	Specifies the name of the host of interest.

Syntax

```
AdminServerManagement.configureEndPointsHost(nodeName, serverName, hostName)
```

Example usage

```
AdminServerManagement.configureEndPointsHost("myNode", "AppServer01", "myHostname")
```

configureJavaVirtualMachine

This script configures a Java virtual machine (JVM). The application server, being a Java process, requires a JVM in order to run, and to support the Java applications running on it.

Table 58. configureJavaVirtualMachine argument descriptions. Run the script with the configuration ID of the JVM of interest, whether to enable debug mode, and additional debug arguments.

Argument	Description
javaVirtualMachineConfigID	Specifies the configuration ID of the Java virtual machine you want to make changes.
debugMode	Specifies whether to run the JVM in debug mode. The default is not to enable debug mode support. If you set the debugMode argument to `true`, then you must specify debug arguments.
debugArgs	Specifies debug arguments to pass to the JVM code that starts the application server process. If you enable debugging on multiple application servers on the same node, make sure that the servers are using different address arguments, which define the port for debugging. For example, if you enable debugging on two servers and leave the default debug port for each server as `address=7777`, the servers could fail to start properly.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["internalClassAccessMode", "RESTRICT""], ["disableJIT", "false"], ["verboseModeJNI", "false"]]`

Syntax

```
AdminServerManagement.configureJavaVirtualMachine(javaVirtualMachineConfigID,
 debugMode, debugArgs, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureJavaVirtualMachine
   ("(cells/WAS00Network/nodes/ndnode1/servers/server1|server.xml#JavaVirtualMachine_1208188803955)", "true",
   "mydebug", [["internalClassAccessMode", "RESTRICT"], ["disableJIT", "false"], ["verboseModeJNI", "false"]])
```

configureORBService

This script configures an Object Request Broker (ORB) service in your server configuration. An Object Request Broker (ORB) manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.

Table 59. configureORBService argument descriptions. Run the script with the node name, server name, request timeout, request retry count, request retry delay, maximum connection cache, minimum connection cache, and locate request timeout arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Table 59. configureORBService argument descriptions (continued). Run the script with the node name, server name, request timeout, request retry count, request retry delay, maximum connection cache, minimum connection cache, and locate request timeout arguments.

Argument	Description
requestTimeout	Specifies the number of seconds to wait before timing out on a request message.
requestRetriesCount	Specifies the number of times that the ORB attempts to send a request if a server fails. Retrying sometimes enables recovery from transient network failures. This field is ignored on the z/OS® platform.
requestRetriesDelay	Specifies the number of milliseconds between request retries. This field is ignored on the z/OS platform.
connectionCacheMax	Specifies the maximum number of entries that can occupy the ORB connection cache before the ORB starts to remove inactive connections from the cache. This field is ignored on the z/OS platform. It is possible that the number of active connections in the cache will temporarily exceed this threshold value. If necessary, the ORB will continue to add connections as long as resources are available.
connectionCacheMin	Specifies the minimum number of entries in the ORB connection cache. This field is ignored on the z/OS platform. The ORB will not remove inactive connections when the number of entries is below this value.
locateRequestTimeout	Specifies the number of seconds to wait before timing out on a LocateRequest message. This field is ignored on the z/OS platform.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["commTraceEnabled", "true"], ["enable", "true"]]`

Syntax

```
AdminServerManagement.configureORBService(nodeName, serverName, requestTimeout, requestRetriesCount, requestRetriesDelay,
   connectionCacheMax, connectionCacheMin, locateRequestTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureMessageListenerService("myNode", "myServer", 5, 120, 20, 600000, 20, 300,
[["commTraceEnabled", "true"], ["enable", "true"]])
```

configureProcessDefinition

This script configures the server process definition. Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.

To run the script, specify the node name and server name arguments, as defined in the following table:

Table 60. configureProcessDefinition argument descriptions. Run the script with the node name, server name and, as needed, additional parameters.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
otherParamList	Specifies additional parameters for the process definition configuration in the following format: `[["executableName", "value1"], ["executableArguments", "value2"], ["workingDirectory", "value3"]]`

Syntax

```
AdminServerManagement.configureProcessDefintion(nodeName, serverName, otherParamList)
```

Example usage

```
AdminServerManagement.configureProcessDefinition("myNode", "myServer",
   [["executableName", "value1"],["executableArguments"."value2"],["workingDirectory","value3"]])
```

configureRuntimeTransactionService

This script configures the transaction service for your server configuration. The transaction service is a server runtime component that coordinates updates to multiple resource managers to ensure atomic updates of data. Transactions are started and ended by applications or the container in which the applications are deployed.

Table 61. configureRuntimeTransactionService argument descriptions. Run the script with the node name, server name, total transaction lifetime timeout, and client inactivity timeout arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
totalTranLifetimeTimeout	Specifies the default maximum time, in seconds, allowed for a transaction that is started on this server before the transaction service initiates timeout completion. Any transaction that does not begin completion processing before this timeout occurs is rolled back.
clientInactivityTimeout	Specifies the maximum duration, in seconds, between transactional requests from a remote client. Any period of client inactivity that exceeds this timeout results in the transaction being rolled back in this application server. If you set this value to 0, there is no timeout limit.

Syntax

```
AdminServerManagement.configureRuntimeTransactionService(nodeName, serverName,
 totalTranLifetimeTimeout, clientInactivityTimeout)
```

Example usage

```
AdminServerManagement.configureRuntimeTransactionService("myNode", "myServer", 600, 600)
```

configureThreadPool

This script configures thread pools in your server configuration. A thread pool enables components of the server to reuse threads, which eliminates the need to create new threads at run time. Creating new threads expends time and resources.

Table 62. configureThreadPool argument descriptions. Run the script with the node name, server name, parent type, thread pool name, maximum size, minimum size, and the amount of time before timeout occurs.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
parentType	Specifies the type of component to configure.
threadPoolName	Specifies the name of the thread pool of interest.
maximumSize	Specifies the maximum number of threads to maintain in the default thread pool. If your Tivoli® Performance Viewer shows the Percent Maxed metric to remain consistently in the double digits, consider increasing the Maximum size. The Percent Maxed metric indicates the amount of time that the configured threads are used.
minimumSize	Specifies the minimum number of threads to allow in the pool. When an application server starts, no threads are initially assigned to the thread pool. Threads are added to the thread pool as the workload assigned to the application server requires them, until the number of threads in the pool equals the number specified in the Minimum size field. After this point in time, additional threads are added and removed as the workload changes. However the number of threads in the pool never decreases below the number specified in the Minimum size field, even if some of the threads are idle.
inactivityTimeout	Specifies the number of milliseconds of inactivity that should elapse before a thread is reclaimed. A value of `0` indicates not to wait and a negative value (less than 0) means to wait forever.
otherAttributeList	Specifies additional configuration attributes in the following format: `[["description", "testing thread pool"], ["isGrowable", "true"], ["name", "myThreadPool"]]`

Syntax

```
AdminServerManagement.configureThreadPool(nodeName, serverName,
 parentType, threadPoolName, maximumSize,
 minimumSize, inactivityTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureThreadPool
   ("acmeNode2", "server1", "ThreadPoolManager", "WebContainer", 15, 25, 60)
```

configureTransactionService

This script configures the transaction service for your application server. You can use transactions with your applications to coordinate multiple updates to resources as atomic units (as indivisible units of work) such that all or none of the updates are made permanent.

Table 63. configureTransactionService argument descriptions. Run the script with the node name, server name, total transaction lifetime timeout, client inactivity timeout, maximum transaction timeout, heuristic retry limit, heuristic retry wait, propogate or BMT transaction lifetime timeout, and asynchronous response timeout arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the server name of the of interest.
totalTranLifetimeTimeout	Specifies the default maximum time, in seconds, allowed for a transaction that is started on this server before the transaction service initiates timeout completion. Any transaction that does not begin completion processing before this timeout occurs is rolled back. This timeout is used only if the application component does not set its own transaction timeout. Only the total transaction lifetime timeout and the maximum transaction timeout have grace periods. You can disable the grace periods using the DISABLE_TRANSACTION_TIMEOUT_GRACE_PERIOD custom property.
clientInactivityTimeout	Specifies the maximum duration, in seconds, between transactional requests from a remote client. Any period of client inactivity that exceeds this timeout results in the transaction being rolled back in this application server. If you set this value to 0, there is no timeout limit.
maximumTransactionTimeout	Specifies the upper limit of the transaction timeout, in seconds, for transactions that run in this server. This value should be greater than or equal to the total transaction timeout. This timeout constrains the upper limit of all other transaction timeouts.
heuristicRetryLimit	Specifies the number of times that the application server retries a completion signal, such as commit or rollback. Retries occur after a transient exception from a resource manager or remote partner, or if the configured asynchronous response timeout expires before all Web Services Atomic Transaction (WS-AT) partners have responded.
heuristicRetryWait	Specifies the number of seconds that the application server waits before retrying a completion signal, such as commit or rollback, after a transient exception from a resource manager or remote partner.
propogateOrBMTTranLifetimeTimeout	Specifies the number of seconds that a transaction remains inactive before it is rolled back.
asyncResponseTimeout	Specifies the amount of time, in seconds, that the server waits for an inbound Web Services Atomic Transaction (WS-AT) protocol response before resending the previous WS-AT protocol message.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["LPSHeuristicCompletion", "ROLLBACK"], ["WSTransactionSpecificationLevel", "WSTX_10"], ["enable", "true"]]`

Syntax

```
AdminServerManagement.configureTransactionService(nodeName, serverName,
totalTranLifetimeTimeout, clientInactivityTimeout,
   maximumTransactionTimeout, heuristicRetryLimit, heuristicRetryWait,
   propogateOrBMTTranLifetimeTimeout, asyncResponseTimeout, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureTransactionService("myNode", "myServer",
    120, 60, 5, 2, 5, 300, 30,
    [["LPSHeuristicCompletion", "ROLLBACK"], ["WSTransactionSpecificationLevel", "WSTX_10"], ["enable", "true"]])
```

setJVMProperties

This script sets additional properties for your JVM configuration.

Table 64. setJVMProperties argument descriptions. Run the script with the node name, server name, classpath, boot class path, initial heap size, maximum heap size, whether to enable debug mode, and debug arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.

Table 64. setJVMProperties argument descriptions (continued). Run the script with the node name, server name, classpath, boot class path, initial heap size, maximum heap size, whether to enable debug mode, and debug arguments.

Argument	Description
serverName	Specifies the name of the server of interest.
classPath	Optionally specifies the standard class path in which the Java virtual machine code looks for classes.
bootClasspath	Optionally specifies bootstrap classes and resources for JVM code. This option is only available for JVM instructions that support bootstrap classes and resources.
initialHeapSize	Optionally specifies the initial heap size available to the JVM code, in megabytes. Increasing the minimum heap size can improve startup. The number of garbage collection occurrences are reduced and a 10% gain in performance is realized. Increasing the size of the Java heap improves throughput until the heap no longer resides in physical memory, in general. After the heap begins swapping to disk, Java performance suffers drastically.
maxHeapSize	Optionally specifies the maximum heap size available to the JVM code, in megabytes. Increasing the heap size can improve startup. By increasing heap size, you can reduce the number of garbage collection occurrences with a 10% gain in performance.
debugMode	Optionally specifies whether to run the JVM in debug mode. The default is not to enable debug mode support. If you set the debugMode argument to `true`, then you must specify debug arguments.
debugArgs	Optionally specifies debug arguments to pass to the JVM code that starts the application server process. If you enable debugging on multiple application servers on the same node, make sure that the servers are using different address arguments, which define the port for debugging. For example, if you enable debugging on two servers and leave the default debug port for each server as `address=7777`, the servers could fail to start properly.

Syntax

```
AdminServerManagement.setJVMProperties(nodeName, serverName,
classPath, bootClasspath, initialHeapSize,
maxHeapSize, debugMode, debugArgs)
```

Example usage

```
AdminServerManagement.setJVMProperties("myNode", "myServer", "/a.jar", "", "", "", "", "")
```

setTraceSpecification

This script sets the trace specification for your configuration.

Table 65. setTraceSpecification argument descriptions. Run the script with the node name, server name, and trace specification arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
traceSpecification	Optionally specifies debug arguments to pass to the JVM code that starts the application server process. If you enable debugging on multiple application servers on the same node, make sure that the servers are using different address arguments, which define the port for debugging. For example, if you enable debugging on two servers and leave the default debug port for each server as `address=7777`, the servers could fail to start properly.

Syntax

```
AdminServerManagement.setJVMProperties(nodeName, serverName, traceSpecification)
```

Example usage

```
AdminServerManagement.setTraceSpecification("myNode", "myServer", "com.ibm.ws.management.*=all")
```

configureCookieForServer

This script configures cookies in your application server configuration. Configure cookies to track sessions.

Table 66. configureCookieForServer argument descriptions. Run the script with the node name, server name, cookie name, domain, maximum cookie age, and whether to secure the cookie.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
cookieName	Specifies a unique name for the session management cookie. The servlet specification requires the name `JSESSIONID`. However, for flexibility this value can be configured.
domain	Specifies the domain field of a session tracking cookie. This value controls whether or not a browser sends a cookie to particular servers. For example, if you specify a particular domain, session cookies are sent to hosts in that domain. The default domain is the server.
maximumAge	Specifies the amount of time that the cookie lives on the client browser. Specify that the cookie lives only as long as the current browser session, or to a maximum age. If you choose the maximum age option, specify the age in seconds. This value corresponds to the Time to Live (TTL) value described in the Cookie specification. Default is the current browser session which is equivalent to setting the value to `-1`.
secure	Specifies that the session cookies include the secure field. Enabling the feature restricts the exchange of cookies to HTTPS sessions only.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["path", "C:/temp/mycookie"]]`

Syntax

```
AdminServerManagement.configureCookieForServer(nodeName, serverName, cookieName, domain, maximumAge, secure, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureCookieForServer("myNode", "myServer", "myCookie", "uk.kingdom.com", -1, "true", [["path", "C:/temp/mycookie"]])
```

configureHTTPTransportForWebContainer

This script configures HTTP transports for a web container. Transports provide request queues between application server plug-ins for Web servers and web containers in which the web modules of applications reside. When you request an application in a web browser, the request is passed to the web server, then along the transport to the web container.

Table 67. configureHTTPTransportForWebContainer argument descriptions. Run the script with the node name, server name, whether to adjust the port, whether external, the Secure Socket Layer (SSL) configuration to use, and whether to enable SSL.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
adjustPort	Specifies whether to automatically adjust the port for the web container of interest.
external	Specifies whether to set the HTTP Transport for the web container to external.
sslConfig	Specifies the Secure Sockets Layer (SSL) settings type for connections between the WebSphere Application Server plug-in and application server. The options include one or more SSL settings defined in the Security Center; for example, DefaultSSLSettings, ORBSSLSettings, or LDAPSSLSettings.
sslEnabled	Specifies whether to protect connections between the WebSphere Application Server plug-in and application server with Secure Sockets Layer (SSL). The default is not to use SSL.

Syntax

```
AdminServerManagement.configureHTTPTransportForWebContainer(nodeName, serverName,
 adjustPort, external, sslConfig, sslEnabled)
```

Example usage

```
AdminServerManagement.configureHTTPTransportForWebContainer("myNode", "myServer", "true", "true", "mySSLConfig", "true")
```

configureSessionManagerForServer

This script configures the session manager for the application server. Sessions allow applications running in a web container to keep track of individual users.

Table 68. configureSessionManagerForServer argument descriptions. Run the script with the node name, server name, and session persistence mode.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
sessionPersistenceMode	Specifies the session persistence mode. Valid values include `DATABASE` and `NONE`.
otherAttributeList	Optionally specifies additional attributes in the following format: `[["accessSessionOnTimeout", "true"], ["enabled", "true"]]`

Syntax

```
AdminServerManagement.configureSessionManagerForServer(nodeName, serverName,
sessionPersistenceMode, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureSessionManagerForServer("myNode", "myServer", "DATABASE",
[["accessSessionOnTimeout", "true"], ["enabled", "true"]])
```

configureWebContainer

This script configures web containers in your application server configuration. A web container handles requests for servlets, JavaServer Pages (JSP) files, and other types of files that include server-side code. The web container creates servlet instances, loads and unloads servlets, creates and manages request and response objects, and performs other servlet management tasks.

Table 69. configureWebContainer argument descriptions. Run the script with the node name, server name, default virtual host name, and whether to enable servlet cache.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
webContainerName	Specifies the name of the web container of interest.
defaultVirtualHostName	Specifies a virtual host that enables a single host machine to resemble multiple host machines. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine. Valid values include: **default_host** The product provides a default virtual host with some common aliases such as the machine IP address, short host name, and fully qualified host name. The alias comprises the first part of the path for accessing a resource such as a servlet. For example, it is `localhost:9080` in the request `http://localhost:9080/myServlet`. **admin_host** This is another name for the application server; also known as server1 in the base installation. This process supports the use of the administrative console. **proxy_host** The virtual host called proxy_host, includes default port definitions, port 80 and 443, which are typically initialized as part of the proxy server initialization. Use this proxy host as appropriate with routing rules associated with the proxy server.
enableServletCaching	Specifies that if a servlet is invoked once and it generates output to be cached, a cache entry is created containing not only the output, but also side effects of the invocation. These side effects can include calls to other servlets or JavaServer Pages (JSP) files, as well as metadata about the entry, including timeout and entry priority information. Portlet fragment caching requires that servlet caching is enabled. Therefore, enabling portlet fragment caching automatically enables servlet caching. Disabling servlet caching automatically disables portlet fragment caching.

Table 69. configureWebContainer argument descriptions (continued). Run the script with the node name, server name, default virtual host name, and whether to enable servlet cache.

Argument	Description
otherAttributeList	Optionally specifies additional attributes in the following format: `[["allowAsyncRequestDispatching", "true"], ["disablePooling", "true"], ["sessionAffinityTimeout", 20]]`

Syntax

```
AdminServerManagement.configureWebContainer(nodeName, serverName,
defaultVirtualHostName, enableServletCaching, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureWebContainer("myNode", "myServer", "myVH.uk.kingdom.com",
 true,
   [["allowAsyncRequestDispatching", "true"], ["disablePooling", "true"], ["sessionAffinityTimeout", 20]])
```

configureJavaProcessLogs

This script configures Java process logs for the application server. The system creates the JVM logs by redirecting the System.out and System.err streams of the JVM to independent log files.

Table 70. configureJavaProcessLogs argument descriptions. Run the script with the Java process definition of interest and root directory for the process logs.

Argument	Description
javaProcessDefConfigID	Specifies the configuration ID of the Java Process Definition of interest.
processLogRoot	Specifies the root directory for the process logs.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["stdinFilename", "/temp/mystdin.log"]]`

Syntax

```
AdminServerManagement.configureJavaProcessLogs(javaProcessDefConfigID, processLogRoot,
 otherAttributeList)
```

Example usage

```
AdminServerManagement.configureJavaProcessLogs
   ("(cells/WAS00Network/nodes/ndnode1/servers/server1|server.xml#JavaProcessDef_1184194176408)",
    "/temp/myJavaLog", [["stdinFilename", "/temp/mystdin.log"]])
```

configurePerformanceMonitoringService

This script configures performance monitoring infrastructure (PMI) in your configuration. PMI enables the server to collect performance data from various product components. PMI provides information about average system resource usage statistics, with no correlation between the data across different components.

Table 71. configurePerformanceMonitoringService argument descriptions. Run the script with the node name, server name, whether to enable PMI, and the initial specification level arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
enable	Specifies whether the application server attempts to enable Performance Monitoring Infrastructure (PMI). If an application server is started when the PMI is disabled, you have to restart the server in order to enable it.

Table 71. configurePerformanceMonitoringService argument descriptions (continued). Run the script with the node name, server name, whether to enable PMI, and the initial specification level arguments.

Argument	Description
initialSpecLevel	Specifies a pre-defined set of Performance Monitoring Infrastructure (PMI) statistics for all components in the server. **None** All statistics are disabled. **Basic** Provides basic monitoring for application server resources and applications. This includes Java Platform Enterprise Edition (Java EE) components, HTTP session information, CPU usage information, and the top 38 statistics. This is the default setting. **Extended** Provides extended monitoring, including the basic level of monitoring plus workload monitor, performance advisor, and Tivoli resource models. Extended provides key statistics from frequently used WebSphere Application Server components. **All** Enables all statistics. **Custom** Provides fine-grained control with the ability to enable and disable individual statistics.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["statisticSet", "test statistic set"], ["synchronizedUpdate", "true"]]`

Syntax

```
AdminServerManagement.configurePerformanceMonitoringService(nodeName, serverName,
enable, initialSpecLevel, otherAttributeList)
```

Example usage

```
AdminServerManagement.configurePerformanceMonitoringService("myNode", "myServer", "true", "Basic",
[["statisticSet", "test statistic set"], ["synchronizedUpdate", "true"]])
```

configurePMIRequestMetrics

This script configures PMI request metrics in your configuration. Request metrics provide data about each transaction, correlating this information across the various product components to provide an end-to-end picture of the transaction.

Table 72. configurePMIRequestMetrics argument descriptions. Run the script and specify whether to enable request metrics and the trace level.

Argument	Description
enable	Specifies whether to turn on the request metrics feature. When disabled, the request metrics function is disabled.
traceLevel	Specifies how much trace data to accumulate for a given transaction. Note that trace level and components to be instrumented work together to control whether or not a request will be instrumented. **NONE** No instrumentation. **HOPS** Generates instrumentation information on process boundaries only (for example, a servlet request coming from a browser or a web server and a JDBC request going to a database). **PERFORMANCE_DEBUG** Generates the data at Hops level and the first level of the intra-process servlet and Enterprise JavaBeans (EJB) call (for example, when an inbound servlet forwards to a servlet and an inbound EJB calls another EJB). Other intra-process calls like naming and service integration bus (SIB) are not enabled at this level. **DEBUG** Provides detailed instrumentation data, including response times for all intra-process calls. Requests to servlet filters will only be instrumented at this level.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["armType", "TIVOLI_ARM"], ["enableARM", "true"]]`

Syntax

```
AdminServerManagement.configurePMIRequestMetrics(enable, traceLevel, otherAttributeList)
```

Example usage

```
AdminServerManagement.configurePMIRequestMetrics("true", "DEBUG",
   [["armType", "TIVOLI_ARM"], ["enableARM", "true"]])
```

configureServerLogs

This script configures server logs for the application server of interest.

Table 73. configureServerLogs argument descriptions. Run the script with the node name, server name, and root directory for the server logs.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
serverLogRoot	Specifies the root directory for the server logs.
otherAttributeList	Optionally specifies additional attributes using the following name and value pair format: `[["formatWrites", "true"], ["messageFormatKind", "BASIC"], ["rolloverType", "BOTH"]]`

Syntax

```
AdminServerManagement.configureServerLogs(nodeName, serverName,
serverLogRoot, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureServerLogs("myNode", "myServer", "/temp/mylog",
   [["formatWrites", "true"], ["messageFormatKind", "BASIC"], ["rolloverType", "BOTH"]])
```

configureTraceService

This script configures trace settings for the application server. Configure trace to obtain detailed information about running the application server.

Table 74. configureTraceService argument descriptions. Run the script with the node name, server name, trace specification, and output type arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
startupTraceSpecification	Specifies the trace specification to enable for the component of interest. For example, the `com.ibm.ws.webservices.trace.MessageTrace=all` trace specification traces the contents of a SOAP message, including the binary attachment data.
traceOutputType	Specifies where trace output should be written. The trace output can be written directly to an output file, or stored in memory.
otherAttributeList	Optionally specifies additional attributes for the trace service using the following name and value pair format: `[["enable", "true"], ["traceFormat", "LOG_ANALYZER"]]`

Syntax

```
AdminServerManagement.configureTraceService(nodeName, serverName,
traceString, outputType, otherAttributeList)
```

Example usage

```
AdminServerManagement.configureTraceService("myNode", "myServer", "com.ibm.ws.management.*=all=enabled",
   "SPECIFIED_FILE", [["enable", "true"], ["traceFormat", "LOG_ANALYZER"]])
```

Server configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the application server scripts to create application servers, web servers, and generic servers. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/scriptLibraries/servers/V80 directory. If you do not want to set an argument, specify an empty string as the value for the argument, as the following syntax demonstrates: "".

Use the following script procedures to administer your application server:

- "createApplicationServer"
- "createAppServerTemplate"
- "createGenericServer" on page 110
- "createWebServer" on page 110
- "deleteServer" on page 111
- "deleteServerTemplate" on page 111

createApplicationServer

This script creates a new application server in your environment. During the installation process, the product creates a default application server, named `server1`. Most installations require several application servers to handle the application serving needs of their production environment.

Table 75. createApplicationServer argument descriptions. Run the script with the node, server, and template names.

Argument	Description
nodeName	Specifies the name of the node on which to create the application server.
serverName	Specifies the name of the server to create.
templateName	Optionally specifies the template to use to create the application server.

Syntax

```
AdminServerManagement.createApplicationServer(nodeName, serverName, templateName)
```

Example usage

```
AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
```

createAppServerTemplate

This script creates a new application server template in your configuration. A server template is used to define the configuration settings for a new application server. When you create a new application server, you either select the default server template or a template you previously created, that is based on another, already existing application server. The default template is used if you do not specify a different template when you create the server.

Table 76. createAppServerTemplate argument descriptions. Run the script with the node name, server name, and new template name arguments.

Argument	Description
nodeName	Specifies the node that corresponds to the server from which to base the template.
serverName	Specifies the name of the server from which to base the template.
newTemplateName	Specifies the name of the new template to create.

Syntax

```
AdminServerManagement.createAppServerTemplate(nodeName, serverName, newTemplateName)
```

Example usage

```
AdminServerManagement.createAppServerTemplate("myNode", "myServer", "myNewTemplate")
```

createGenericServer

This script configures a new generic server in the configuration. A generic server is a server that the application server manages, but does not supply. If you do not want to set an argument, specify an empty string as the value for the argument, as the following syntax demonstrates: "".

Table 77. createGenericServer argument description. Run the script with the node name, new server name, template name, start command path and arguments, working directory, and stop command path and arguments.

Argument	Description
nodeName	Specifies the name of the node on which to create the server.
newServerName	Specifies the name of the server to create.
templateName	Optionally specifies the template to use to create the server.
startCmdPath	Optionally specifies the path to the command that will run when this generic server is started.
startCmdArguments	Optionally specifies the arguments to pass to the startCommand when the generic server is started.
workingDirectory	Optionally specifies the working directory for the generic server.
stopCmdPath	Optionally specifies the path to the command that will run when this generic server is stopped.
stopCmdArguments	Optionally specifies the arguments to pass to the stopCommand parameter when the generic server is stopped.

Syntax

```
AdminServerManagement.createGenericServer(nodeName, newServerName, templateName,
startCmdPath, startCmdArguments, workingDirectory, stopCmdPath,
 stopCmdArguments)
```

Example usage

```
AdminServerManagement.createGenericServer("myNode", "myServer",
 "default", "", "", "/temp", "", "")
```

createWebServer

This script configures a web server in your configuration. An application server works with a web server to handle requests for dynamic content, such as servlets, from web applications. A web server uses Web Server Plug-ins to establish and maintain persistent HTTP and HTTPS connections with an application server. If you do not want to set an argument, specify an empty string as the value for the argument, as the following syntax demonstrates: "".

Table 78. createWebServer argument descriptions. Run the script with the node name, new server name, port number, server install root, plug-in installation root, configuration file path, Windows operating system service name, error log path, access log path, and web protocol type.

Argument	Description
nodeName	Specifies the name of the node on which the web server is defined.
newServerName	Specifies the name of the web server to create.
port	Optionally specifies the port from which to ping the status of the web server.
serverInstallRoot	Optionally specifies the fully qualified path where the web server is installed. This field is required if you are using IBM HTTP Server. For all other web servers, this field is not required. If you enable any administrative function for non-IBM HTTP Server web servers, the installation path is necessary.
pluginInstallPath	Specifies the installation path for the Web server plug-in.
configFilePath	Specifies the configuration file for your Web server. Specify the file and not just the directory of the file. The application server generates the plugin-cfg.xml file by default. The configuration file identifies applications, application servers, clusters, and HTTP ports for the web server. The web server uses the file to access deployed applications on various application servers.
errorLogPath	Specifies the location of the error log file.

Table 78. createWebServer argument descriptions (continued). Run the script with the node name, new server name, port number, server install root, plug-in installation root, configuration file path, Windows operating system service name, error log path, access log path, and web protocol type.

Argument	Description
accessLogPath	Specifies the location of the access log file.
webProtocol	Specifies the protocol to use for web server communications. Use the `HTTPS` protocol to securely communicate with the web server. The default is `HTTP`.

Syntax

```
AdminServerManagement.createWebServer(nodeName, newServerName, port,
 serverInstallRoot, pluginInstallPath, configFilePath,
 errorLogPath,
  accessLogPath, webProtocol)
```

Example usage

```
AdminServerManagement.createWebServer("myNode", "myWebServer", "", "", "", "", "", "", "", "" )
```

deleteServer

This script removes a server from the application server configuration.

Table 79. deleteServer argument descriptions. Run the script with the node and server names.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server to delete.

Syntax

```
AdminServerManagement.deleteServer(nodeName, serverName)
```

Example usage

```
AdminServerManagement.deleteServer("myNode", "myServer")
```

deleteServerTemplate

This script deletes a server template from your configuration.

Table 80. deleteServerTemplate argument description. Run the script with the template name.

Argument	Description
templateName	Specifies the name of the template to delete.

Syntax

```
AdminServerManagement.deleteServerTemplate(templateName)
```

Example usage

```
AdminServerManagement.deleteServerTemplate("newServerTemplate")
```

Server query scripts

The scripting library provides multiple script procedures to automate your server configurations. This topic provides usage information for scripts that query your application server configuration. You can run each script individually, or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/`scriptLibraries/servers/V70` directory. Use the following script procedures to query your application server configuration:

- “checkIfServerExists” on page 112
- “checkIfServerTemplateExists ” on page 112

checkIfServerExists

This script determines whether the server of interest exists in your configuration. To run the script, specify the node name and server name arguments, as defined in the following table:

Table 81. checkIfServerExists argument descriptions. Run the script to see if a server exists.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.checkIfServerExists(nodeName, serverName)
```

Example usage

```
AdminServerManagement.checkIfServerExists("myNode", "myServer")
```

checkIfServerTemplateExists

This script determines whether the server template of interest exists in your configuration. To run the script, specify the template name arguments, as defined in the following table:

Table 82. checkIfServerTemplateExists argument descriptions. Run the script to see if a template exists.

Argument	Description
templateName	Specifies the name of the server template of interest.

Syntax

```
AdminServerManagement.checkIfServerTemplateExists(templateName)
```

Example usage

```
AdminServerManagement.checkIfServerTemplateExists("newServer")
```

getJavaHome

This script displays the Java home value. To run the script, specify the node name and server name arguments, as defined in the following table:

Table 83. getJavaHome argument descriptions. Run the script to see the Java home value.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.getJavaHome(nodeName, serverName)
```

Example usage

```
AdminServerManagement.getJavaHome("myNode", "myServer")
```

getServerProcessType

This script displays the type of server process for a specific server. To run the script, specify the node and server name arguments for the server of interest, as defined in the following table:

Table 84. getServerProcessType argument descriptions. Run the script to see the type of server process.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.getServerProcessType(nodeName, serverName)
```

Example usage

```
AdminServerManagement.getServerProcessType("myNode", "server1")
```

getServerPID

This script displays the running server process ID for a specific target. To run the script, specify the node and server name arguments for the server of interest, as defined in the following table:

Table 85. getServerPID argument descriptions. Run the script to see a running server process ID.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.getServerPID(nodeName, serverName)
```

Example usage

```
AdminServerManagement.getServerPID("myNode", "server1")
```

help

This script displays the script procedures that the AdminServerManagement script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Table 86. help argument description. Run the script to see help.

Argument	Description
scriptName	Specifies the name of the script of interest.

Syntax

```
AdminServerManagement.help(scriptName)
```

Example usage

```
AdminServerManagement.help("getServerProcessType")
```

listJVMProperties

This script displays the properties that are associated with your Java virtual machine (JVM) configuration. To run the script, specify the node name, server name, and optionally the JVM property of interest, as defined in the following table:

Table 87. listJVMProperties argument descriptions. Run the script to see JVM properties.

Argument	Description
nodeName	Optionally specifies the name of the node of interest.
serverName	Optionally specifies the name of the server of interest.
JVMProperty	Optionally specifies the JVM property to query.

Syntax

```
AdminServerManagement.listJVMProperties(nodeName, serverName, JVMProperty)
```

Example usage

```
AdminServerManagement.listJVMProperties("myNode", "myServer", "")
```

listServers

This script displays the servers that exist in your configuration. You can optionally specify the node name or server type to query for a specific scope, as defined in the following table:

Table 88. listServers argument descriptions. Run the script to see what servers exist.

Argument	Description
serverType	Specifies the name of the server to query.
nodeName	Specifies the name of the node to query.

Syntax

```
AdminServerManagement.listServers(serverType, nodeName)
```

Example usage

```
AdminServerManagement.listServers("APPLICATION_SERVER", "myNode")
```

listServerTemplates

This script displays the server templates in your configuration. To run the script, specify the template version, server type, and template name, as defined in the following table:

Table 89. listServerTemplates argument descriptions. Run the script to see what templates exists.

Argument	Description
templateVersion	Optionally specifies the version of the template of interest.
serverType	Optionally specifies the type of server. Valid values include the `GENERIC_SERVER`, `WEB_SERVER`, `APPLICATION_SERVER` , and `PROXY_SERVER` server types.
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminServerManagement.listServerTemplates(templateVersion, serverType, templateName)
```

Example usage

```
AdminServerManagement.listServerTemplates("", "APPLICATION_SERVER", "default")
```

listServerTypes

This script displays the server types that are available on the node of interest. To run the script, specify the node name, as defined in the following table:

Table 90. listServerTypes argument descriptions. Run the script to see the server types.

Argument	Description
nodeName	Optionally specifies the name of the node of interest.

Syntax

```
AdminServerManagement.listServerTypes(nodeName)
```

Example usage

```
AdminServerManagement.listServerTypes("myNode")
```

queryMBeans

This script queries the application server for Managed Beans (MBeans). Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.

Table 91. queryMBeans argument descriptions. Run the script with the node name and server name arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mbeanType	Specifies the type of MBean to query.

Syntax

```
AdminServerManagement.queryMBeans(nodeName, serverName, mbeanType)
```

Example usage

```
AdminServerManagement.queryMBeans("myNode", "server1", "Server")
```

showServerInfo

This script displays server configuration properties for the server of interest. The script displays the cell name, server type, product version, node name, and server name.

Table 92. showServerInfo argument descriptions. Run the script with the node name and server name arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.showServerInfo(nodeName, serverName)
```

Example usage

```
AdminServerManagement.showServerInfo("myNode", "myServer")
```

viewProductInformation

This script displays the application server product version.

Syntax

```
AdminServerManagement.viewProductInformation()
```

Example usage

```
AdminServerManagement.viewProductInformation()
```

Server administration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the application server scripts to delete, start, and stop servers. You can run each script individually or combine procedures to create custom automation scripts for your environment.

All server management script procedures are located in the *app_server_root*/scriptLibraries/servers/V8 directory.

Use the following script procedures to administer your application server:

- "startAllServers"
- "startSingleServer"
- "stopAllServers" on page 117
- "stopSingleServer" on page 117

startAllServers

This script starts all servers on a node in your configuration.

Table 93. startAllServers argument description. Run the script with the node name.

Argument	Description
nodeName	Specifies the name of the node of interest.

Syntax

```
AdminServerManagement.startAllServers(nodeName)
```

Example usage

```
AdminServerManagement.startAllServers("myNode")
```

startSingleServer

This script starts a specific server in your configuration.

Table 94. startSingleServer argument descriptions. Run the script with the node name and server name.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server to start.

Syntax

```
AdminServerManagement.startSingleServer(nodeName, serverName)
```

Example usage

```
AdminServerManagement.startSingleServer("myNode", "myServer")
```

stopAllServers

This script stops all servers on a node in your configuration.

Table 95. stopAllServers argument description. Run the script with the node name.

Argument	Description
nodeName	Specifies the name of the node of interest.

Syntax

```
AdminServerManagement.stopAllServers(nodeName)
```

Example usage

```
AdminServerManagement.stopAllServers("myNode")
```

stopSingleServer

This script stops a single server in your configuration.

Table 96. stopSingleServer argument descriptions. Run the script with the node name and server name.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminServerManagement.stopSingleServer(nodeName, serverName, classname, displayname, classpath, otherAttributeList)
```

Example usage

```
AdminServerManagement.stopSingleServer("myNode", "myServer")
```

Automating administrative architecture setup using wsadmin scripting library

The scripting library provides Jython script procedures to assist in automating your environment. Use the server, node, and cluster management scripts to configure servers, nodes, node groups, and clusters in your application server environment.

Before you begin

Before you can complete this task, you must install an application server in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")
  ```

```
# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
    "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

Use the scripts in the following directories to configure your administrative architecture:

- The server and cluster management procedures are located in the *app_server_root*/scriptLibraries/servers/V70 subdirectory.
- The node and node group management procedures are located in the *app_server_root*/scriptLibraries/system/V70 subdirectory.

Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory, and save existing automation scripts in the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

This topic provides one sample combination of procedures. Use the following steps to create a node group and add three nodes to the group:

Procedure

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, job manager, or administrative agent profile, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode using the Jython scripting language:

```
wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Display the nodes in your environment.

   Run the listNodes script procedure from the AdminNodeManagement script library, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminNodeManagement.listNodes()"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminNodeManagement.listNodes()
```

For this example, the command returns the following output:

```
Node1
Node2
Node3
```

3. Create a node group.

   Run the createNodeGroup script procedure from the AdminNodeGroupManagement script library, specifying the name to assign to the new node group, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminNodeGroupManagement.createNodeGroup("NodeGroup1")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminNodeGroupManagement.createNodeGroup("myNodeGroup")
```

4. Add nodes to the node group.

   Run the addNodeGroupMember script procedure from the AdminNodeGroupManagement script library to add the `Node1`, `Node2`, and `Node3` nodes to the `NodeGroup1` node group, specifying the node name and node group name, as the following examples demonstrate:

```
wsadmin -lang jython -c "AdminNodeGroupManagement.addNodeGroupMember("Node1", "NodeGroup1")"
wsadmin -lang jython -c "AdminNodeGroupManagement.addNodeGroupMember("Node2", "NodeGroup1")"
wsadmin -lang jython -c "AdminNodeGroupManagement.addNodeGroupMember("Node3", "NodeGroup1")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminNodeGroupManagement.addNodeGroupMember("Node1", "NodeGroup1")
wsadmin>AdminNodeGroupManagement.addNodeGroupMember("Node2", "NodeGroup1")
wsadmin>AdminNodeGroupManagement.addNodeGroupMember("Node3", "NodeGroup1")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Automating application configurations using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the application management scripts to install, uninstall, export, start, stop, and manage applications in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

```
#
# My Custom Jython Script - file.py
#
AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
    "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The application management procedures in scripting library are located in the *app_server_root*/scriptLibraries/application/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminApplication.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. Use the following steps to use the scripting library to install an application on a cluster and start the application:

Procedure

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Create a cluster.

   Run the createClusterWithoutMember script procedure from the AdminClusterManagement script library, and specify the required arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminClusterManagement.createClusterWithoutMember('myCluster')"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminClusterManagement.createClusterWithoutMember("myCluster")
```

3. Create a cluster member for the new cluster.

   Run the createClusterMember script procedure from the AdminClusterManagement script library, and specify the required arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminClusterManagement.createClusterMember('myCluster', 'myNode', 'myNewMember')"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminClusterManagement.createClusterWithoutMember("myCluster", "myNode", "myNewMember")
```

4. Install the application on the newly created cluster.

   Run the installAppWithClusterOption script procedure from the AdminApplication script library, and specify the required arguments, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminApplication.installAppWithClusterOption('myApplication','myApplicationEar.ear','myCluster')"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminApplication.installAppWithClusterOption("myApplication", "myApplicationEar.ear", "myCluster")
```

5. Start the application on the cluster.

   Run the startApplicationOnCluster script procedure from the AdminApplication script library and specify the required arguments, as the following example displays:

```
bin>wsadmin -lang jython -c "AdminApplication.startApplicationOnCluster('myApplication','myCluster')"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication", "myCluster")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Application installation and uninstallation scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that install applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V80 directory. Use the following script procedures to install and uninstall applications:

installAppWithDefaultBindingOption

This script installs an application using the -usedefaultbindings option.

To run the script, specify the application name, enterprise archive (EAR) file, data source Java Naming and Directory Interface (JNDI) name, data source user name, data source password, connection factory, Enterprise JavaBeans prefix, and virtual host name arguments, as defined in the following table:

Table 97. installAppWithDefaultBindingOption argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the target node.
serverName	Specifies the name of the target server.
dsJndiName	Specifies the JNDI name of the data source to use.
dsUserName	Specifies the user name for the data source.
dsPassword	Specifies the password for the data source.
connFactory	Specifies the name of the connection factory to use.
EJBprefix	Specifies the Enterprise JavaBeans (EJB) prefix to use.
virtualHostName	Specifies the virtual host for the application to install.

Syntax

```
AdminApplication.installAppWithDefaultBindingOption(appName,
 earFile, nodeName, serverName, dsJndiName,
 dsUserName, dsPassword, connFactory, EJBprefix,
 virtualHostName)
```

Example usage

```
AdminApplication.installAppWithDefaultBindingOption("myApp", "\ears\DefaultApplication.ear", "myNode",
 "myServer", "myJndi", "user1", "password", "myCf", "myEjb", "myVH")
```

installAppWithNodeAndServerOptions

This script installs an application using the -node and -server options.

To run the script, specify the application name, EAR file, node name, and server name arguments, as defined in the following table:

Table 98. installAppWithNodeAndServerOptions argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.

Syntax

```
AdminApplication.installAppWithNodeAndServerOptions(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppWithNodeAndServerOptions("myApp", "/ears/DefaultApplication.ear",
 "myNode", "myServer")
```

installAppWithClusterOption

This script installs an application using the -cluster option.

To run the script, specify the application name, EAR file, and cluster name arguments, as defined in the following table:

Table 99. installAppWithClusterOption argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
clusterName	Specifies the name of the cluster of interest.

Syntax

```
AdminApplication.installAppWithClusterOption(appName,
 earFile, clusterName)
```

Example usage

```
AdminApplication.installAppWithClusterOption("myApp", "/ears/DefaultApplication.ear",
 "myCluster")
```

installAppModulesToSameServerWithMapModulesToServersOption

This script deploys application modules to the same server using the -MapModulesToServers option.

To run the script, specify the application name, EAR file, node name, and server name arguments, as defined in the following table:

Table 100. installAppModulesToSameServerWithMapModulesToServersOption argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.

Syntax

```
AdminApplication.installAppModulesToSameServerWithMapModulesToServersOption(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppModulesToSameServerWithMapModulesToServersOption("myApp",
 "/ears/DefaultApplication.ear", "myNode", "myServer")
```

installAppModulesToDiffServersWithMapModulesToServersOption

This script deploys application modules to different servers using the -MapModulesToServers option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and both server name arguments, as defined in the following table:

Table 101. installAppModulesToDiffServersWithMapModulesToServersOption argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppModulesToDiffServersWithMapModulesToServersOption(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppModulesToDiffServersWithMapModulesToServersOption("myApp",
 "/ears/DefaultApplication.ear", "myCell", "myNode", "myServer1", "myServer2")
```

installAppModulesToSameServerWithPatternMatching

This script deploys application modules with the -MapModulesToServers pattern matching option.

To run the script, specify the application name, EAR file, node name, and server name arguments, as defined in the following table:

Table 102. installAppModulesToSameServerWithPatternMatching argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.

Syntax

```
AdminApplication.installAppModulesToSameServerWithPatternMatching(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppModulesToSameServerUingPatternMatching("myApp",
 "/ears/DefaultApplication.ear", "myNode", "myServer")
```

installAppModulesToDiffServersWithPatternMatching

This script deploys application modules to different servers using the -MapModulesToServers pattern matching option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and both server name arguments, as defined in the following table:

Table 103. installAppModulesToDiffServersWithPatternMatching argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.

Table 103. installAppModulesToDiffServersWithPatternMatching argument descriptions (continued). Run the script with argument values.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppModulesToDiffServersWithPatternMatching(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppModulesToDiffServersWithPatternMatching("myApp", "/ears/DefaultApplication.ear", "myNode", "myServer1", "myServer2")
```

installAppModulesToMultiServersWithPatternMatching

This script deploys application modules to multiple servers using the -MapModulesToServers pattern matching option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the AdminApplication.py file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and each server name arguments, as defined in the following table:

Table 104. installAppModulesToMultiServersWithPatternMatching argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppModulesToMultiServersWithPatternMatching(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppModulesToMultiServersWithPatternMatching("myApp",
 "/ears/DefaultApplication.ear", "myCell", "myNode", "myServer1", "myServer2")
```

installAppWithTargetOption

This script deploys an application to multiple servers using the -target option. Use this script to install application modules to one or two servers. To install to additional servers, create a custom script based on the syntax in the `AdminApplication.py` file, or run the script multiple times.

To run the script, specify the application name, EAR file, node name, and each server name arguments, as defined in the following table:

Table 105. installAppWithTargetOption argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.

Table 105. installAppWithTargetOption argument descriptions (continued). Run the script with argument values.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName1	Specifies the name of the application server to which the application is deployed.
serverName2	Specifies the name of an additional application server to which the application is deployed.

Syntax

```
AdminApplication.installAppWithTargetOption(appName,
 earFile, nodeName, serverName1,
 serverName2)
```

Example usage

```
AdminApplication.installAppWithTargetOption("myApp", "/ears/DefaultApplication.ear", "myCell",
 "myNode", "myServer1", "myServer2")
```

installAppWithDeployEjbOptions

This script deploys an application with the -deployejb option.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Table 106. installAppWithDeployEjbOptions argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.
nodeName	Specifies the name of the target node.
serverName	Specifies the name of the target server.

Syntax

```
AdminApplication.installAppWithDeployEjbOptions(appName,
 earFile, nodeName, serverName)
```

Example usage

```
AdminApplication.installAppWithDeployEjbOptions("myApp", "/ears/DefultApplication.ear", "myNode",
 "myServer")
```

installAppWithVariousTasksAndNonTasksOptions

This script deploys an application with various tasks and non-tasks options.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Table 107. installAppWithVariousTasksAndNonTasksOptions argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
earFile	Specifies the EAR file to deploy.

Syntax

```
AdminApplication.installAppWithVariousTasksAndNonTasksOptions(appName,
 earFile)
```

Example usage

```
AdminApplication.installAppWithVariousTasksAndNonTasksOptions("myApp",
 "/ears/DefaultApplication.ear")
```

installWarFile

This script installs a web application archive (WAR) file. A web module is created by assembling servlets, JavaServer Pages (JSP) files, and static content such as Hypertext Markup Language (HTML) pages into a single deployable unit. Web modules are stored in web application archive (WAR) files, which are standard Java archive files.

To run the script, specify the application name, WAR file, node name, server name, and context root arguments, as defined in the following table:

Table 108. installWarFile argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to install.
warFile	Specifies the WAR file to deploy.
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the application server of interest.
contextRoot	Specifies the context root of the web application. The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is http://host:port/gettingstarted/MySession.

Syntax

```
AdminApplication.installWarFile(appName, warFile,
 nodeName, serverName, contextRoot)
```

Example usage

```
AdminApplication.installWarFile("myApp", "/binaries/DefaultWebApplication.war", "myNode",
 "myServer", "/")
```

uninstallApplication

This script uninstalls an application.

To run the script, specify the application name argument, as defined in the following table:

Table 109. uninstallApplication argument descriptions. Run the script with argument values.

Argument	Description
appName	Specifies the name of the application to uninstall.

Syntax

```
AdminApplication.uninstallApplication(appName)
```

Example usage

```
AdminApplication.uninstallApplication("myApp")
```

Application query scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that query your application configuration. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to query application configurations:

- “checkIfAppExists” on page 128
- “getAppDeployedNodes” on page 128
- “getAppDeploymentTarget” on page 128

checkIfAppExists

This script checks if the application is deployed on the application server.

To run the script, specify the application name argument, as defined in the following table:

Table 110. checkIfAppExists argument description. Run the script to see if an application exists.

Argument	Description
appName	Specifies the name of the application of interest.

Syntax

```
AdminApplication.checkIfAppExists(appName)
```

Example usage

```
AdminApplication.checkIfAppExists("myApp")
```

getAppDeployedNodes

This script lists the nodes on which the application of interest is deployed.

To run the script, specify the application name argument, as defined in the following table:

Table 111. getAppDeployedNodes argument description. Run the script to list the nodes to which an application is deployed.

Argument	Description
appName	Specifies the name of the application of interest.

Syntax

```
AdminApplication.getAppDeployedNodes(appName)
```

Example usage

```
AdminApplication.getAppDeployedNodes("myApp")
```

getAppDeploymentTarget

This script displays the application deployment target for the application of interest.

To run the script, specify the application name argument, as defined in the following table:

Table 112. getAppDeploymentTarget argument description. Run the script to see information about a deployment target.

Argument	Description
appName	Specifies the name of the application of interest.

Syntax

```
AdminApplication.getAppDeploymentTarget(appName)
```

Example usage

```
AdminApplication.getAppDeploymentTarget("myApp")
```

getTaskInfoForAnApp

This script displays task information for a specific application Enterprise Archive (EAR) file. The script obtains information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or require an update.

To run the script, specify the EAR file and the task arguments, as defined in the following table:

Table 113. getTaskInfoForAnApp argument descriptions. Run the script to see information about an EAR file.

Argument	Description
earFile	Specifies the name of the EAR file of interest.
taskName	Specifies the name of the task of interest.

Syntax

```
AdminApplication.getTaskInfoForAnApp(appName, taskName)
```

Example usage

```
AdminApplication.getTaskInfoForAnApp("/ears/DefaultApplication.ear", "MapWebModToVH")
```

listApplications

This script lists all deployed applications. The script does not require arguments.

Syntax

```
AdminApplication.listApplications()
```

Example usage

```
AdminApplication.listApplications()
```

listApplicationsWithTarget

This script lists all deployed applications for a specific target.

To run the script, specify the node name and server name arguments, as defined in the following table:

Table 114. listApplicationsWithTarget argument descriptions. Run the script to list deployed applications.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminApplication.listApplicationsWithTarget(nodeName, serverName)
```

Example usage

```
AdminApplication.listApplicationsWithTarget("myNode", "server1")
```

listModulesInAnApp

This script lists each module in a deployed application.

To run the script, specify the application name and server name arguments, as defined in the following table:

Table 115. listModulesInAnApp argument descriptions. Run the script to list modules in a deployed application.

Argument	Description
appName	Specifies the name of the application of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminApplication.listModulesInAnApp(appName, serverName)
```

Example usage

```
AdminApplication.listModulesInAnApp("myApp", "myServer")
```

Application update scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that update applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/`scriptLibraries/application/V70` directory. Use the following script procedures to update application configurations:

- "addSingleFileToAnAppWithUpdateCommand"
- "addSingleModuleFileToAnAppWithUpdateCommand" on page 131
- "addUpdateSingleModuleFileToAnAppWithUpdateCommand" on page 131
- "addPartialAppToAnAppWithUpdateCommand" on page 131
- "deleteSingleFileToAnAppWithUpdateCommand" on page 132
- "deleteSingleModuleFileToAnAppWithUpdateCommand" on page 132
- "deletePartialAppToAnAppWithUpdateCommand" on page 133
- "updateApplicationUsingDefaultMerge" on page 133
- "updateApplicationWithUpdateIgnoreNewOption" on page 133
- "updateApplicationWithUpdateIgnoreOldOption" on page 134
- "updateEntireAppToAnAppWithUpdateCommand " on page 134
- "updatePartialAppToAnAppWithUpdateCommand" on page 134
- "updateSingleFileToAnAppWithUpdateCommand" on page 135
- "updateSingleModuleFileToAnAppWithUpdateCommand" on page 135

addSingleFileToAnAppWithUpdateCommand

This script uses the update command to add a single file to a deployed application.

To run the script, specify the application name, file name, and the content uniform resource identifier (URI) arguments, as defined in the following table:

Table 116. addSingleFileToAnAppWithUpdateCommand argument descriptions. Run the script to add a file to a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.addSingleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.addSingleFileToAnAppWithUpdateCommand("myApp", "/sample.txt", "META-INFO/sample.txt")
```

addSingleModuleFileToAnAppWithUpdateCommand

This script uses the update command to add a single module file to a deployed application.

To run the script, specify the application name, file name, and content URI arguments, as defined in the following table:

Table 117. addSingleModuleFileToAnAppWithUpdateCommand argument descriptions. Run the script to add a module file to a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.addSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.addSingleModuleFileToAnAppWithUpdateCommand("myApp", "/Increment.jar", "Increment.jar")
```

addUpdateSingleModuleFileToAnAppWithUpdateCommand

This script uses the update command to add and update a single module file for a deployed application.

To run the script, specify the application name, file name, content URI, and context root arguments, as defined in the following table:

Table 118. addUpdateSingleModuleFileToAnAppWithUpdateCommand argument descriptions. Run the script to update a file in a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.
contextRoot	Specifies the context root for web modules in the application.

Syntax

```
AdminApplication.addUpdateSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI, contextRoot)
```

Example usage

```
AdminApplication.addUpdateSingleModuleFileToAnAppWithUpdateCommand("myApp",
"/DefaultWebApplication.war", "DefaultWebApplication.war",
"/webapp/defaultapp")
```

addPartialAppToAnAppWithUpdateCommand

This script uses the update command to add a partial application to a deployed application.

To run the script, specify the application name and file content arguments, as defined in the following table:

Table 119. addPartialAppToAnAppWithUpdateCommand argument descriptions. Run the script to update part of a deployed application.

Argument	Description
appName	Specifies the name of the application to update.

Table 119. addPartialAppToAnAppWithUpdateCommand argument descriptions (continued). Run the script to update part of a deployed application.

Argument	Description
fileContent	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.addPartialAppToAnAppWithUpdateCommand(appName, fileContent)
```

Example usage

```
AdminApplication.addPartialAppToAnAppWithUpdateCommand("myApp", "/partialadd.zip")
```

deleteSingleFileToAnAppWithUpdateCommand

This script uses the update command to delete a single file from a deployed application.

To run the script, specify the application name, file name, and content URI arguments, as defined in the following table:

Table 120. deleteSingleFileToAnAppWithUpdateCommand argument descriptions. Run the script to delete a file from a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.deleteSingleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.deleteSingleFileToAnAppWithUpdateCommand("myApp", "/sample.
txt", "META-INFO/sample.txt")
```

deleteSingleModuleFileToAnAppWithUpdateCommand

This script uses the update command to delete a single module file from a deployed application.

To run the script, specify the application name, file name, and content URI arguments, as defined in the following table:

Table 121. deleteSingleModuleFileToAnAppWithUpdateCommand argument descriptions. Run the script to delete a module file from a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.deleteSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.deleteSingleModuleFileToAnAppWithUpdateCommand("myApp",
"/Increment.jar", "Increment.jar")
```

deletePartialAppToAnAppWithUpdateCommand

This script uses the update command to delete a partial application from a deployed application.

To run the script, specify the application name, file name, and the content URI arguments, as defined in the following table:

Table 122. deletePartialAppToAnAppWithUpdateCommand argument descriptions. Run the script to delete part of a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.deletePartialAppToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.deletePartialAppToAnAppWithUpdateCommand("myApp", "/part
ialdelete.zip", "partialdelete")
```

updateApplicationUsingDefaultMerge

This script updates an application using default merging.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Table 123. updateApplicationUsingDefaultMerge argument descriptions. Run the script to update a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateApplicationUsingDefaultMerge(appName, earFile)
```

Example usage

```
AdminApplication.updateApplicationUsingDefaultMerge("myApp", "/ears/D
efaultApplication.ear")
```

updateApplicationWithUpdateIgnoreNewOption

This script updates an application using -update.ignore.new option. The system ignores the bindings from the new version of the application.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Table 124. updateApplicationWithUpdateIgnoreNewOption argument descriptions. Run the script to update a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateApplicationWithUpdateIgnoreNewOption(appName, earFile)
```

Example usage

```
AdminApplication.updateApplicationWithUpdateIgnoreNewOption("myApp",
"c:/ears/DefaultApplication.ear")
```

updateApplicationWithUpdateIgnoreOldOption

This script updates an application using the -update.ignore.old option. The system ignores the bindings from the installed version of the application.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Table 125. updateApplicationWithUpdateIgnoreOldOption argument descriptions. Run the script to update a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateApplicationWithUpdateIgnoreOldOption(appName, earFile)
```

Example usage

```
AdminApplication.updateApplicationWithUpdateIgnoreOldOption("myApp",
"/ears/DefaultApplication.ear")
```

updateEntireAppToAnAppWithUpdateCommand

This script uses the update command to update an entire deployed application.

To run the script, specify the application name and EAR file arguments, as defined in the following table:

Table 126. updateEntireAppToAnAppWithUpdateCommand argument descriptions. Run the script to update a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
earFile	Specifies the name of the file to use to update the application.

Syntax

```
AdminApplication.updateEntireAppToAnAppWithUpdateCommand(appName, earFile)
```

Example usage

```
AdminApplication.updateEntireAppToAnAppWithUpdateCommand("myApp", "/new.ear")
```

updatePartialAppToAnAppWithUpdateCommand

This script uses the update command to update a partial application for a deployed application.

To run the script, specify the application name, file name,and the content URI arguments, as defined in the following table:

Table 127. updatePartialAppToAnAppWithUpdateCommand argument descriptions. Run the script to update part of a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.updatePartialAppToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.updatePartialAppToAnAppWithUpdateCommand("myApp", "/part
ialadd.zip", "partialadd")
```

updateSingleFileToAnAppWithUpdateCommand

This script uses the update command to update a single file on a deployed application.

To run the script, specify the application name, file name, and the content URI arguments, as defined in the following table:

Table 128. updateSingleFileToAnAppWithUpdateCommand argument descriptions. Run the script to update a file in a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.updateSingleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.updateSingleFileToAnAppWithUpdateCommand("myApp", "/sample.
txt", "META-INFO/sample.txt")
```

updateSingleModuleFileToAnAppWithUpdateCommand

This script uses the update command to update a single module file for a deployed application.

To run the script, specify the application name, file name, and the content URI arguments, as defined in the following table:

Table 129. updateSingleModuleFileToAnAppWithUpdateCommand argument descriptions. Run the script to update a module file in a deployed application.

Argument	Description
appName	Specifies the name of the application to update.
fileContent	Specifies the name of the file to use to update the application.
contentURI	Specifies the URI of the file content.

Syntax

```
AdminApplication.updateSingleModuleFileToAnAppWithUpdateCommand(appName, fileContent, contentURI)
```

Example usage

```
AdminApplication.updateSingleModuleFileToAnAppWithUpdateCommand("myApp",
"/sample.jar", "Increment.jar")
```

Application export scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that export applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to export applications:

- “exportAnAppToFile”
- “exportAllApplicationsToDir”
- “exportAnAppDDLToDir”

exportAnAppToFile

This script exports a deployed application to a specific file.

To run the script, specify the application name and export file name arguments, as defined in the following table:

Table 130. exportAnAppToFile argument descriptions. Run the script to export a deployed application.

Argument	Description
appName	Specifies the name of the application of interest.
exportFileName	Specifies the name of the file to which the system exports the application.

Syntax

```
AdminApplication.exportAnAppToFile(appName, exportFileName)
```

Example usage

```
AdminApplication.exportAnAppToFile("myApp", "exported.ear")
```

exportAllApplicationsToDir

This script exports all deployed applications to a specific directory.

To run the script, specify the application name and export file name arguments, as defined in the following table:

Table 131. exportAllApplicationsToDir argument description. Run the script to export all deployed applications.

Argument	Description
exportDirectory	Specifies the fully qualified directory path to which the system exports each application.

Syntax

```
AdminApplication.exportAllApplicationsToDir(exportDirectory)
```

Example usage

```
AdminApplication.exportAllApplicationsToDir("/export")
```

exportAnAppDDLToDir

This script exports the data definition language (DDL) from the application to a specific directory.

To run the script, specify the application name, export directory, and options arguments, as defined in the following table:

Table 132. exportAnAppDDLToDir argument descriptions. Run the script to export a DDL.

Argument	Description
appName	Specifies the name of the application to export.
exportDirectory	Specifies the fully qualified directory path to which the system exports each application.
options	Optionally specifies additional export options.

Syntax

```
AdminApplication.exportAnAppDDLToDir(appName, exportFileName, options)
```

Example usage

```
AdminApplication.exportAnAppDDLToDir("myApp", "/export", "")
```

Application deployment configuration scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that deploy applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. The application deployment script procedures contain multiple arguments. If you do not want to specify an argument with the script, specify the value of the argument as an empty string, as the following syntax demonstrates: "".

Use the following script procedures to deploy applications:

- “configureStartingWeightForAnApplication”
- “configureClassLoaderPolicyForAnApplication”
- “configureClassLoaderLoadingModeForAnApplication” on page 138
- “configureSessionManagementForAnApplication” on page 138
- “configureApplicationLoading” on page 139
- “configureLibraryReferenceForAnApplication” on page 140
- “configureEJBModulesOfAnApplication” on page 140
- “configureWebModulesOfAnApplication” on page 140
- “configureConnectorModulesOfAnApplication” on page 141

configureStartingWeightForAnApplication

This script configures the starting weight attribute for an application.

To run the script, specify the application name and starting weight arguments, as defined in the following table:

Table 133. configureStartingWeightForAnApplication argument descriptions. Run the script to set an application starting weight.

Argument	Description
appName	Specifies the name of the application to configure.
startingWeight	Specifies the starting weight to set for the application of interest.

Syntax

```
AdminApplication.configureStartingWeightForAnApplication(appName,
 startingWeight)
```

Example usage

```
AdminApplication.configureStartingWeightForAnApplication("myApp",
 "10")
```

configureClassLoaderPolicyForAnApplication

This script configures the class loader policy attribute for an application.

To run the script, specify the application name argument, as defined in the following table:

Table 134. configureClassLoaderPolicyForAnApplication argument descriptions. Run the script to set an application class loader policy.

Argument	Description
appName	Specifies the name of the application to configure.
classloaderPolicy	Specifies the class loader policy for the application of interest. For each application server in the system, you can set the application class-loader policy to `SINGLE` or `MULTIPLE`. When the application class-loader policy is set to `SINGLE`, then a single application class loader loads all EJB modules, dependency JAR files, and shared libraries in the system. When the application class-loader policy is set to `MULTIPLE`, then each application receives its own class loader that is used for loading the EJB modules, dependency JAR files, and shared libraries for that application.

Syntax

```
AdminApplication.configureClassLoaderPolicyForAnApplication(appName,
 classloaderPolicy)
```

Example usage

```
AdminApplication.configureClassLoaderPolicyForAnApplication("myApp",
 "SINGLE")
```

configureClassLoaderLoadingModeForAnApplication

This script configures the class loader loading mode for an application. The class-loader delegation mode, also known as the class loader order, determines whether a class loader delegates the loading of classes to the parent class loader.

To run the script, specify the application name argument, as defined in the following table:

Table 135. configureClassLoaderLoadingModeForAnApplication argument descriptions. Run the script to set an application class loader mode.

Argument	Description
appName	Specifies the name of the application to configure.
classloaderMode	Specifies the class loader mode to set for the application of interest. You can set the class loader mode to `PARENT_FIRST` or `PARENT_LAST`. The `PARENT_FIRST` class-loader mode causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. This value is the default for the class-loader policy and for standard JVM class loaders. The `PARENT_LAST` class-loader mode causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.

Syntax

```
AdminApplication.configureClassLoaderLoadingModeForAnApplication(appName,
 classloaderMode)
```

Example usage

```
AdminApplication.configureClassLoaderLoadingModeForAnApplication("myApp",
 "PARENT_LAST")
```

configureSessionManagementForAnApplication

This script configures session management for an application.

To run the script, specify the application name argument, as defined in the following table:

Table 136. configureSessionManagementForAnApplication argument descriptions. Run the script to configure application session management.

Argument	Description
appName	Specifies the name of the application to configure.
enableCookie	Specifies whether to enable cookies.
enableProtocolSwitching	Specifies whether session tracking uses cookies to carry session IDs. If cookies are enabled, session tracking recognizes session IDs that arrive as cookies and tries to use cookies for sending session IDs. If cookies are not enabled, session tracking uses Uniform Resource Identifier (URL) rewriting instead of cookies (if URL rewriting is enabled).
enableURLRewriting	Specifies whether the session management facility uses rewritten URLs to carry the session IDs. If URL rewriting is enabled, the session management facility recognizes session IDs that arrive in the URL if the encodeURL method is called in the servlet.
enableSSLTracking	**Note:** This feature is deprecated in WebSphere Application Server Version 7.0. You can reconfigure session tracking to use cookies or modify the application to use URL rewriting. If you do not want to specify this argument, specify the value as an empty string, as the following syntax demonstrates: `""`. Specifies that session tracking uses Secure Sockets Layer (SSL) information as a session ID. Enabling SSL tracking takes precedence over cookie-based session tracking and URL rewriting.
enableSerializedSession	Specifies whether to allow concurrent session access in a given server.
accessSessionOnTimeout	Specifies whether the servlet is started normally or aborted in the event of a timeout. If you specify `true`, the servlet is started normally. If you specify `false`, the servlet execution aborts and error logs are generated.
maxWaitTime	Specifies the maximum amount of time a servlet request waits on an HTTP session before continuing execution. This parameter is optional and expressed in seconds. The default is 5 seconds. Under normal conditions, a servlet request waiting for access to an HTTP session gets notified by the request that currently owns the given HTTP session when the request finishes.
sessionPersistMode	Specifies whether to enable session persistence mode.
allowOverflow	Specifies whether the number of sessions in memory can exceed the value specified by the Max in-memory session count property. This option is valid only in non-distributed sessions mode.
maxInMemorySessionCount	Specifies the maximum number of sessions to maintain in memory.
invalidTimeout	Specifies the amount of time, in minutes, before a timeout occurs that is not valid.
sessionEnable	Specifies whether to enable session.

Syntax

```
AdminApplication.configureSessionManagementForAnApplication(appName,
 enableCookie, enableProtocolSwitching, enableURLRewriting,
 enableSSLTracking, enableSerializedSession, accessSessionOnTimeout,
 maxWaitTime, sessionPersistMode, allowOverflow,
 maxInMemorySessionCount, invalidTimeout, sessionEnable)
```

Example usage

```
AdminApplication.configureSessionManagementForAnApplication("myApplication",
 "false", "false", "true", "", "true", "90",  "NONE", "true", "1500",
 "40", "true")
```

configureApplicationLoading

This script configures the application loading attribute for an application.

To run the script, specify the application name argument, as defined in the following table:

Table 137. configureApplicationLoading argument descriptions. Run the script to configure application loading.

Argument	Description
appName	Specifies the name of the application to configure.
enableTargetMapping	Specifies whether to enable target mapping during application loading.

Syntax

```
AdminApplication.configureApplicationLoading(appName,
 enableTargetMapping)
```

Example usage

```
AdminApplication.configureApplicationLoading("myApp", "true")
```

configureLibraryReferenceForAnApplication

This script configures the library reference for an application.

To run the script, specify the application name and shared library name arguments, as defined in the following table:

Table 138. configureLibraryReferenceForAnApplication argument descriptions. Run the script to configure application library reference.

Argument	Description
appName	Specifies the name of the application to configure.
libraryName	Specifies the name of the shared library to configure.

Syntax

```
AdminApplication.configureLibraryReferenceForAnApplication(appName,
 libraryName)
```

Example usage

```
AdminApplication.configureLibraryReferenceForAnApplication("myApp",
 "sharedLibaray")
```

configureEJBModulesOfAnApplication

This script configures the EJB modules of an application.

To run the script, specify the application name argument, as defined in the following table:

Table 139. configureEJBModulesOfAnApplication argument descriptions. Run the script to configure EJB modules of an application.

Argument	Description
appName	Specifies the name of the application to configure.
startingWeight	Specifies the target weight of the EJB modules in the application of interest.
enableTargetMapping	Specifies whether to enable target mapping for EJB modules.

Syntax

```
AdminApplication.configureEJBModulesOfAnApplication(appName,
 startingWeight, enableTargetMapping)
```

Example usage

```
AdminApplication.configureEJBModulesOfAnApplication("myApp", "1500",
 "true")
```

configureWebModulesOfAnApplication

This script configures the web modules of an application.

To run the script, specify the application name argument, as defined in the following table:

Table 140. configureWebModulesOfAnApplication argument descriptions. Run the script to configure Web modules of an application.

Argument	Description
appName	Specifies the name of the application of interest.
webModuleName	Specifies the name of the web module to configure.
startingWeight	Specifies the starting weight for the web module of interest.
classloaderMode	Specifies the class loader mode to set for the application of interest. You can set the class loader mode to `PARENT_FIRST` or `PARENT_LAST`. The `PARENT_FIRST` class-loader mode causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. This value is the default for the class-loader policy and for standard JVM class loaders. The `PARENT_LAST` class-loader mode causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader.

Syntax

```
AdminApplication.configureWebModulesOfAnApplication(appName,
 webModuleName, startingWeight, classloaderMode)
```

Example usage

```
AdminApplication.configureWebModulesOfAnApplication("myApp", "myWebModule",
 "250", "PARENT_FIRST")
```

configureConnectorModulesOfAnApplication

This script configures the connector modules of an application. To run the script, specify the application name, J2C connection factory, and node name arguments.

To run the script, specify the application name argument, as defined in the following table:

Table 141. configureConnectorModulesOfAnApplication argument descriptions. Run the script to configure connector modules of an application.

Argument	Description
appName	Specifies the name of the application of interest.
j2cConnFactory	Specifies the name of the Java 2 Connector (J2C) connection factory to configure.
jndiName	Specifies the name of the Java Naming and Directory Interface (JNDI) of interest.
authDataAlias	Specifies the name of the authentication data alias of interest.
connectionTimeout	Specifies the number of seconds that a connection request waits when there are no connections available in the free pool and no new connections can be created. This usually occurs because the maximum value of connections in the particular connection pool has been reached.

Syntax

```
AdminApplication.configureConnectorModulesOfAnApplication(appName,
 j2cConnFactory, jndiName, authDataAlias,
 connectionTimeout)
```

Example usage

```
AdminApplication.configureConnectorModulesOfAnApplication("myApp",
 "myConnFactory", "myDefaultSSLSettings", "150")
```

Application administration scripts

The scripting library provides multiple script procedures to automate your application configurations. This topic provides usage information for scripts that start and stop applications. You can run each script individually or combine procedures to create custom automation scripts for your environment.

Each application management script procedure is located in the *app_server_root*/scriptLibraries/application/V70 directory. Use the following script procedures to start and stop applications:

- "startApplicationOnSingleServer"
- "startApplicationOnAllDeployedTargets"
- "startApplicationOnCluster"
- "stopApplicationOnSingleServer " on page 143
- "stopApplicationOnAllDeployedTargets" on page 143
- "stopApplicationOnCluster using wsadmin scripting" on page 143

startApplicationOnSingleServer

This script starts an application on a single server.

To run the script, specify the application name, node name, and server name arguments, as defined in the following table:

Table 142. startApplicationOnSingleServer argument descriptions. Run the script to start an application on a server.

Argument	Description
appName	Specifies the name of the application to start.
nodeName	Specifies the name of the node on which the application is deployed.
serverName	Specifies the name of the application server on which the application is deployed.

Syntax

```
AdminApplication.startApplicationOnSingleServer(appName, nodeName, serverName)
```

Example usage

```
AdminApplication.startApplicationOnSingleServer("myApp", "myNode", "myServer")
```

startApplicationOnAllDeployedTargets

This script starts an application on all deployed nodes.

To run the script, specify the application name and node name arguments, as defined in the following table:

Table 143. startApplicationOnAllDeployedTargets argument descriptions. Run the script to start an application on all deployable targets.

Argument	Description
appName	Specifies the name of the application to start.
nodeName	Specifies the name of the node on which the application is deployed.

Syntax

```
AdminApplication.startApplicationOnAllDeployedTargets(appName, nodeName)
```

Example usage

```
AdminApplication.startApplicationOnAllDeployedTargets("myApp", "myNode")
```

startApplicationOnCluster

This script starts an application on a cluster.

To run the script, specify the application name and cluster name arguments, as defined in the following table:

Table 144. startApplicationOnCluster argument descriptions. Run the script to start an application on a cluster.

Argument	Description
appName	Specifies the name of the application to start.
clusterName	Specifies the name of the cluster on which the application is deployed.

Syntax

```
AdminApplication.startApplicationOnCluster(appName, clusterName)
```

Example usage

```
AdminApplication.startApplicationOnCluster("myApp", "myCluster")
```

stopApplicationOnSingleServer

This script stops an application on a single server.

To run the script, specify the application name, node name, and server name arguments, as defined in the following table:

Table 145. stopApplicationOnSingleServer argument descriptions. Run the script to stop an application on a server.

Argument	Description
appName	Specifies the name of the application to stop.
nodeName	Specifies the name of the node on which the application is deployed.
serverName	Specifies the name of the application server on which the application is deployed.

Syntax

```
AdminApplication.stopApplicationOnSingleServer(appName, nodeName, serverName)
```

Example usage

```
AdminApplication.stopApplicationOnSingleServer("myApp", "myNode", "myServer")
```

stopApplicationOnAllDeployedTargets

This script stops an application on all deployed nodes.

To run the script, specify the application name, cell name, and node name arguments, as defined in the following table:

Table 146. stopApplicationOnAllDeployedTargets argument descriptions. Run the script to stop an application on all deployment targets.

Argument	Description
appName	Specifies the name of the application to stop.
nodeName	Specifies the name of the node on which the application is deployed.

Syntax

```
AdminApplication.stopApplicationOnAllDeployedTargets(appName, nodeName)
```

Example usage

```
AdminApplication.stopApplicationOnAllDeployedTargets("myApp", "myNode")
```

stopApplicationOnCluster using wsadmin scripting

This script stops an application on a cluster.

To run the script, specify the application name and cluster name arguments, as defined in the following table:

Table 147. stopApplicationOnCluster argument descriptions. Run the script to stop an application on a cluster.

Argument	Description
appName	Specifies the name of the application to stop.
clusterName	Specifies the name of the cluster on which the application is deployed.

Syntax

```
AdminApplication.stopApplicationOnCluster(appName, clusterName)
```

Example usage

```
AdminApplication.stopApplicationOnCluster("myApp", "myCluster")
```

Automating business-level application configurations using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the application management scripts to install, uninstall, export, start, stop, and manage business-level applications in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
      "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication",
      "..\installableApps\DefaultApplication.ear", "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The business-level application procedures in scripting library are located in the *app_server_root*/scriptLibraries/application/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminBLA.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. See the business-level application configuration scripts documentation to view argument descriptions and syntax examples.

Use this topic and the scripting library to create an empty business-level application, add assets as composition units, and start the business-level application.

Procedure

1. Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Import assets to your configuration.

   Assets represent application binaries that contain business logic that runs on the target run time environment and serves client requests. An asset can contain a file, an archive of files such as a ZIP or Java archive (JAR) file, or an archive of archive files such as a Java Platform, Enterprise Edition (Java EE) EAR file. Other examples of assets include Enterprise JavaBeans (EJB) JAR files, EAR files, Service Component Architecture (SCA) composite JAR files, OSGi bundles, mediation JAR files, shared library JAR files, and non-Java EE contents such as PHP applications.

   Run the importAsset script from the AdminBLA script library to import assets to the application server configuration repository, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminBLA.importAsset("asset.zip", "true", "true")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminBLA.importAsset("asset.zip", "true", "true")
```

3. Create an empty business-level application.

   Run the createEmptyBLA script from the AdminBLA script library to create a new business-level application, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminBLA.createEmptyBLA("myBLA", "bla to control transactions")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminBLA.createEmptyBLA("myBLA", "bla to control transactions")
```

4. Add the assets, as composition units, to the business-level application.

   Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-WebSphere Application Server runtime environments without backing assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

Run the addCompUnit script from the AdminBLA script library to add asset.zip to myBLA as a composition unit, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminBLA.addCompUnit("myBLA", "asset.zip", "default",
 "myCompositionUnit", "cu description", "1", "server1", "specname=actplan1")"
```

You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminBLA.addCompUnit("myBLA", "asset.zip", "default", "myCompositionUnit",
 "cu description", "1", "server1", "specname=actplan1")
```

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

6. Start the business-level application.

   Use the startBLA script from the AdminBLA script library to start each composition unit of the business-level application on the deployment targets for which the composition units are configured, as the following example demonstrates:

```
wsadmin>AdminBLA.startBLA("myBLA")
```

Results

The business-level application is configured and started on the deployment target of interest.

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Use the business-level application configuration scripts to create custom scripts to automate your environment. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Business-level application configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to create, query, and manage your business-level applications. You can run each script individually or combine procedures to create custom automation scripts.

The AdminBLA script procedures are located in the *app_server_root*/scriptLibraries/application/V70 directory.

Use the following script procedures to configure and administer your business-level applications:

Use the following script procedures to query your business-level application configurations:

gotcha: The commands, **viewBLA**, **viewAsset**, and **viewCompUnit** only display output to the console and do not return data to the calling Jython script. These commands are not intended to be used as part of a script to store the output of the command to a string variable. By using these commands in a Jython script to store the output to a string variable, the string variable will only contain the value "Operation Successful!".

When invoked, as intended, with the Jython wsadmin interface, these commands properly display output to the console.

addCompUnit

This script adds assets, shared libraries, or additional business-level applications as composition units to the empty business-level application. A composition unit represents an asset in a business-level application. A configuration unit enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents.

To run the script, specify the business-level application name and the composition unit source arguments, as defined in the following table:

Table 148. addCompUnit argument descriptions. Run the script to add a composition unit to a business-level application.

Argument	Description
blaName	Specifies the name of the business-level application to which the system adds the composition unit.
compUnitID	Specifies the name of the composition unit to add to the business-level application of interest.
deployableUnit	Optionally specifies the name of the deployable unit for the asset. A deployable unit is the smallest portion of an asset that can be individually chosen for deployment
compUnitName	Optionally specifies the name for the composition unit to add.
compUnitDescription	Optionally specifies a description for the new composition unit.
startingWeight	Optionally specifies the starting weight of the composition unit.
target	Optionally specifies the target to which the composition unit is mapped.
activationPlan	Optionally specifies the activation plan for the composition unit.

Syntax

```
AdminBLA.addCompUnit(blaName, compUnitID, deployableUnit, compUnitName,
 compUnitDescription, startingWeight, target, activationPlan)
```

Example usage

```
AdminBLA.addCompUnit("bla1", "asset1.zip", "default", "myCompositionUnit", "cu description", "1",
"server1", "specname=actplan1")
```

createEmptyBLA

This script creates a new business-level application in your environment. Create an empty business-level application and then add assets, shared libraries, or business-level applications as composition units to the empty business-level application.

To run the script, specify the business-level application name argument, as defined in the following table:

Table 149. createEmptyBLA argument descriptions. Run the script to create a business-level application.

Argument	Description
blaName	Specifies the name to assign to the new business-level application.
description	Optionally specifies a description for the business-level application.

Syntax

```
AdminBLA.createEmptyBLA(blaName, description)
```

Example usage

```
AdminBLA.createEmptyBLA("myBLA", "bla to control transactions")
```

deleteAsset

This script removes a registered asset from your configuration.

To run the script, specify the asset ID argument, as defined in the following table:

Table 150. deleteAsset argument description. Run the script to delete an asset.

Argument	Description
assetID	Specifies the name of the asset to delete.

Syntax

```
AdminBLA.deleteAsset(assetID)
```

Example usage

```
AdminBLA.deleteAsset("asset.zip")
```

deleteBLA

This script removes a business-level application from your configuration.

To run the script, specify the business-level application name argument, as defined in the following table:

Table 151. deleteBLA argument description. Run the script to delete a business-level application.

Argument	Description
blaName	Specifies the name of the business-level application to delete.

Syntax

```
AdminBLA.deleteBLA(blaName)
```

Example usage

```
AdminBLA.deleteBLA("myBLA")
```

deleteCompUnit

This script removes a composition unit from a specific business-level application configuration.

To run the script, specify the business-level application name and composition unit arguments, as defined in the following table:

Table 152. deleteCompUnit argument descriptions. Run the script to delete a composition unit from a business-level application.

Argument	Description
blaName	Specifies the name of the business-level application of interest.
compUnitID	Specifies the identifier of the composition unit to delete.

Syntax

```
AdminBLA.deleteCompUnit(blaName, compUnitID)
```

Example usage

```
AdminBLA.deleteCompUnit("myBLA", "asset.zip")
```

editAsset

This script edits the metadata of a specific registered asset.

To run the script, specify the arguments that are defined in the following table:

Table 153. editAsset argument descriptions. Run the script to change an asset.

Argument	Description
assetID	Specifies the name of the asset to edit.
assetDescription	Optionally specifies the new description of the asset of interest.
assetDestinationURL	Optionally specifies the new destination URL for the asset of interest.
assetTypeAspects	Optionally specifies the new type aspects for the asset of interest.
assetRelationships	Optionally specifies the new asset relationship configurations.
filePermission	Optionally specifies the new file permission configuration for the asset of interest.
validateAsset	Optionally specifies whether the command validates the asset.

Syntax

```
AdminBLA.editAsset(assetID, assetDescription, assetDestinationURL,
assetTypeAspects, assetRelationships, filePermission, validateAsset)
```

Example usage

```
AdminBLA.editAsset("asset1.zip", "asset for testing", "c:/installedAssets/asset1.zip",
"WebSphere:spec=sharedlib", "", ".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755", "true")
```

editCompUnit

This script edits a specific composition unit within a business-level application.

To run the script, specify the business-level application name and composition unit ID arguments, as defined in the following table:

Table 154. editCompUnit argument descriptions. Run the script to change a composition unit.

Argument	Description
blaName	Specifies the name of the business-level application to which the composition unit is associated.
compUnitID	Specifies the name of the composition unit to edit.
compUnitDescription	Optionally specifies a new description for the composition unit.
startingWeight	Optionally specifies a new starting weight for the composition unit.
target	Optionally specifies a new target to which the composition unit is mapped.
activationPlan	Optionally specifies a new activation plan for the composition unit.

Syntax

```
AdminBLA.editCompUnit(blaName, compUnitID, compUnitDescription,
startingWeight, target, activationPlan)
```

Example usage

```
AdminBLA.editCompUnit("bla1", "asset1.zip","cu description", "1",
"server1", "specname=actplan1")
```

exportAsset

This script exports a registered asset to a file on your system.

To run the script, specify the asset ID and file name arguments, as defined in the following table:

Table 155. exportAsset argument descriptions. Run the script to export an asset.

Argument	Description
assetID	Specifies the identifier of the asset to export.
fileName	Specifies the fully qualified file path to which the system exports the asset.

Syntax

```
AdminBLA.exportAsset(assetID, fileName)
```

Example usage

```
AdminBLA.exportAsset("asset.zip", "/temp/a.zip")
```

importAsset

This script imports and registers an asset to a management domain in your configuration.

To run the script, specify the assetID, displayDescription, and deployableUnit arguments, as defined in the following table:

Table 156. importAsset argument descriptions. Run the script to import an asset.

Argument	Description
assetID	Specifies the asset to import.
displayDescription	Optionally specifies whether the script displays the description of the asset.
dispDeployableUnit	Optionally specifies whether the script displays the deployable units for the asset to import.

Syntax

```
AdminBLA.importAsset(assetID, displayDescription, dispDeployableUnit)
```

Example usage

```
AdminBLA.importAsset("asset.zip", "true", "true")
```

startBLA

This script starts the business-level application process in your configuration.

To run the script, specify business-level application name argument, as defined in the following table:

Table 157. startBLA argument description. Run the script to start a business-level application.

Argument	Description
blaName	Specifies the name of the business-level application to start.

Syntax

```
AdminBLA.startBLA(blaName)
```

Example usage

```
AdminBLA.startBLA("myBLA")
```

stopBLA

This script stops the business-level application process in your configuration.

To run the script, specify the business-level application name argument, as defined in the following table:

Table 158. stopBLA argument description. Run the script to stop a business-level application.

Argument	Description
blaName	Specifies the name of the business-level application to stop.

Syntax

```
AdminBLA.stopBLA(blaName)
```

Example usage

```
AdminBLA.stopBLA("myBLA")
```

help

This script displays the script procedures that the AdminBLA script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Table 159. help argument description. Run the script to display help.

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminBLA.help(script)
```

Example usage

```
AdminBLA.help("createEmptyBLA")
```

listAssets

This script displays the registered assets in your configuration.

To run the script, you can choose to specify the asset ID, display description, and display deployable units arguments, as defined in the following table:

Table 160. listAssets argument descriptions. Run the script to list assets.

Argument	Description
assetID	Optionally specifies the group ID for which to display authorization groups.
displayDescription	Optionally specifies whether the command displays a description for each asset. Specify `true` to display descriptions.
displayDeployUnits	Optionally specifies whether the command displays the deployable units that are associated with the assets. Specify `true` to display the deployable units.

Syntax

```
AdminBLA.listAssets(assetID, displayDescription, displayDeployUnits)
```

Example usage

```
AdminBLA.listAssets("asset.zip", "true", "true")
```

listBLAs

This script displays each or specific business-level applications in your configuration.

To run the script, you can choose to specify the business-level application name and the display description arguments, as defined in the following table:

Table 161. listBLAs argument descriptions. Run the script to list business-level applications.

Argument	Description
blaName	Optionally specifies the name of a business-level application of interest.
displayDescription	Optionally specifies whether the command displays a description for each business-level application. Specify `true` to display descriptions.

Syntax

```
AdminBLA.listBLAs(blaName, displayDescription)
```

Example usage

```
AdminBLA.listBLAs("", "true")
```

listCompUnits

This script displays composition units within a business-level application.

To run the script, specify the business-level application name argument, as defined in the following table:

Table 162. listCompUnits argument descriptions. Run the script to list composition units.

Argument	Description
blaName	Specifies the name of the authorization group of interest.
displayDescription	Optionally specifies whether the command displays a description for each composition unit. Specify `true` to display descriptions.

Syntax

```
AdminBLA.listCompUnits(blaName, displayDescription)
```

Example usage

```
AdminBLA.listCompUnits("myBLA", "true")
```

viewBLA

This script displays the name and description of the business-level application of interest.

To run the script, specify the configuration ID argument of the business-level application of interest as defined in the following table:

Table 163. viewBLA argument description. Run the script to view information about a business-level application.

Argument	Description
blaID	Specifies the configuration ID of the business-level asset of interest.

Syntax

```
AdminBLA.viewBLA(blaID)
```

Example usage

```
AdminBLA.viewBLA("bla01.zip")
```

viewAsset

This script displays the configuration attributes for a specific registered asset.

To run the script, specify the asset ID argument, as defined in the following table:

Table 164. viewAsset argument description. Run the script to view information about an asset.

Argument	Description
assetID	Specifies the name of the asset of interest.

Syntax

```
AdminBLA.viewAsset(assetID)
```

Example usage

```
AdminBLA.viewAsset("asset.zip")
```

viewCompUnit

This script displays the configuration attributes for a specific composition unit within a business-level application.

To run the script, specify the business-level application and composition unit ID arguments, as defined in the following table:

Table 165. viewCompUnit argument descriptions. Run the script to view information about a composition unit.

Argument	Description
blaName	Specifies the name of the business-level application of interest.
compUnitID	Specifies the identifier for the composition unit of interest.

Syntax

```
AdminBLA.viewCompUnit(blaName, compUnitID)
```

Example usage

```
AdminBLA.viewCompUnit("myBLA", "asset.zip")
```

Automating data access resource configuration using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the resource management scripts to configure and manage your Java Database Connectivity (JDBC) configurations.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  ```

```
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
    "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The data access resource management procedures in the scripting library are located in the *app_server_root*/scriptLibraries/resources/JDBC/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, save your automation scripts to a new subdirectory in the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the scripts to perform many combinations of administration functions. Use the following sample combination of procedures to configure a JDBC provider and data source.

Procedure

1. Verify that all of the necessary JDBC driver files are installed on your node manager. If you opt to configure a user-defined JDBC provider, check your database documentation for information about the driver files.
2. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the `bin` directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the `bin` directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.
3. Configure a JDBC provider.

   Run the createJDBCProvider procedure from the script library and specify the required arguments. To run the script, specify the node name, server name, name to assign to the new JDBC provider, and the implementation class name. You can optionally specify additional attributes in the following format:

[["attr1", "value1"], ["attr2", "value2"]]. Custom properties for specific vendor JDBC drivers must be set on the application server data source. Consult your database documentation for information about available custom properties.

The following example creates a JDBC provider in your configuration:

```
bin>wsadmin -lang jython -c "AdminJDBC.createJDBCProvider("myNode",
 "myServer", "myJDBCProvider", "myImplementationClass", [["description", "testing"],
 ["xa", "false"], ["providerType", "provType"]])"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJDBC.createJDBCProvider("myNode", "myServer",
 "myJDBCProvider", "myImplementationClass",    [["description", "testing"], ["xa",
 "false"], ["providerType", "provType"]])
```

The script returns the configuration ID of the new JDBC provider.

4. Use a template to configure a data source.

   Run the createDataSourceUsingTemplate procedure from the script library and specify the required arguments. To run the script, specify the node name, server name, JDBC provider name, configuration ID of the template to use, and the name to assign to the new data source. You can optionally specify additional attributes in the following format: [["attr1", "value1"], ["attr2", "value2"]].

   The following example uses a template to create a data source in your configuration:

```
bin>wsadmin -lang jython -c "AdminJDBC.createDataSourceUsingTemplate("myNode",
 "myServer", "myJDBCProvider",    "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)",     "myDataSource",
 [["authDataAlias", "myalias"], ["authMechanismPreference", "BASIC_PASSWORD"],
 ["description", "testing"], ["jndiName", "dsjndi1"], ["logMissingTransactionContext",
 "true"],    ["statementCacheSize", "5"]])"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJDBC.createDataSourceUsingTemplate("myNode", "myServer",
 "myJDBCProvider",    "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)",    "myDataSource",
 [["authDataAlias", "myalias"], ["authMechanismPreference", "BASIC_PASSWORD"],
 ["description", "testing"], ["jndiName", "dsjndi1"], ["logMissingTransactionContext",
 "true"], ["statementCacheSize", "5"]])
```

The script returns the configuration ID of the new data source.

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of true if the cluster exists, or false if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify true as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

J2C query scripts

The scripting library provides many script procedures to manage your Java 2 Connector (J2C) configurations. This topic provides usage information for scripts that query your J2C configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each J2C management script procedure is located in the *app_server_root*/scriptLibraries/resources/J2C directory.

Use the following script procedures to query your J2C configurations:

- "listAdminObjectInterfaces"
- "listConnectionFactoryInterfaces"
- "listJ2CActivationSpecs"
- "listJ2CAdminObjects" on page 157
- "listJ2CConnectionFactories" on page 157
- "listJ2CResourceAdapters" on page 157
- "listMessageListenerTypes" on page 158

listAdminObjectInterfaces

This script returns and displays a list of the administrative object interfaces for the J2C resource adapter of interest.

To run the script, specify the J2C resource adapter argument, as defined in the following table:

Table 166. listAdminObjectInterfaces script. Run the script to list administrative object interfaces.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.

Syntax

```
AdminJ2C.listAdminObjectInterfaces(resourceAdapterID)
```

Example usage

```
AdminJ2C.listAdminObjectInterfaces("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)")
```

listConnectionFactoryInterfaces

This script returns and displays a list of the connection factory interfaces for the J2C resource adapter of interest.

To run the script, specify the J2C resource adapter argument, as defined in the following table:

Table 167. listConnectioinFactoryInterfaces script. Run the script to list connection factory interfaces.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.

Syntax

```
AdminJ2C.listConnectionFactoryInterfaces(resourceAdapterID)
```

Example usage

```
AdminJ2C.listConnectionFactoryInterfaces("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)")
```

listJ2CActivationSpecs

This script returns and displays a list of the J2C activation specifications in your J2C configuration.

To run the script, specify the J2C resource adapter and message listener type arguments, as defined in the following table:

Table 168. listJ2CActivationSpecs script. Run the script to list activation specification interfaces.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
messageListenerType	Specifies the message listener type.

Syntax

```
AdminJ2C.listJ2CActivationSpecs(resourceAdapterID, messageListenerType)
```

Example usage

```
AdminJ2C.listJ2CActivationSpecs("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
  "javax.jms.MessageListener2")
```

listJ2CAdminObjects

This script returns and displays a list of the administrative objects in your J2C configuration.

To run the script, specify the application name and server name arguments, as defined in the following table:

Table 169. listJ2CAdminObjects script. Run the script to list administrative object interfaces in a J2C configuration.

Argument	Description
resourceAdapterID	Specifies the name of the application of interest.
adminObjectInterface	Specifies the name of the administrative object interface of interest.

Syntax

```
AdminJ2C.listJ2CAdminObjects(resourceAdapterID, adminObjectInterface)
```

Example usage

```
AdminJ2C.listJ2CAdminObjects("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
  "fvt.adapter.message.FVTMessageProvider2")
```

listJ2CConnectionFactories

This script returns and displays a list of the J2C connection factories in your J2C configuration.

To run the script, specify the J2C resource adapter and connection factory interface arguments, as defined in the following table:

Table 170. listJ2CConnectionFactories script. Run the script to list J2C connection factories.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
connFactoryInterface	Specifies the name of the connection factory interface of interest.

Syntax

```
AdminJ2C.listJ2CConnectionFactories(resourceAdapterID, connFactoryInterface)
```

Example usage

```
AdminJ2C.listJ2CConnectionFactories("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
  "javax.sql.DataSource2")
```

listJ2CResourceAdapters

This script displays a list of the J2C resource adapters in your configuration. The script returns either a list of J2CResourceAdapters with the resource adapter name or a list of all J2C resource adapters in the environment.

To run the script, you can optionally specify the J2C resource adapter argument, as defined in the following table:

Table 171. listJ2CResourceAdapters script. Run the script to list J2C resource adapters.

Argument	Description
resourceAdapterName	Specifies the name of the resource adapter to display.

Syntax

```
AdminJ2C.listJ2CResourceAdapters(resourceAdapterName)
```

Example usage

```
AdminJ2C.listJ2CResourceAdapters()
AdminJ2C.listJ2CResourceAdapters("myResourceAdapter")
```

listMessageListenerTypes

This script returns and displays a list of the message listener types for the J2C resource adapter of interest.

To run the script, specify the J2C resource adapter argument, as defined in the following table:

Table 172. listMessageListenerTypes script. Run the script to list message listener types.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.

Syntax

```
AdminJ2C.listMessageListenerTypes(resourceAdapterID)
```

Example usage

```
AdminJ2C.listMessageListenerTypes("J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)")
```

J2C configuration scripts

The scripting library provides many script procedures to manage your Java 2 Connector (J2C) configurations. Use the scripts in this topic to create activation specifications, administrative objects, and connection factories, and to install resource adapters. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each J2C management script procedure is located in the *app_server_root*/scriptLibraries/resources/J2C directory.

Use the following script procedures to configure J2C in your environment:

- “createJ2CActivationSpec”
- “createJ2CAdminObject” on page 159
- “createJ2CConnectionFactory” on page 160
- “installJ2CResourceAdapter” on page 162

createJ2CActivationSpec

This script creates a J2C activation specification in your configuration. The script returns the configuration ID of the new J2C activation specification.

To run the script, specify the resource adapter, activation specification name, message listener type, and the Java Naming and Directory Interface (JNDI) name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 173. createJ2CActivationSpec script. Run the script to create a J2C activation specification.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
activationSpecName	Specifies the name to assign to the new activation specification.
messageListenerType	Specifies the message listener type.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 174. Optional attributes. Other attributes available for the script.

Attributes	Description	Example
activationSpec	Specifies the activation specification bean that a resource adapter provides.	`["activationSpec",[["activationSpecClass", "com.my.class"], ["requiredConfigProperties",[["name", "myName"],["type", "myType"], ["value", "myvalue"]]]]]`
authenticationAlias	Specifies the authentication alias of the created J2C activation specification.	`["authenticationAlias", "myAlias"]`
description	Specifies the description for the created J2C activation specification.	`["description", "My description"]`
destinationJndiName	Specifies the destination JNDI name of the created J2C activation.	`["destinationJndiName", "myDestinationJndi"]`
jndiName	Specifies the JNDI name of the created J2C activation.	`["jndiName", "myJndiName"]`
resourceProperties	Specifies additional properties that the activation specification supports.	`["resourceProperties",[["description", "My Description"], ["name", "myName"],["required", "false"],["type", "String"], ["value", "myValue"]]]`

Syntax

```
AdminJ2C.createJ2CActivationSpec(resourceAdapterID,
 activationSpecName, messageListenerType, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJ2C.createJ2CActivationSpec(
"J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578))", "J2CASTest", "javax.jms.MessageListener2", "jndiAS")
```

The following example script includes optional attributes in a string format:

```
AdminJ2C.createJ2CActivationSpec(
"J2CTest(cells/AMYLIN4Cell01/nodes/AMYLIN4CellManager03|resources.xml#J2CResourceAdapter_1245171531343)",
"J2CASTest","javax.jms.MessageListener", "jndi/as1",
"description=this is my J2C ActivationSpecification,
destinationJndiName=jndi/J2CAS, authenticationAlias=J2CASTest")
```

The following example script includes optional attributes in a list format:

```
AdminJ2C.createJ2CActivationSpec(
"myj2c(cells/AMYLIN4Cell01/nodes/AMYLIN4CellManager03|resources.xml#J2CResourceAdapter_1238380711218)",
"J2CAC1", "javax.jms.MessageListener", "jndi/as", [['description','new j2cActivationSpec'],
['destinationJndiName','ds/jndi'], ['authenticationAlias','test']])
```

createJ2CAdminObject

This script creates a J2C administrative object in your configuration. The script returns the configuration ID of the new J2C administrative object.

To run the script, specify the resource adapter, activation specification name, Java Naming and Directory Interface (JNDI) name, and the administrative object interface name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 175. createJ2CAdminObject script. Run the script to create a J2C administrative object.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
activationSpecName	Specifies the name to assign to the new activation specification.
adminObjectInterface	Specifies the name of the administrative object interface.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 176. Optional attributes. Other attributes available for the script.

Attributes	Description	Example
adminObject	Specifies the administrative object bean that the resource adapter provides.	`["adminObject",[["adminObjectClass", "com.my.class"], ["adminObjectInterface","com.my.interface"],["configProperties", [["description","My Description"],["descriptions", [["lang", "myLanguage"],["value", "myValue"]]], ["name", "myName"],["type", "myType"],["value", "myValue"]]]]]`
description	Specifies the description for the created J2C administrative object.	`["description", "My description"]`
jndiName	Specifies the name of the Java Naming and Directory Interface (JNDI).	`["jndiName", "myJndiName"]`
properties	Specifies additional properties that the administrative object supports.	`["properties",[["description", "My Description"],["name", "myName"], ["required", "false"],["type", "String"],["value", "myValue"]]]`

Syntax

```
AdminJ2C.createJ2CAdminObject(resourceAdapterID,
 activationSpecName, adminObjectInterface, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJ2C.createJ2CAdminObject(
"J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_1184091767578)",
 "J2CAOTest", "fvt.adapter.message.FVTMessageProvider2", "jndiAO")
```

The following example script includes optional attributes in a string format:

```
AdminJ2C.createJ2CAdminObject(
"J2CTest(cells/AMYLIN4Cell01/nodes/AMYLIN4CellManager03|resources.xml#J2CResourceAdapter_1245171531343)",
"J2CAOTest", "fvt.adapter.message.FVTMessageProvider", "jndi/ao1",
"description=this is my J2C AdminObject")
```

The following example script includes optional attributes in a list format:

```
AdminJ2C.createJ2CAdminObject(
"myj2c(cells/AMYLIN4Cell01/nodes/AMYLIN4CellManager03|resources.xml#J2CResourceAdapter_1238380711218)",
"J2CAO1", "fvt.adapter.message.FVTMessageProvider", "jndi/ao",
[['description','new j2cAdminObject']])
```

createJ2CConnectionFactory

This script creates a new J2C connection factory in your configuration. The script returns the configuration ID of the new J2C connection factory.

To run the script, To run the script, specify the resource adapter, connection factory name, the connection factory interface, and the Java Naming and Directory Interface (JNDI) name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 177. createJ2CConnectionFactory script. Run the script to create a J2C connection factory.

Argument	Description
resourceAdapterID	Specifies the configuration ID of the resource adapter of interest.
connFactoryName	Specifies the name to assign to the new connection factory.
connFactoryInterface	Specifies the connection factory interface.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 178. Optional attributes. Other attributes available for the script.

Attributes	Description	Example
authDataAlias	Specifies the component-managed authentication data alias of the created connection factory.	`["authDataAlias", "myAuthDataAlias"]`
authMechanismPreference	Specifies the authentication mechanism. Use one of the following values: `BASIC_PASSWORD` or `KERBEROS`.	`["authMechanismPreference", "KERBEROS"]`
category	Specifies a category that classifies or groups the resource.	`["category", "myCategory"]`
connectionPool	Specifies the connection pool settings.	`["connectionPool",[["agedTimeout","100"], ["connectionTimeout","1000"], ["freePoolDistributionTableSize",10], ["maxConnections","12"],["minConnections","5"], ["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name","myName"], ["required","false"],["type","String"], ["validationExpression",""], ["value","myValue"]]],["purgePolicy","EntirePool"], ["reapTime","10000"],["struckThreshold","3"], ["struckTime","10"],["struckTimerTime","10"], ["surgeCreationInterval","10"],["surgeThreshold","10"], ["testConnection","true"],["testConnectionInterval","10"], ["unusedTimeout","10000"]]]`
customProperties	Specifies either a TypedProperty or DescriptiveProperty type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`
description	Specifies the description of the created J2C connection factory.	`["description", "My description"]`
logMissingTransactionContext	Specifies whether to enable or disable missing transaction context logging.	`["logMissingTransactionContext", "true"]`
manageCachedHandles	Specifies whether the data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`["manageCachedHandles", "true"]`
mapping	Maps the configuration login to the indicated authentication alias name.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies pretest connection configuration settings.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a TypedProperty or DescriptiveProperty type.	`['properties',[['description','My description'], ['name','myName'],['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]]`

Table 178. Optional attributes (continued). Other attributes available for the script.

Attributes	Description	Example
propertySet	Specifies resource properties in the following format: `[propertySet [[resourceProperties [[[name1 nameValue1][type1 typeValue1][value1 valueValue1]]... [[namen nameValuen][typen typeValuen][valuen valueValuen]] ]]]]`	`[propertySet [[resourceProperties [ [[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`["provider", "myJMSProvider"]`
providerType	Specifies the JMS provider type that the JMS provider uses.	`["providerType", "myJMSProviderType"]`
xaRecoveryAuthAlias	Specifies the database authentication alias that is used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`["xaRecoveryAuthAlias", "myCellManager01/a1"]`

Syntax

```
AdminJ2C.createJ2CConnectionFactory(resourceAdapterID,
 connFactoryName, connFactoryInterface, jndiName,
 attributes)
```

Example usage

```
AdminJ2C.createJ2CConnectionFactory(
"J2CTest(cells/myCell/nodes/myNode|resources.xml#J2CResourceAdapter_11840917675
578)", "J2CCFTest", "javax.sql.DataSource2", "jndiCF")
```

The following example script includes optional attributes in a string format:

```
AdminJ2C.createJ2CConnectionFactory(
"J2CTest(cells/AMYLIN4Cell01/nodes/AMYLIN4CellManager03|resources.xml#J2CResourceAdapter_1245171531343)",
"J2CCFTest", "javax.sql.DataSource", "jndi/j2ccftest",
"description=this is my J2CConnectionFactory, authDataAlias=J2CTest")
```

The following example script includes optional attributes in a list format:

```
AdminJ2C.createJ2CConnectionFactory(
"myj2c(cells/AMYLIN4Cell01/nodes/AMYLIN4CellManager03|resources.xml#J2CResourceAdapter_1238380711218)",
"J2CCFTest", "javax.sql.DataSource2", "jndi/cf",
[['description', 'new j2ccf'], ['authDataAlias', 'test']])
```

installJ2CResourceAdapter

This script installs a J2C resource adapter in your configuration. The script returns the configuration ID of the new J2C resource adapter.

To run the script, specify the node name, resource adapter archive (RAR) file, and the resource adapter name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 179. installJ2CResourceAdapter script. Run the script to install a J2C resource adapter.

Argument	Description
nodeName	Specifies the name of the node of interest.
rarFile	Specifies the fully qualified file path for the RAR file to install.
resourceAdapterName	Specifies the name to assign to the new resource adapter.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 180. Optional attributes. Other attributes available for the script.

Attributes	Description
rar.desc	Specifies the description of the J2C resource adapter.
rar.archivePath	Specifies the name of the path where the file is extracted. If this path is not specified, then the archive is extracted to the `$CONNECTOR_INSTALL_ ROOT` directory.
rar.classpath	Specifies the additional classpath.
rar.nativePath	Specifies the native path.
rar.threadPoolAlias	Specifies the alias of the thread pool.
rar.propertiesSet	Specifies the property set of the J2C resource adapter.
rar.DeleteSourceRar	Specifies whether to delete the source RAR file.
rar.isolatedClassLoader	Specifies the boolean value of the isolated class loader.
rar.enableHASupport	Specifies the boolean value of the enabled high availability.
rar.HACapability	Specifies the kind of high availability capability.

Syntax

```
AdminJ2C.installJ2CResourceAdapter(nodeName, rarFile,
 resourceAdapterName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJ2C.installJ2CResourceAdapter("myNode", "/temp/jca15cmd.rar", "J2CTest")
```

The following example script includes optional attributes in a string format:

```
dminJ2C.installJ2CResourceAdapter("AMYLIN4CellManager03", "/ears/jca15cmd.rar", "J2CTest", "rar.desc=this is J2C,
rar.archivePath=/temp/test.rar, rar.classpath=/temp, rar.isolatedClassLoader=false, rar.enableHASupport=true,
rar.DeleteSourceRar=false")
```

The following example script includes optional attributes in a list format:

```
AdminJ2C.installJ2CResourceAdapter("AMYLIN4Node09", "/ears/jca15cmd.rar", "j2ctest", [['rar.desc', 'this is J2C'],
['rar.archivePath', '/temp/test.rar'], ['rar.classpath', '/temp'], ['rar.nativePath', ''], ['rar.threadPoolAlias', 'test'],
['rar.isolatedClassLoader', 'false'], ['rar.enableHASupport', 'true'], ['rar.DeleteSourceRar', 'false']])
```

JDBC configuration scripts

The scripting library provides many script procedures to manage Java Database Connectivity (JDBC) configurations in your environment. This topic provides usage information for scripts that configure JDBC settings. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each AdminJDBC script procedure is located in the *app_server_root*/scriptLibraries/resources/JDBC/V70 directory.

Beginning with Version 7, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/`scriptLibraries` directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Fast path: Beginning with Fix Pack 5, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the cell, node, server, or cluster scope. Resource providers include mail providers, URL providers, and resource environment providers. You do not have to write custom scripts to configure resources at a particular scope.

Attention: The example usage scripts and the script syntax are split on multiple lines for printing purposes.

Use the following script procedures to configure JDBC in your environment:

- “createDataSource” on page 165
- “createDataSourceUsingTemplate” on page 168
- “createDataSourceAtScope” on page 171
- “createDataSourceUsingTemplateAtScope” on page 173
- “createJDBCProvider” on page 176
- “createJDBCProviderUsingTemplate” on page 179
- “createJDBCProviderAtScope” on page 182
- “createJDBCProviderUsingTemplateAtScope” on page 184

Format for the scope argument

The scope format applies to the scripts in the script library that have the scope argument.

A cell is optional on node, server, and cluster scopes. A node is required on the server scope.

You can delimit the type by using a comma (,) or a colon (:). You can use lower case for the type (cell=, node=, server=, or cluster=.)

The examples in the following table are split on multiple lines for publishing purposes.

Table 181. Examples of the containment path, configuration ID, and type for a particular scope. The scope can be Cell, Node, Server, or Cluster.

Scope	Containment path	Configuration ID	Type
Cell	/Cell:*myCell*/	*myCell(cells/myCell|cell.xml#Cell_1)*	Cell=*myCell* or cell=*myCell*
Node	/Cell:myCell/Node:*myNode*/ or /Node:*myNode*/	*myNode(cells/myCell/nodes/myNode|node.xml#Node_1)*	Cell=*myCell*, Node=*myNode* or Cell=*myCell*: Node=*myNode* or cell=*myCell*, node=*myNode*
Server	/Cell:*myCell*/Node: *myNode*/ Server:*myServer*/ or /Node:*myNode*/Server: *myServer*/	*myServer(cells/myCell/nodes/myNode/servers/myServer|server.xml#Server_1)*	Cell=*myCell*, Node=*myNode*, Server=*myServer* or Node=*myNode*: Server=*myServer* or cell=*myCell*, Node=*myNode*, Server=*myServer*

Table 181. Examples of the containment path, configuration ID, and type for a particular scope (continued). The scope can be Cell, Node, Server, or Cluster.

Scope	Containment path	Configuration ID	Type
Cluster	/Cell:*myCell*/ ServerCluster: *myCluster*/ or /ServerCluster: *myCluster*/	*myCluster(cells* */myCell/clusters/* *myCluster*| *cluster.xml* *#ServerCluster_1)*	Cell=*myCell*, Cluster=*myCluster* or Cell=*myCell*: Cluster=*myCluster* or cell=*myCell*, Cluster=*myCluster*

createDataSource

This script creates a new data source in your configuration. The script returns the configuration ID of the new data source.

To run the script, specify the node name, server name, JDBC provider, and data source name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 182. createDataSource script. Required and optional arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jdbcProvider	Specifies the name of the JDBC provider of interest.
dsName	Specifies the name to assign to the new data source.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 183. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
authDataAlias	Specifies the alias used for database authentication at run time.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication.	
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JDBC connection pooling properties for the parent JDBC connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["connectionPool",[["agedTimeout","100"],` `["connectionTimeout","1000"],` `["freePoolDistributionTableSize",10],` `["maxConnections","12"],["minConnections","5"],` `["numberOfFreePoolPartitions","3"],` `["numberOfSharedPoolPartitions","6"],` `["numberOfUnsharedPoolPartitions","3"],` `["properties",[["description","My description"],` `["name","myName"],["required","false"],["type","String"],` `["validationExpression",""],["value","myValue"]]],` `["purgePolicy","EntirePool"],["reapTime","10000"],` `["struckThreshold","3"],["struckTime","10"],` `["struckTimerTime","10"],["surgeCreationInterval","10"],` `["surgeThreshold","10"],["testConnection","true"],` `["testConnectionInterval","10"],` `["unusedTimeout","10000"]]]`

Table 183. Optional attributes (continued). Several scripts have these attributes.

Attributes	Description	Example
datasourceHelperClassname	Specifies the name of the DataStoreHelper implementation class that extends the capabilities of the implementation class of the JDBC driver. The extended capabilities allow the JDBC drive to perform functions that are specific to the data.	`com.ibm.websphere.rsadapter.DB2DataStoreHelper` `com.ibm.websphere.rsadapter.DerbyDataStoreHelper` `...`
description	Specifies a description of the data source.	`['description', 'My description']`
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name for this data source.	`['jndiName', 'myJndiName']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'false']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'false']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"],` `["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"],` `"retryInterval", "12343"],"retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"],` `["name", "myName"],"required","false"],` `["type","String"],["validationExpression",""],` `"value','myValue"]]]`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[name databaseName][type string][value mys]]` `[name driverType][type integer][value 4]]` `[name serverName][type string][value localhost]]` `[[name portNumber][type integer][value 50000]] ]]]]`
relationalResourceAdapter	Specifies the relational resource adapter that the data source uses. The available Java 2 Connector (J2C) resource adapter ID of J2CResourceAdapterID can be identified with the `AdminConfig.list ('J2CResourceAdapter')` command.	`[relationalResourceAdapter "WebSphere Relational Resource` `Adapter(cells/pongo/nodes/pongo/servers/server1|` `resources.xml#builtin_rra)"]`
statementCacheSize	Specifies the number of statements that the product can cache for each connection. The product optimizes the processing of prepared statements and callable statements by caching statements that are not used in an active connection. Both statement types improve the performance of transactions between an application and a datastore. Caching the statements makes them more readily available.	`['statementCacheSize', 5]`

Table 183. Optional attributes (continued). Several scripts have these attributes.

Attributes	Description	Example
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`['-xaRecoveryAuthAlias', 'myCellManager01/a1']`

Table 184. Optional attributes, continued. The providerType attribute is also available for the script.

Attributes	Description	Example
providerType	Specifies the JDBC provider type that this JDBC provider uses.	`['providerType', 'DB2 Using IBM JCC Driver']`

Syntax

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)

AdminJDBC.createDataSource(nodeName, serverName,
 jdbcProvider, dsName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createDataSource("myNode", "myServer", "myJDBCProvider",
 "myDataSource")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSource("IBM-F4A849C57A0Node01", "server1", "My JDBC Name2", "MyJDBCDS",
"authDataAlias=cellManager01/myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
datasourceHelperClassname=com.ibm.websphere.rsadapter.DB2DataStoreHelper,
description='My description', diagnoseConnectionUsage=true, jndiName=myJndiName,
logMissingTransactionContext=false, manageCachedHandles=false, providerType='DB2 Using IBM JCC Driver',
xaRecoveryAuthAlias=myCellManager01/xa1")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSource("IBM-F4A849C57A0Node01", "server1", "My JDBC Name2", "MyJDBCDS",
[['authDataAlias', 'cellManager01/myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', 100],
['connectionTimeout', 1000], ['freePoolDistributionTableSize', 10], ['maxConnections', 12], ['minConnections', 5],
['numberOfFreePoolPartitions', 3], ['numberOfSharedPoolPartitions', 6], ['numberOfUnsharedPoolPartitions', 3],
['properties', [[['name', 'name1a'], ['value', 'value1a']], [['name', 'name1b'], ['value', 'value1b']]]],
```

```
['purgePolicy', 'EntirePool'], ['reapTime', 10000], ['stuckThreshold', 3], ['stuckTime', 10], ['stuckTimerTime', 10],
['surgeThreshold', 10], ['testConnection', 'true']]], ['datasourceHelperClassname',
'com.ibm.websphere.rsadapter.DB2DataStoreHelper'],
['description', 'My description'], ['diagnoseConnectionUsage', 'true'], ['jndiName', 'myJndiName'],
['logMissingTransactionContext', 'false'],
['manageCachedHandles', 'false'], ['mapping', [['authDataAlias', 'anAlias'], ['mappingConfigAlias', 'anotherTest']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', 12343], ['retryLimit', 4]]], ['properties',
[[['name', 'name1'], ['value', 'value1']], [['name', 'name2'], ['value', 'value2']]]], ['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'integer'], ['value', 4]], [['name', 'serverName'], ['type', 'String'], ['value', 'localhost']],
[['name', 'portNumber'], ['type', 'integer'], ['value', 50000]]]]]], ['providerType', 'DB2 Using IBM JCC Driver'],
['relationalResourceAdapter', 'SIB JMS Resource Adapter
(cells/IBM-F4A849C57A0Cell01/nodes/IBM-F4A849C57A0Node01/servers/server1|resources.xml#J2CResourceAdapter_1232911649746)'],
['statementCacheSize', 5], ['xaRecoveryAuthAlias', 'myCellManager01/xa1']])
```

createDataSourceUsingTemplate

This script uses a template to create a new data source in your configuration. The script returns the configuration ID of the new data source.

To run the script, specify the node name, server name, JDBC provider, template ID, and data source name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 185. createDataSourceUsingTemplate script. Required and optional arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jdbcProvider	Specifies the name of the JDBC provider of interest.
templateID	Specifies the configuration ID of the template to use to create the data source.
dsName	Specifies the name to assign to the new data source.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 186. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
authDataAlias	Specifies the alias used for database authentication at run time.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication.	
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JDBC connection pooling properties for the parent JDBC connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["connectionPool",[["agedTimeout","100"],` `["connectionTimeout","1000"],` `["freePoolDistributionTableSize",10],` `["maxConnections","12"],["minConnections","5"],` `["numberOfFreePoolPartitions","3"],` `["numberOfSharedPoolPartitions","6"],` `["numberOfUnsharedPoolPartitions","3"],` `["properties",[["description","My description"],` `["name","myName"],["required","false"],["type","String"],` `["validationExpression",""],["value","myValue"]]],` `["purgePolicy","EntirePool"],["reapTime","10000"],` `["struckThreshold","3"],["struckTime","10"],` `["struckTimerTime","10"],["surgeCreationInterval","10"],` `["surgeThreshold","10"],["testConnection","true"],` `["testConnectionInterval","10"],` `["unusedTimeout","10000"]]]`

Table 186. Optional attributes (continued). Several scripts have these attributes.

Attributes	Description	Example
datasourceHelperClassname	Specifies the name of the DataStoreHelper implementation class that extends the capabilities of the implementation class of the JDBC driver. The extended capabilities allow the JDBC drive to perform functions that are specific to the data.	`com.ibm.websphere.rsadapter.DB2DataStoreHelper` `com.ibm.websphere.rsadapter.DerbyDataStoreHelper` `...`
description	Specifies a description of the data source.	`['description', 'My description']`
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name for this data source.	`['jndiName', 'myJndiName']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'false']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'false']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"],` `["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"],` `"retryInterval", "12343"],"retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"],` `["name", "myName"],"required","false"],` `["type","String"],["validationExpression",""],` `"value','myValue"]]]`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[name databaseName][type string][value mys]]` `[name driverType][type integer][value 4]]` `[name serverName][type string][value localhost]]` `[[name portNumber][type integer][value 50000]] ]]]]`
relationalResourceAdapter	Specifies the relational resource adapter that the data source uses. The available Java 2 Connector (J2C) resource adapter ID of J2CResourceAdapterID can be identified with the `AdminConfig.list('J2CResourceAdapter')` command.	`[relationalResourceAdapter "WebSphere Relational Resource` `Adapter(cells/pongo/nodes/pongo/servers/server1|` `resources.xml#builtin_rra)"]`
statementCacheSize	Specifies the number of statements that the product can cache for each connection. The product optimizes the processing of prepared statements and callable statements by caching statements that are not used in an active connection. Both statement types improve the performance of transactions between an application and a datastore. Caching the statements makes them more readily available.	`['statementCacheSize', 5]`

Table 186. Optional attributes (continued). Several scripts have these attributes.

Attributes	Description	Example
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`['-xaRecoveryAuthAlias', 'myCellManager01/a1']`

Table 187. Optional attributes, continued. Several scripts have this attribute.

Attributes	Description	Example
providerType	Specifies the JDBC provider type that this JDBC provider uses.	`['providerType', 'DB2 Using IBM JCC Driver']`

Syntax

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)

AdminJDBC.createDataSourceUsingTemplate(nodeName,
 serverName, jdbcProvider, templateID, dsName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createDataSourceUsingTemplate("myNode", "myServer",
 "myJDBCProvider", "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)", "myDataSource")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JDBC Name2",
"DB2 Universal JDBC Driver DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_DB2_UNI_1)",
"MyJDBCDS", "authDataAlias=cellManager01/myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
datasourceHelperClassname=com.ibm.websphere.rsadapter.DB2DataStoreHelper, description='My description',
diagnoseConnectionUsage=true, jndiName=myJndiName, logMissingTransactionContext=false, manageCachedHandles=false,
providerType='DB2 Using IBM JCC Driver', xaRecoveryAuthAlias=myCellManager01/xa1")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JDBC Name2",
"DB2 Universal JDBC Driver DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_DB2_UNI_1)",
"MyJDBCDS", [['authDataAlias', 'cellManager01/myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', 100], ['connectionTimeout', 1000],
```

```
['freePoolDistributionTableSize', 10],
['maxConnections', 12],
['minConnections', 5], ['numberOfFreePoolPartitions', 3], ['numberOfSharedPoolPartitions', 6],
['numberOfUnsharedPoolPartitions', 3],
['properties', [[['name', 'name1a'], ['value', 'value1a']], [['name', 'name1b'], ['value', 'value1b']]]],
['purgePolicy', 'EntirePool'], ['reapTime', 10000], ['stuckThreshold', 3], ['stuckTime', 10], ['stuckTimerTime', 10],
['surgeThreshold', 10], ['testConnection', 'true']]], ['datasourceHelperClassname',
'com.ibm.websphere.rsadapter.DB2DataStoreHelper'],
['description', 'My description'], ['diagnoseConnectionUsage', 'true'], ['jndiName', 'myJndiName'],
['logMissingTransactionContext', 'false'],
['manageCachedHandles', 'false'], ['mapping', [['authDataAlias', 'anAlias'], ['mappingConfigAlias', 'anotherTest']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', 12343], ['retryLimit', 4]]],
 ['properties', [[['name', 'name1'],
['value', 'value1']], [['name', 'name2'], ['value', 'value2']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'String'], ['value', 'localhost']], [['name', 'portNumber'], ['type', 'integer'],
['value', 50000]]]]]], ['providerType', 'DB2 Using IBM JCC Driver'],
['relationalResourceAdapter',
'SIB JMS Resource Adapter(cells/IBM-F4A849C57A0Cell01/clusters/c1|resources.xml#J2CResourceAdapter_1232911649746)'],
['statementCacheSize', 5], ['xaRecoveryAuthAlias', 'myCellManager01/xa1']])
```

createDataSourceAtScope

This script creates a new data source in your configuration at the scope that you specify. The script returns the configuration ID of the new data source. The script procedure uses the createDatasource administrative command to create a new data source. The createDataSource script creates a new data source using the AdminConfig create command.

To run the script, specify the scope, JDBC provider, data source name, and database name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 188. createDataSourceAtScope script. Required and optional arguments.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JDBC provider.
jdbcProvider	Specifies the name of the JDBC provider of interest.
dsName	Specifies the name to assign to the new data source.
databaseName (*URL* for the Oracle database)	Specifies the name to assign the database for the JDBC provider. For a JDBC provider that uses the Oracle database, the argument specifies the URL of the database from which the datasource obtains connections. Examples are `jdbc:oracle:thin:@localhost:1521:sample` for a thin driver and `jdbc:oracle:oci8:@sample` for a thick driver.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 189. Optional attributes. Several scripts have these attributes.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
componentManagedAuthenticationAlias	Specifies the alias used for database authentication at runtime.
containerManagedPersistence	Specifies that container managed persistence is enabled when set to `true`.
description	Specifies a description of the data source.
driverType	Specifies the data source type. The data source type is valid only for JDBC providers that have a database type of `DB2`.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name for this data source.
portNumber	Specifies the port number of the database server. The port number is valid only for JDBC providers that have a database type of `DB2`, `Informix`, `Sybase`, or `SQLServer`.
serverName	Specifies the host name of the database server or IP address. The server name is valid only for JDBC providers that have a database type of `DB2`, `Informix`, `Sybase`, or `SQLServer`. For the Informix® JDBC Driver, the *serverName* refers to the name of the Informix instance. Example: ol_myserver.

Table 189. Optional attributes (continued). Several scripts have these attributes.

Attributes	Description
ifxIFHOST	Specifies the physical machine name of the server hosting the Informix instance. You can enter a host name or IP address. You can also enter an Internet Protocol Version 6 (IPv6) if the host database supports it. This attribute is valid only for JDBC providers that have a database type of `Informix`.
informixLockModeWait	Specifies the connection wait time for obtaining a lock on the database. By default, the Informix database returns an error when it cannot acquire a lock, rather than wait for the current owner of the lock to release it. To modify the behavior, set the property to the number of seconds to wait for a lock. The default is 2 seconds. Any negative value translates into an unlimited wait time. This attribute is valid only for JDBC providers that have a database type of `Informix`.
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.

Syntax

```
AdminJDBC.createDataSourceAtScope(scope,
 jdbcProvider, dsName, databaseName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createDataSourceAtScope("myScope",  "myJDBCProvider",
 "myDataSource", "myDatabase")
```

Examples scripts for the DB2 database type:

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"MyTestJDBCProviderName", "newds2", "newds2/jndi", "com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper", "db1",
" category=myCategory, componentManagedAuthenticationAlias=CellManager01/AuthDataAliase, containerManagedPersistence=true,
description='My description', xaRecoveryAuthAlias=CellManager01/xaAliase", "serverName=localhost,
driverType=4,portNumber=50000")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"MyTestJDBCProviderName", "newds2", "newds2/jndi", "com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper",
"db1", [['category', 'myCategory'], ['componentManagedAuthenticationAlias',
'CellManager01/AuthDataAliase'], ['containerManagedPersistence', 'true'], ['description', 'My description'],
['xaRecoveryAuthAlias', 'CellManager01/xaAliase']] , [['serverName', 'localhost'],
['driverType', 4], ['portNumber', 50000]])
```

Examples scripts for the Derby database:

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Derby JDBC Provider", "Derby DataSource", "newds2/jndi", "com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper",
"db1",  " category=myCategory, componentManagedAuthenticationAlias=CellManager01/AuthDataAliase,
containerManagedPersistence=true, description=My description, xaRecoveryAuthAlias=CellManager01/xaAliase")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Derby JDBC Provider", "Derby DataSource", "newds2/jndi", "com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper",
"db1",  [['category', 'myCategory'], ['componentManagedAuthenticationAlias', 'CellManager01/AuthDataAliase'],
['containerManagedPersistence', 'true'], ['description', 'My description'], ['xaRecoveryAuthAlias',
'CellManager01/xaAliase']])
```

Examples scripts for the Informix database:

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01", "Informix JDBC Driver Test", "My DataSource",
"My JNDIName", "com.ibm.websphere.rsadapter.InformixDataStoreHelper", "MyDB", " category=myCategory,
componentManagedAuthenticationAlias=CellManager01/AuthDataAliase, containerManagedPersistence=true,
description='My description',
xaRecoveryAuthAlias=CellManager01/xaAliase", "serverName=ol_myserver, portNumber=50000, ifxIFXHOST=localhost,
informixLockModeWait=2")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01", "Informix JDBC Driver Test", "My DataSource",
"My JNDIName", "com.ibm.websphere.rsadapter.InformixDataStoreHelper", "MyDB", [['category', 'myCategory'],
['componentManagedAuthenticationAlias', 'CellManager01/AuthDataAliase'], ['containerManagedPersistence', 'true'],
['description', 'My description'], ['xaRecoveryAuthAlias', 'CellManager01/xaAliase']] , [['serverName', 'ol_myserver'],
['portNumber', 1526], ['ifxIFXHOST', 'localhost'], ['informixLockModeWait', 2]])
```

Examples scripts for the Oracle database:

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Oracle JDBC Driver", "My DataSource", "My JNDIName", "com.ibm.websphere.rsadapter.OracleDataStoreHelper",
"http://myURL.com",  "category=myCategory, componentManagedAuthenticationAlias=CellManager01/AuthDataAliase,
containerManagedPersistence=true, description='My description', xaRecoveryAuthAlias=CellManager01/xaAliase")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Oracle JDBC Driver", "My DataSource", "My JNDIName", "com.ibm.websphere.rsadapter.OracleDataStoreHelper",
"http://myURL.com",
[['category', 'myCategory'], ['componentManagedAuthenticationAlias', 'CellManager01/AuthDataAliase'],
['containerManagedPersistence', 'true'], ['description', 'My description'], ['xaRecoveryAuthAlias',
 'CellManager01/xaAliase']])
```

Examples scripts for the SQLServer database:

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Microsoft SQL Server JDBC Driver", "My DataSource", "My JNDIName", "com.ibm.websphere.rsadapter.SQLserverDataStoreHelper",
"myDBName", " category=myCategory, componentManagedAuthenticationAlias=CellManager01/AuthDataAliase,
containerManagedPersistence=true, description='My description', xaRecoveryAuthAlias=CellManager01/xaAliase",
"serverName=localhost, portNumber=1433")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Microsoft SQL Server JDBC Driver", "My DataSource", "My JNDIName", "com.ibm.websphere.rsadapter.SQLserverDataStoreHelper",
"myDBName", [['category', 'myCategory'], ['componentManagedAuthenticationAlias', 'CellManager01/AuthDataAliase'],
['containerManagedPersistence', 'true'], ['description', 'My description'], ['xaRecoveryAuthAlias', 'CellManager01/xaAliase']] ,
[['serverName', 'localhost'], ['portNumber', 1433]])
```

Examples scripts for the Sybase database:

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Sybase JDBC 3 Driver", "My DataSource", "My JNDIName", "com.ibm.websphere.rsadapter.SybaseserverDataStoreHelper",
"myDBName", " category=myCategory, componentManagedAuthenticationAlias=CellManager01/AuthDataAliase,
containerManagedPersistence=true, description='My description', xaRecoveryAuthAlias=CellManager01/xaAliase",
"serverName=localhost, portNumber=1433")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"Sybase JDBC 3 Driver", "My DataSource", "My JNDIName", "com.ibm.websphere.rsadapter.SybaseserverDataStoreHelper",
"myDBName", [['category', 'myCategory'], ['componentManagedAuthenticationAlias', 'CellManager01/AuthDataAliase'],
['containerManagedPersistence', 'true'], ['description', 'My description'], ['xaRecoveryAuthAlias', 'CellManager01/xaAliase']] ,
[['serverName', 'localhost'], ['portNumber', 2638]])
```

createDataSourceUsingTemplateAtScope

This script uses a template to create a new data source in your configuration at the scope that you specify. The script returns the configuration ID of the new data source.

To run the script, specify the scope, JDBC provider, template ID, and data source name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 190. createDataSourceUsingTemplateAtScope script. Required and optional arguments.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JDBC provider.
jdbcProvider	Specifies the name of the JDBC provider of interest.

Table 190. createDataSourceUsingTemplateAtScope script (continued). Required and optional arguments.

Argument	Description
templateID	Specifies the configuration ID of the template to use to create the data source.
dsName	Specifies the name to assign to the new data source.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 191. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
authDataAlias	Specifies the alias used for database authentication at run time.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication.	
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JDBC connection pooling properties for the parent JDBC connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["connectionPool",[["agedTimeout","100"],` `["connectionTimeout","1000"],` `["freePoolDistributionTableSize",10],` `["maxConnections","12"],["minConnections","5"],` `["numberOfFreePoolPartitions","3"],` `["numberOfSharedPoolPartitions","6"],` `["numberOfUnsharedPoolPartitions","3"],` `["properties",[["description","My description"],` `["name","myName"],["required","false"],["type","String"],` `["validationExpression",""],["value","myValue"]]],` `["purgePolicy","EntirePool"],["reapTime","10000"],` `["struckThreshold","3"],["struckTime","10"],` `["struckTimerTime","10"],["surgeCreationInterval","10"],` `["surgeThreshold","10"],["testConnection","true"],` `["testConnectionInterval","10"],` `["unusedTimeout","10000"]]]`
datasourceHelperClassname	Specifies the name of the DataStoreHelper implementation class that extends the capabilities of the implementation class of the JDBC driver. The extended capabilities allow the JDBC drive to perform functions that are specific to the data.	`com.ibm.websphere.rsadapter.DB2DataStoreHelper` `com.ibm.websphere.rsadapter.DerbyDataStoreHelper` `...`
description	Specifies a description of the data source.	`['description', 'My description']`
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name for this data source.	`['jndiName', 'myJndiName']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'false']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'false']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"],` `["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"],` `"retryInterval", "12343"],"retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"],` `["name", "myName"],"required","false"],` `["type","String"],["validationExpression",""],` `"value','myValue"]]]`

Table 191. Optional attributes (continued). Several scripts have these attributes.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[name databaseName][type string][value mys]] [name driverType][type integer][value 4]] [name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
relationalResourceAdapter	Specifies the relational resource adapter that the data source uses. The available Java 2 Connector (J2C) resource adapter ID of J2CResourceAdapterID can be identified with the `AdminConfig.list('J2CResourceAdapter')` command.	`[relationalResourceAdapter "WebSphere Relational Resource Adapter(cells/pongo/nodes/pongo/servers/server1| resources.xml#builtin_rra)"]`
statementCacheSize	Specifies the number of statements that the product can cache for each connection. The product optimizes the processing of prepared statements and callable statements by caching statements that are not used in an active connection. Both statement types improve the performance of transactions between an application and a datastore. Caching the statements makes them more readily available.	`['statementCacheSize', 5]`
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`['-xaRecoveryAuthAlias', 'myCellManager01/a1']`

Table 192. Optional attributes, continued. Several scripts have this attribute.

Attributes	Description	Example
providerType	Specifies the JDBC provider type that this JDBC provider uses.	`['providerType', 'DB2 Using IBM JCC Driver']`

Syntax

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)

AdminJDBC.createDataSourceUsingTemplateAtScope(nodeName,
 serverName, jdbcProvider, templateID, dsName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createDataSourceUsingTemplateAtScope("myNode", "myServer",
 "myJDBCProvider", "Derby JDBC Driver
 DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_derby_1)", "myDataSource")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createDataSourceUsingTemplateAtScope("/Cell:IBM-F4A849C57A0Cell01/Node:IBM-F4A849C57A0Node01/Server:server1",
"My JDBC Name2",
"DB2 Universal JDBC Driver DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_DB2_UNI_1)",
"MyJDBCDS", "authDataAlias=cellManager01/myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
datasourceHelperClassname=com.ibm.websphere.rsadapter.DB2DataStoreHelper, description='My description',
diagnoseConnectionUsage=true, jndiName=myJndiName, logMissingTransactionContext=false, manageCachedHandles=false,
providerType='DB2 Using IBM JCC Driver', xaRecoveryAuthAlias=myCellManager01/xa1")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createDataSourceUsingTemplateAtScope("/Cell:IBM-F4A849C57A0Cell01/Node:IBM-F4A849C57A0Node01/Server:server1",
"My JDBC Name2",
"DB2 Universal JDBC Driver DataSource(templates/system|jdbc-resource-provider-templates.xml#DataSource_DB2_UNI_1)",
"MyJDBCDS", [['authDataAlias', 'cellManager01/myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', 100], ['connectionTimeout', 1000],
['freePoolDistributionTableSize', 10],
['maxConnections', 12], ['minConnections', 5], ['numberOfFreePoolPartitions', 3], ['numberOfSharedPoolPartitions', 6],
['numberOfUnsharedPoolPartitions', 3], ['properties', [[['name', 'name1a'], ['value', 'value1a']], [['name', 'name1b'],
['value', 'value1b']]]], ['purgePolicy', 'EntirePool'], ['reapTime', 10000], ['stuckThreshold', 3], ['stuckTime', 10],
['stuckTimerTime', 10], ['surgeThreshold', 10], ['testConnection', 'true']]],
['datasourceHelperClassname', 'com.ibm.websphere.rsadapter.DB2DataStoreHelper'],
['description', 'My description'], ['diagnoseConnectionUsage', 'true'], ['jndiName', 'myJndiName'],
['logMissingTransactionContext', 'false'],
['manageCachedHandles', 'false'], ['mapping', [['authDataAlias', 'anAlias'], ['mappingConfigAlias', 'anotherTest']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', 12343], ['retryLimit', 4]]],
['properties', [[['name', 'name1'],
['value', 'value1']], [['name', 'name2'], ['value', 'value2']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'],
['type', 'String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'integer'], ['value', 4]], [['name', 'serverName'],
['type', 'String'], ['value', 'localhost']], [['name', 'portNumber'], ['type', 'integer'], ['value', 50000]]]]],
['providerType', 'DB2 Using IBM JCC Driver'],
['relationalResourceAdapter',
'SIB JMS Resource Adapter(cells/IBM-F4A849C57A0Cell01/clusters/c1|resources.xml#J2CResourceAdapter_1232911649746)'],
['statementCacheSize', 5], ['xaRecoveryAuthAlias', 'myCellManager01/xa1']] )
```

createJDBCProvider

This script creates a new JDBC provider in your environment. The script returns the configuration ID of the new JDBC provider.

To run the script, specify the node name, server name, JDBC provider, and implementation class arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 193. createJDBCProvider script. Required and optional arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jdbcProvider	Specifies the name to assign to the new JDBC provider.
implementationClass	Specifies the name of the implementation class to use.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 194. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`['-classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar; ${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
isolatedClassLoader	If set to `true`, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'false']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`['-nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}']`

Table 195. Optional attributes, continued. Several scripts have these attributes.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JDBC provider type that this JDBC provider uses.	`['providerType', 'DB2 Universal JDBC Driver Provider']`
xa	Possible values are `true` and `false`. If set to `true`, data sources for the provider produce connections that applications use in two-phase commit, global transactions. If set to `false`, the data sources produce connections that applications use in single-phase commit, local transactions.	`true` `false`

Syntax

Implementation class optional attribute

Syntax

Use the following command syntax to find the implementationClassName attribute by specifying the JDBC provider template ID for JDBCProvID:

```
AdminConfig.showAttribute(JDBCProvID,'implementationClassName')
```

implementationClassName attribute example usage:

```
print AdminConfig.showAttribute("DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)",
"implementationClassName")
```

Result:

```
com.ibm.db2.jcc.DB2XADataSource
```

Some possible implementation class names:

com.ibm.db2.jcc.DB2ConnectionPoolDataSource

com.ibm.db2.jcc.DB2XADataSource

com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource
com.ibm.db2.jdbc.app.UDBXADataSource
com.ibm.as400.access.AS400JDBCConnectionPoolDataSource
com.ibm.as400.access.AS400JDBCXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource
org.apache.derby.jdbc.ClientXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource40
org.apache.derby.jdbc.ClientXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40
org.apache.derby.jdbc.EmbeddedXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
org.apache.derby.jdbc.EmbeddedXADataSource
com.informix.jdbcx.IfxConnectionPoolDataSource
com.informix.jdbcx.IfxXADataSource oracle.jdbc.pool.OracleConnectionPoolDataSource
oracle.jdbc.xa.client.OracleXADataSource
com.sybase.jdbc3.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc3.jdbc.SybXADataSource
com.sybase.jdbc4.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc4.jdbc.SybXADataSource
com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource
com.microsoft.sqlserver.jdbc.SQLServerXADataSource
com.ddtek.jdbcx.sqlserver.SQLServerDataSource

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)

AdminJDBC.createJDBCProvider(nodeName, serverName,
 jdbcProvider, implementationClass, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createJDBCProvider("myNode", "myServer", "myJDBCProvider",
 "myImplementationClass")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createJDBCProvider("IBM-F4A849C57A0Node01", "server1", "My JDBC Name",
"com.ibm.db2.jcc.DB2ConnectionPoolDataSource",
" classpath=${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar,
description='My description', isolatedClassLoader=false, nativepath=${DB2_JCC_DRIVER_NATIVEPATH},
providerType='DB2 Univesal JDBC Driver Provider', xa=true ")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createJDBCProvider("IBM-F4A849C57A0Node01", "server1", "My JDBC Name",
"com.ibm.db2.jcc.DB2ConnectionPoolDataSource",
[['classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'],
['description', 'My description'], ['isolatedClassLoader', 'false'], ['nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}'],
['providerType', 'DB2 Univesal JDBC Driver Provider'], ['xa', 'true'], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'integer'],
['value', 4]], [['name', 'serverName'], ['type', 'String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'Integer'], ['value', 50000]]]]]]])
```

createJDBCProviderUsingTemplate

This script uses a template to create a new JDBC provider in your environment. The script returns the configuration ID of the new JDBC provider.

To run the script, specify the node name, server name, template ID, JDBC provider name, and implementation class arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 196. createJDBCProviderUsingTemplate script. Required and optional arguments.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
templateID	Specifies the configuration ID of the template to use to create the JDBC provider.
jdbcProvider	Specifies the name to assign to the new JDBC provider.
implementationClass	Specifies the name of the implementation class to use.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 197. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`['-classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar; ${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
isolatedClassLoader	If set to `true`, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'false']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`['-nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}']`

Table 198. Optional attributes, continued. Several scripts have these attributes.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JDBC provider type that this JDBC provider uses.	`['providerType', 'DB2 Universal JDBC Driver Provider']`
xa	Possible values are `true` and `false`. If set to `true`, data sources for the provider produce connections that applications use in two-phase commit, global transactions. If set to `false`, the data sources produce connections that applications use in single-phase commit, local transactions.	`true` `false`

Syntax

Implementation class optional attribute

Syntax

Use the following command syntax to find the implementationClassName attribute by specifying the JDBC provider template ID for JDBCProvID:

```
AdminConfig.showAttribute(JDBCProvID,'implementationClassName')
```

implementationClassName attribute example usage:

```
print AdminConfig.showAttribute("DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)",
"implementationClassName")
```

Result:

```
com.ibm.db2.jcc.DB2XADataSource
```

Some possible implementation class names:

com.ibm.db2.jcc.DB2ConnectionPoolDataSource
com.ibm.db2.jcc.DB2XADataSource
com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource
com.ibm.db2.jdbc.app.UDBXADataSource
com.ibm.as400.access.AS400JDBCConnectionPoolDataSource
com.ibm.as400.access.AS400JDBCXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource
org.apache.derby.jdbc.ClientXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource40
org.apache.derby.jdbc.ClientXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40
org.apache.derby.jdbc.EmbeddedXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
org.apache.derby.jdbc.EmbeddedXADataSource
com.informix.jdbcx.IfxConnectionPoolDataSource

com.informix.jdbcx.IfxXADataSource oracle.jdbc.pool.OracleConnectionPoolDataSource
oracle.jdbc.xa.client.OracleXADataSource
com.sybase.jdbc3.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc3.jdbc.SybXADataSource
com.sybase.jdbc4.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc4.jdbc.SybXADataSource
com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource
com.microsoft.sqlserver.jdbc.SQLServerXADataSource
com.ddtek.jdbcx.sqlserver.SQLServerDataSource

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)

AdminJDBC.createJDBCProviderUsingTemplate(nodeName,
 serverName, templateID, jdbcProvider,
 implementationClass, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createJDBCProviderUsingTemplate("myNode", "myServer", "Derby JDBC
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#JDBCProvider_1124467079638)",
 "myJDBCProvider", "myImplementationClass")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createJDBCProviderUsingTemplate("IBM-F4A849C57A0Node01", "server1",
"DB2 Universal JDBC Driver Provider(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_1)",
"My JDBC Name", "com.ibm.db2.jcc.DB2ConnectionPoolDataSource",
"classpath= ${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;
${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar,
description='My description', isolatedClassLoader=false, nativepath=${DB2_JCC_DRIVER_NATIVEPATH},
providerType='DB2 Univesal JDBC Driver Provider', xa=true ")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createJDBCProviderUsingTemplate("IBM-F4A849C57A0Node01", "server1",
"DB2 Universal JDBC Driver Provider(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_1)",
 My JDBC Name",
"com.ibm.db2.jcc.DB2ConnectionPoolDataSource",
[['classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;
${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'],
['description', 'My description'], ['isolatedClassLoader', 'false'], ['nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}'],
```

```
['providerType', 'DB2 Univesal JDBC Driver Provider'], ['xa', 'true'], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'integer'],
['value', 4]], [['name', 'serverName'], ['type', 'String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'Integer'], ['value', 50000]]]]]]])
```

createJDBCProviderAtScope

This script creates a new JDBC provider in your environment at the scope that you specify. The script returns the configuration ID of the new JDBC provider. The script procedure uses the createJDBCProvider administrative command to create a new JDBC provider. The createJDBCProvider script procedure creates a new JDBC provider by using the AdminConfig create command.

To run the script, specify the scope, JDBC provider, database type, provider type, and implementation types arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 199. createJDBCProviderAtScope script. Required and optional arguments.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JDBC provider.
jdbcProvider	Specifies the name to assign to the new JDBC provider.
databaseType	Specifies the database type that this JDBC provider uses. Valid values include `DB2`, `Derby`, `Informix`, `Oracle`, `Sybase`, `SQL Server`, and user-defined values.
providerType	Specifies the JDBC provider type that this JDBC provider uses.
implementationType	Specifies the implementation type that this JDBC provider uses. Valid values are `Connection pool datasource` and `XA data source`.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 200. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`['-classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar; ${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
isolatedClassLoader	If set to `true`, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'false']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`['-nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}']`

Table 201. Optional attributes, continued. Several scripts have this attribute.

Attributes	Description
implementationClassName	Specifies the implementation class to use for a given JDBC provider template.

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)
```

implementationClassName attribute

Syntax

Use the following command syntax to find the implementationClassName attribute by specifying the JDBC provider template ID for JDBCProvID:

```
AdminConfig.showAttribute(JDBCProvID,'implementationClassName')
```

implementationClassName attribute example usage:

```
print AdminConfig.showAttribute("DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)",
"implementationClassName")
```

Result:

```
com.ibm.db2.jcc.DB2XADataSource
```

Some possible implementation class names:

com.ibm.db2.jcc.DB2ConnectionPoolDataSource
com.ibm.db2.jcc.DB2XADataSource
com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource
com.ibm.db2.jdbc.app.UDBXADataSource
com.ibm.as400.access.AS400JDBCConnectionPoolDataSource
com.ibm.as400.access.AS400JDBCXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource
org.apache.derby.jdbc.ClientXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource40
org.apache.derby.jdbc.ClientXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40
org.apache.derby.jdbc.EmbeddedXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
org.apache.derby.jdbc.EmbeddedXADataSource
com.informix.jdbcx.IfxConnectionPoolDataSource
com.informix.jdbcx.IfxXADataSource oracle.jdbc.pool.OracleConnectionPoolDataSource

oracle.jdbc.xa.client.OracleXADataSource
com.sybase.jdbc3.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc3.jdbc.SybXADataSource
com.sybase.jdbc4.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc4.jdbc.SybXADataSource
com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource
com.microsoft.sqlserver.jdbc.SQLServerXADataSource
com.ddtek.jdbcx.sqlserver.SQLServerDataSource

createJDBCProviderAtScope script

Syntax

```
AdminJDBC.createJDBCProviderAtScope(scope,
 jdbcProvider, databaseType, providerType, implementationType,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createJDBCProviderAtScope("myScope", "myJDBCProvider", "myDatabaseType","myProviderType",
"myImplementationClass")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createJDBCProviderAtScope("/Cell:IBM-F4A849C57A0Cell01/Node:IBM-F4A849C57A0Node01", "DB2",
"DB2 Universal JDBC Driver Provider", "Connection pool data source", "My JDBCProvider Name",
"description='My description', implementationClassName=com.ibm.db2.jcc.DB2ConnectionPoolDataSource,
classpath=${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;
${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar,
nativePath=${DB2_JCC_DRIVER_NATIVEPATH}, isolated=false")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createJDBCProviderAtScope("/Cell:IBM-F4A849C57A0Cell01/Node:IBM-F4A849C57A0Node01", "DB2",
"DB2 Universal JDBC Driver Provider", "Connection pool data source", "My JDBCProvider Name", [['description', 'My description'],
[ 'implementationClassName ', 'com.ibm.db2.jcc.DB2ConnectionPoolDataSource'],
[ 'classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;
${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'],
[ 'nativePath', '${DB2_JCC_DRIVER_NATIVEPATH}'], ['isolated', 'false'] ])
```

createJDBCProviderUsingTemplateAtScope

This script uses a template to create a new JDBC provider in your environment at the scope that you specify. The script returns the configuration ID of the new JDBC provider.

To run the script, specify the scope, template ID, JDBC provider name, and implementation class arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 202. createJDBCProviderUsingTemplateAtScope script. Required and optional arguments.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JDBC provider.
templateID	Specifies the configuration ID of the template to use to create the JDBC provider.
jdbcProvider	Specifies the name to assign to the new JDBC provider.
implementationClassName	Specifies the name of the implementation class to use.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 203. Optional attributes. Several scripts have these attributes.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`['-classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar; ${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
isolatedClassLoader	If set to `true`, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'false']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`['-nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}']`

Table 204. Optional attributes, continued. Several scripts have these attributes.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JDBC provider type that this JDBC provider uses.	`['providerType', 'DB2 Universal JDBC Driver Provider']`
xa	Possible values are `true` and `false`. If set to `true`, data sources for the provider produce connections that applications use in two-phase commit, global transactions. If set to `false`, the data sources produce connections that applications use in single-phase commit, local transactions.	`true` `false`

Implementation class optional attribute

Syntax

Use the following command syntax to find the implementationClassName attribute by specifying the JDBC provider template ID for JDBCProvID:

```
AdminConfig.showAttribute(JDBCProvID,'implementationClassName')
```

implementationClassName attribute example usage:

```
print AdminConfig.showAttribute("DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)",
"implementationClassName")
```

Result:

```
com.ibm.db2.jcc.DB2XADataSource
```

Some possible implementation class names:

com.ibm.db2.jcc.DB2ConnectionPoolDataSource

com.ibm.db2.jcc.DB2XADataSource

com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource

com.ibm.db2.jdbc.app.UDBXADataSource
com.ibm.as400.access.AS400JDBCConnectionPoolDataSource
com.ibm.as400.access.AS400JDBCXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource
org.apache.derby.jdbc.ClientXADataSource
org.apache.derby.jdbc.ClientConnectionPoolDataSource40
org.apache.derby.jdbc.ClientXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40
org.apache.derby.jdbc.EmbeddedXADataSource40
org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
org.apache.derby.jdbc.EmbeddedXADataSource
com.informix.jdbcx.IfxConnectionPoolDataSource
com.informix.jdbcx.IfxXADataSource oracle.jdbc.pool.OracleConnectionPoolDataSource
oracle.jdbc.xa.client.OracleXADataSource
com.sybase.jdbc3.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc3.jdbc.SybXADataSource
com.sybase.jdbc4.jdbc.SybConnectionPoolDataSource
com.sybase.jdbc4.jdbc.SybXADataSource
com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource
com.microsoft.sqlserver.jdbc.SQLServerXADataSource
com.ddtek.jdbcx.sqlserver.SQLServerDataSource

providerType optional attribute

Syntax

Use the following command syntax to find the JDBC provider type name. Only JDBC provider template IDs that contain the templates/system|jdbc-resource-provider-templates.xml substring have valid JDBC Provider type names. The JDBC provider type name and its substring form the JDBC provider template ID.

```
AdminConfig.listTemplates('JDBCProvider')
```

Example partial result showing the JDBC provider template ID for the JDBC provider type name of Cloudscape JDBC Provider (XA):

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)
```

Some JDBC provider template IDs:

```
Cloudscape JDBC Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"

Cloudscape Network Server Using Universal JDBC Driver
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2jN_1)

DB2 Legacy CLI-based Type 2 JDBC Driver (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_4)

DB2 UDB for iSeries (Native - V5R1 and earlier)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2400_5)"

DB2 Universal JDBC Driver Provider (XA)
(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_2)
```

createJDBCProviderUsingTemplateAtScope script

Syntax

```
AdminJDBC.createJDBCProviderUsingTemplateAtScope(scope,
  templateID, jdbcProvider,
 implementationClass, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJDBC.createJDBCProviderUsingTemplateAtScope("myScope", "Derby JDBC
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#JDBCProvider_1124467079638)",
 "myJDBCProvider", "myImplementationClass")
```

The following example script includes optional attributes in a string format:

```
AdminJDBC.createJDBCProviderUsingTemplateAtScope("/Cell:IBM-F4A849C57A0Cell01/ServerCluster:cluster1",
"DB2 Universal JDBC Driver Provider(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_1)",
"My JDBC Name001", " com.ibm.db2.jcc.DB2ConnectionPoolDataSource",
"classpath= ${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;
${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar,
description='My description', isolatedClassLoader=false, nativepath=${DB2_JCC_DRIVER_NATIVEPATH},
providerType='DB2 Univesal JDBC Driver Provider', xa=true ")
```

The following example script includes optional attributes in a list format:

```
AdminJDBC.createJDBCProviderUsingTemplateAtScope("/Cell:IBM-F4A849C57A0Cell01/ServerCluster:cluster1",
"DB2 Universal JDBC Driver Provider(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_DB2_UNI_1)",
"My JDBC Name001", " com.ibm.db2.jcc.DB2ConnectionPoolDataSource",
[['classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;
${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'],
['description', 'My description'], ['isolatedClassLoader', 'false'], ['nativepath', '${DB2_JCC_DRIVER_NATIVEPATH}'],
['providerType', 'DB2 Univesal JDBC Driver Provider'], ['xa', 'true'], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'integer'],
['value', 4]], [['name', 'serverName'], ['type', 'String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'Integer'],
['value', 50000]]]]]])
```

JDBC query scripts

The scripting library provides many script procedures to manage Java Database Connectivity (JDBC) configurations in your environment. This topic provides usage information for scripts that retrieve configuration IDs for your JDBC configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each AdminJDBC script procedure is located in the *app_server_root*/scriptLibraries/resources/JDBC/V70 directory.

Beginning with Version 7, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/`scriptLibraries` directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Fast path: Beginning with Fix Pack 5, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the cell, node, server, or cluster scope. Resource providers include mail providers, URL providers, and resource environment providers. You do not have to write custom scripts to configure resources at a particular scope.

Use the following script procedures to query your JDBC configuration:

- “listDataSources”
- “listDataSourceTemplates” on page 188
- “listJDBCProviders” on page 188
- “listJDBCProviderTemplates” on page 188

listDataSources

This script displays a list of configuration IDs for the data sources in your configuration.

No input arguments are required for the script. However, you can specify a data source name to return a specific configuration id, as defined in the following table:

Table 205. listDataSources script. Run the script to list data sources.

Argument	Description
dsName	Optionally specifies the name of the data source of interest.

Syntax

```
AdminJDBC.listDataSources(dsName)
```

Example usage

```
AdminJDBC.listDataSources()
AdminJDBC.listDataSources("myDataSource")
```

listDataSourceTemplates

This script displays a list of configuration IDs for the data source templates in your environment.

No input arguments are required for the script. However, you can specify a template name to return a specific configuration id, as defined in the following table:

Table 206. listDataSourceTemplates script. Run the script to list data source templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJDBC.listDataSourceTemplates(templateName)
```

Example usage

```
AdminJDBC.listDataSourceTemplates()
AdminJDBC.listDataSourceTemplates("Derby JDBC Driver DataSource")
```

listJDBCProviders

This script displays a list of configuration IDs for the JDBC providers in your environment.

No input arguments are required for the script. However, you can specify a JDBC provider name to return a specific configuration id, as defined in the following table:

Table 207. listJDBCProviders script. Run the script to list JDBC providers.

Argument	Description
jdbcName	Optionally specifies the name of the JDBC provider of interest.

Syntax

```
AdminJDBC.listJDBCProviders(jdbcName)
```

Example usage

```
AdminJDBC.listJDBCProviders()
AdminJDBC.listJDBCProviders("myJDBCProvider")
```

listJDBCProviderTemplates

This script displays a list of configuration IDs for the JDBC provider templates in your environment.

No input arguments are required for the script. However, you can specify a template name to return a specific configuration id, as defined in the following table:

Table 208. listJDBCProviderTemplates script. Run the script to list JDBC provider templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJDBC.listJDBCProviderTemplates(templateName)
```

Example usage

```
AdminJDBC.listJDBCProviderTemplates()
AdminJDBC.listJDBCProviderTemplates("Derby JDBC Provider")
```

Automating messaging resource configurations using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the resource management scripts to configure and manage your Java Messaging Service (JMS) configurations.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
      "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication",
      "..\installableApps\DefaultApplication.ear", "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The messaging resource management procedures in the scripting library are located in the *app_server_root*/scriptLibraries/resources/JMS/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your custom Jython scripts (*.py) when the wsadmin tool starts, save your automation scripts to a new subdirectory in the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the scripts to perform multiple combinations of administration functions. Use the following sample combination of procedures to create a JMS provider and configure JMS resources for the JMS provider.

Procedure

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the `bin` directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   - Enter the following command from the `bin` directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
bin>wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Configure a JMS provider.

   Run the createJMSProvider procedure from the script library and specify the required arguments. To run the script, specify the node, server, JMS provider name, external initial contextual factory name, and external provider URL. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. The following table provides additional information about the arguments to specify:

Table 209. createJMSProvider script arguments. Run the script to create a JMS provider.

Argument	Description
Node name	Specifies the name of the node of interest.
Server name	Specifies the name of the server of interest.
JMS provider name	Specifies the name to assign to the new JMS provider.
External initial contextual factory name	Specifies the Java class name of the initial context factory for the JMS provider.
External provider URL	Specifies the JMS provider URL for external JNDI lookups.

   The following example creates a JMS provider in your configuration:

```
bin>wsadmin -lang jython -c "AdminJMS.createJMSProvider("myNode", "myServer", "myJMSProvider",
   "extInitCF", "extPURL", [["description", "testing"], ["supportsASF", "true"],
   ["providerType", "jmsProvType"]])"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJMS.createJMSProvider("myNode", "myServer", "myJMSProvider", "extInitCF",
   "extPURL", [["description", "testing"], ["supportsASF", "true"], ["providerType",
   "jmsProvType"]])
```

   The script returns the configuration ID of the new JMS provider.

3. Configure a generic JMS connection factory.

   Run the createGenericJMSConnectionFactory procedure from the script library and specify the required arguments. To run the script, specify the node, server, JMS provider name, name of the new connection factory, JNDI name, and external JNDI name. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. The following table provides additional information about the arguments to specify:

Table 210. createGenericJMSConnectionFactory script arguments. Run the script to create a generic JMS connection factory.

Argument	Description
Node name	Specifies the name of the node of interest.
Server name	Specifies the name of the server of interest.
JMS provider name	Specifies the name of the JMS provider.
Connection factory name	Specifies the name to assign to the new connection factory
JNDI name	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
External JNDI name	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

The following example creates a JMS connection factory in your configuration:

```
bin>wsadmin -lang jython -c "AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer",
  "myJMSProvider", "JMSCFTest", "jmsjndi", "extjmsjndi", [["XAEnabled", "true"],
  ["authDataAlias", "myalias"], ["description", "testing"]])"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer", "myJMSProvider",
  "JMSCFTest", "jmsjndi", "extjmsjndi", [["XAEnabled", "true"], ["authDataAlias",
  "myalias"], ["description", "testing"]])
```

The script returns the configuration ID of the new generic JMS connection factory.

4. Create a generic JMS destination.

   Run the createGenericJMSDestination procedure from the script library and specify the required arguments. To run the script, specify the node, server, JMS provider name, generic JMS destination name, JNDI name, and external JNDI name. You can optionally specify additional attributes in the following format: `[["attr1", "value1"], ["attr2", "value2"]]`. The following table provides additional information about the arguments to specify:

Table 211. createGenericJMSDestination script arguments. Run the script to create a generic JMS destination.

Argument	Description
Node name	Specifies the name of the node of interest.
Server name	Specifies the name of the server of interest.
JMS provider name	Specifies the name of the JMS provider.
Generic JMS destination name	Specifies the name to assign to the new generic JMS destination.
JNDI name	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
External JNDI name	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

The following example uses a template to use a template to create a generic JMS destination in your configuration:

```
bin>wsadmin -lang jython -c "AdminJMS.createGenericJMSDestination("myNode", "myServer", "myJMSProvider",
   "JMSDest", "destjndi", "extDestJndi", [["description", "testing"], ["category",
   "jmsDestCatagory"], ["type", "TOPIC"]])"
```

You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminJMS.createGenericJMSDestination("myNode", "myServer", "myJMSProvider", "JMSDest",
   "destjndi", "extDestJndi", [["description", "testing"], ["category", "jmsDestCatagory"],
   ["type", "TOPIC"]])
```

The script returns the configuration ID of the new generic JMS destination.

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The

AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

JMS configuration scripts

The scripting library provides many script procedures to manage your Java Messaging Service (JMS) configurations. This topic provides usage information for scripts that query your JMS configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each AdminJMS management script procedure is located in the *app_server_root*/`scriptLibraries/resources/JMS/V70` directory.

Beginning with Version 7, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/`scriptLibraries` directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Fast path: Beginning with Fix Pack 5, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the cell, node, server, or cluster scope. Resource providers include mail providers, URL providers, and resource environment providers. You do not have to write custom scripts to configure resources at a particular scope.

Attention: The example usage scripts and the script syntax are split on multiple lines for printing purposes.

Use the following script procedures to configure JMS in your environment:

- “createGenericJMSConnectionFactory” on page 194
- “createGenericJMSConnectionFactoryUsingTemplate” on page 197
- “createGenericJMSConnectionFactoryAtScope” on page 200
- “createGenericJMSConnectionFactoryUsingTemplateAtScope” on page 203
- “createGenericJMSDestination” on page 206
- “createGenericJMSDestinationUsingTemplate” on page 207
- “createGenericJMSDestinationAtScope” on page 208
- “createGenericJMSDestinationUsingTemplateAtScope” on page 210
- “createJMSProvider” on page 211
- “createJMSProviderUsingTemplate” on page 213
- “createJMSProviderAtScope” on page 214
- “createJMSProviderUsingTemplateAtScope” on page 215

Format for the scope argument

The scope format applies to the scripts in the script library that have the scope argument.

A cell is optional on node, server, and cluster scopes. A node is required on the server scope.

You can delimit the type by using a comma (,) or a colon (:). You can use lower case for the type (cell=, node=, server=, or cluster=.)

The examples in the following table are split on multiple lines for publishing purposes.

Table 212. Examples of the containment path, configuration ID, and type for a particular scope. The scope can be Cell, Node, Server, or Cluster.

Scope	Containment path	Configuration ID	Type
Cell	/Cell:*myCell*/	*myCell(cells/myCell|cell.xml#Cell_1)*	Cell=*myCell* or cell=*myCell*

Table 212. Examples of the containment path, configuration ID, and type for a particular scope (continued). The scope can be Cell, Node, Server, or Cluster.

Scope	Containment path	Configuration ID	Type
Node	/Cell:myCell/Node:*myNode*/ or /Node:*myNode*/	*myNode(cells/myCell /nodes/myNode| node.xml#Node_1)*	Cell=*myCell*, Node=*myNode* or Cell=*myCell*: Node=*myNode* or cell=*myCell*, node=*myNode*
Server	/Cell:*myCell*/Node: *myNode*/ Server:*myServer*/ or /Node:*myNode*/Server: *myServer*/	*myServer(cells /myCell/ nodes/myNode/ servers/myServer| server.xml#Server_1)*	Cell=*myCell*, Node=*myNode*, Server=*myServer* or Node=*myNode*: Server=*myServer* or cell=*myCell*, Node=*myNode*, Server=*myServer*
Cluster	/Cell:*myCell*/ ServerCluster: *myCluster*/ or /ServerCluster: *myCluster*/	*myCluster(cells /myCell/clusters/ myCluster| cluster.xml #ServerCluster_1)*	Cell=*myCell*, Cluster=*myCluster* or Cell=*myCell*: Cluster=*myCluster* or cell=*myCell*, Cluster=*myCluster*

createGenericJMSConnectionFactory

This script creates a new generic JMS connection factory in your configuration. The script returns the configuration ID of the created JMS connection factory in the respective cell.

To run the script, specify the node, server, JMS provider name, name of the new connection factory, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 213. Arguments for the createGenericJMSConnectionFactory script. Run the script to create a generic JMS connection factory.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
connFactoryName	Specifies the name to assign to the new connection factory
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Table 213. Arguments for the createGenericJMSConnectionFactory script (continued). Run the script to create a generic JMS connection factory.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 214. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`

Table 214. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Table 215. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	['type', 'TOPIC']

Syntax

```
AdminJMS.createGenericJMSConnectionFactory(nodeName,
 serverName, jmsProvider, connFactoryName, jndiName,
 extJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSConnectionFactory("myNode", "myServer",
 "JMSTest", "JMSCFTest", "jmsjndi", "extjmsjndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSConnectionFactory("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My Generic JMSConnection Factory", "JNDIName", "extJNDIName", "XAEnabled=false,
authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
description='my JMS Connection Factory',
diagnoseConnectionUsage=false, logMissingTransactionContext=true, manageCachedHandles=true,
providerType=myJMSProviderType, type=TOPIC, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSConnectionFactory("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My Generic JMSConnection Factory",
"JNDIName", "extJNDIName", [['XAEnabled', 'false'], ['authDataAlias', 'myAuthDataAlias'],
['authMechanismPreference', 'BASIC_PASSWORD'], ['category', 'myCategory'],
['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'],
['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeCreationInterval', '10'],
['surgeThreshold', '10'], ['testConnection', 'true'], ['testConnectionInterval', '10'],
['unusedTimeout', '10000']]], ['description', 'My description'],
['diagnoseConnectionUsage', 'false'], ['logMissingTransactionContext', 'true'],
['manageCachedHandles', 'true'], ['mapping', [['authDataAlias', 'authDataAliasValue'],
['mappingConfigAlias', 'mappingConfigAliasValue']]], ['preTestConfig',
[['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'],
['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'],
['value', 50000]]]]], ['providerType', 'myJMSProviderType'], ['sessionPool',
[['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10],
['maxConnections', '12'], ['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'],
['surgeCreationInterval', '10'], ['surgeThreshold', '10'],
['testConnection', 'true'], ['testConnectionInterval', '10'], [
'unusedTimeout', '10000']]], ['type', 'TOPIC'], ['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createGenericJMSConnectionFactoryUsingTemplate

This script uses a template to create a generic JMS connection factory in your configuration. The script returns the configuration ID of the created JMS connection factory using a template in the respective cell.

To run the script, specify the node, server, JMS provider name, template ID, connection factory name, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 216. Arguments for the createGenericJMSConnectionFactoryUsingTemplate script. Run the script to create a generic JMS connection factory.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
templateID	Specifies the configuration ID of the template to use.
connFactoryName	Specifies the name to assign to the new connection factory
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 217. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`

Table 217. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myJMSProviderType']`
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]`
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`['xaRecoveryAuthAlias', 'myCellManager01/a1']`

Table 218. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'TOPIC']`

Syntax

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate(nodeName,
 serverName, jmsProvider, templateID,
 connFactoryName, jndiName, extJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "Generic QueueConnectionFactory for
 Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)",
"JMSCFTest", "jmsjndi", "extjmsjndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Generic QueueConnectionFactory for Windows
(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)",
"My Generic JMSConnection Factory", "JNDIName", "extJNDIName",  "XAEnabled=false,
```

```
authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
description='my JMS Connection Factory using template',
diagnoseConnectionUsage=false, logMissingTransactionContext=true, manageCachedHandles=true,
providerType=myJMSProviderType, type=TOPIC, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Generic QueueConnectionFactory for Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)",
"My Generic JMSConnection Factory", "JNDIName", "extJNDIName", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeCreationInterval', '10'],
['surgeThreshold', '10'], ['testConnection', 'true'], ['testConnectionInterval', '10'],
['unusedTimeout', '10000']]], ['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myJMSProviderType'], ['sessionPool', [['agedTimeout', '100'],
['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeCreationInterval', '10'], ['surgeThreshold', '10'],
['testConnection', 'true'], ['testConnectionInterval', '10'], ['unusedTimeout', '10000']]],
['type', 'TOPIC'], ['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createGenericJMSConnectionFactoryAtScope

This script creates a new generic JMS connection factory in your configuration at the scope that you specify. The script returns the configuration ID of the created JMS connection factory in the respective cell.

To run the script, specify the scope, JMS provider name, name of the new connection factory, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 219. Arguments for the createGenericJMSConnectionFactoryAtScope script. Run the script to create a generic JMS connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider.
connFactoryName	Specifies the name to assign to the new connection factory
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form `jms/Name`, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 220. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`

Table 220. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`

Table 220. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Table 221. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies QUEUE for queues, TOPIC for topics, and UNIFIED for both queues and topics. Do not put either of the values in quotes for the string format of the command.	['type', 'TOPIC']

Syntax

```
AdminJMS.createGenericJMSConnectionFactoryAtScope(scope,
 jmsProvider, connFactoryName, jndiName,
 extJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSConnectionFactoryAtScope("myScope", "JMSTest", "JMSCFTest", "jmsjndi",
"extjmsjndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSConnectionFactoryAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My Generic JMSConnection Factory", "JNDIName", "extJNDIName",
"XAEnabled=false, authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD,
category=myCategory, diagnoseConnectionUsage=false, logMissingTransactionContext=true,
description='my JMS Connection Factory at scope',
manageCachedHandles=true, providerType=myJMSProviderType, type=TOPIC,
xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSConnectionFactoryAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My Generic JMSConnection Factory", "JNDIName", "extJNDIName", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', '100'],
['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10], ['maxConnections', '12'],
['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
```

```
['reapTime', '10000'], ['surgeCreationInterval', '10'], ['surgeThreshold', '10'],
['testConnection', 'true'], ['testConnectionInterval', '10'], ['unusedTimeout', '10000']]],
['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'],
['retryLimit', '4']]], ['properties', [[['description', 'My description'], ['name', 'myName'],
['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]], ['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']],
[['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myJMSProviderType'], ['sessionPool', [['agedTimeout', '100'],
['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10], ['maxConnections', '12'],
['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeCreationInterval', '10'], ['surgeThreshold', '10'],
['testConnection', 'true'], ['testConnectionInterval', '10'], ['unusedTimeout', '10000']]],
['type', 'TOPIC'], ['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createGenericJMSConnectionFactoryUsingTemplateAtScope

This script uses a template to create a generic JMS connection factory in your configuration at the scope that you specify. The script returns the configuration ID of the created JMS connection factory using a template in the respective cell.

To run the script, specify the scope, JMS provider name, template ID, connection factory name, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 222. Arguments for the createGenericJMSConnectionFactoryUsingTemplateAtScope script. Run the script to create a generic JMS connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider.
templateID	Specifies the configuration ID of the template to use.
connFactoryName	Specifies the name to assign to the new connection factory
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 223. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`

Table 223. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myJMSProviderType']`

Table 223. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Table 224. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	['type', 'TOPIC']

Syntax

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplateAtScope(scope,
  jmsProvider, templateID,
 connFactoryName, jndiName, extJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplateAtScope("myScope",
 "JMSTest", "Generic QueueConnectionFactory for
 Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)",
"JMSCFTest", "jmsjndi", "extjmsjndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1",
"Generic QueueConnectionFactory for Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)",
"My Generic JMSConnection Factory", "JNDIName", "extJNDIName", "XAEnabled=false,
authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
diagnoseConnectionUsage=false, logMissingTransactionContext=true, manageCachedHandles=true,
description='my JMS Connection Factory using a template and scope',
providerType=myJMSProviderType, type=TOPIC, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSConnectionFactoryUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1",
"Generic QueueConnectionFactory for Windows(templates/system|JMS-resource-provider-templates.xml#GenericJMSConnectionFactory_1)",
"My Generic JMSConnection Factory", "JNDIName", "extJNDIName",
[['XAEnabled', 'false'], ['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'], ['reapTime', '10000'],
['surgeCreationInterval', '10'], ['surgeThreshold', '10'], ['testConnection', 'true'],
['testConnectionInterval', '10'], ['unusedTimeout', '10000']]], ['description', 'My description'],
['diagnoseConnectionUsage', 'false'], ['logMissingTransactionContext', 'true'],
['manageCachedHandles', 'true'], ['mapping', [['authDataAlias', 'authDataAliasValue'],
['mappingConfigAlias', 'mappingConfigAliasValue']]], ['preTestConfig', [['preTestConnection', 'true'],
```

```
['retryInterval', '12343'], ['retryLimit', '4']]], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['propertySet', [['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]]], ['providerType', 'myJMSProviderType'],
['sessionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10],
['maxConnections', '12'], ['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'], ['name', 'myName'], ['required', 'false'], ['validationExpression', ''],
['value', 'myValue']]]], ['purgePolicy', 'EntirePool'], ['reapTime', '10000'],
['surgeCreationInterval', '10'], ['surgeThreshold', '10'], ['testConnection', 'true'],
['testConnectionInterval', '10'], ['unusedTimeout', '10000']]], ['type', 'TOPIC'],
['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createGenericJMSDestination

This script creates a generic JMS destination in your configuration. The script returns the configuration ID of the created JMS destination in the respective cell.

To run the script, specify the node, server, JMS provider name, JMS destination name, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 225. Arguments for the createGenericJMSDestination script. Run the script to create a generic JMS destination.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
genericJMSDestination	Specifies the name to assign to the new generic JMS destination.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 226. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType] [type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`

Table 226. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'QUEUE']`

Syntax

```
AdminJMS.createGenericJMSDestination(nodeName,
 serverName, jmsProvider, genericJMSDestination,
 jndiName, extJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSDestination("myNode", "myServer", "JMSTest",
 "JMSDest", "destjndi", "extDestJndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSDestination("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My JMSDestination", "JNDIName", "extJNDIName", "category=myCategory, description='My description',
providerType=myProviderType, type=QUEUE")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSDestination("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My JMSDestination", "JNDIName", "extJNDIName", [['category', 'myCategory'], ['description',
'My description'], ['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['type', 'QUEUE']] )
```

createGenericJMSDestinationUsingTemplate

This script uses a template to create a generic JMS destination in your configuration. The script returns the configuration ID of the created JMS destination in the respective cell.

To run the script, specify the node, server, JMS provider name, template ID, generic JMS destination name, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 227. Arguments for the createGenericJMSDestinationUsingTemplate script. Run the script to create a generic JMS destination.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
templateID	Specifies the configuration ID of the template to use.
genericJMSDestination	Specifies the name to assign to the new generic JMS destination.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 228. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType] [type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'QUEUE']`

Syntax

```
AdminJMS.createGenericJMSDestinationUsingTemplate(nodeName,
 serverName, jmsProvider, templateID,
 genericJMSDestination, jndiName, extJndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSDestinationUsingTemplate("myNode", "myServer",
 "JMSTest",
 "Example.JMS.Generic.Win.Topic(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_2)",
 "JMSDest", "destjndi", "extDestJndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSDestinationUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example.JMS.Generic.Win.Queue(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_1)",
"My JMSDestination", "JNDIName", "extJNDIName","category=myCategory, description='My description',
providerType=myProviderType, type=QUEUE ")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSDestinationUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example.JMS.Generic.Win.Queue(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_1)",
"My JMSDestination", "JNDIName", "extJNDIName", [['category', 'myCategory'],
['description', 'My description'], ['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value','localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myProviderType'], ['type', 'QUEUE']] )
```

createGenericJMSDestinationAtScope

This script creates a generic JMS destination in your configuration at the scope that you specify. The script returns the configuration ID of the created JMS destination in the respective cell.

To run the script, specify the scope, JMS provider name, JMS destination name, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 229. Arguments for the createGenericJMSDestinationAtScope script. Run the script to create a generic JMS destination.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
genericJMSDestination	Specifies the name to assign to the new generic JMS destination.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 230. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType] [type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'QUEUE']`

Syntax

```
AdminJMS.createGenericJMSDestinationAtScope(scope,
 jmsProvider, genericJMSDestination,
 jndiName, extJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSDestinationAtScope("myScope", "JMSTest",
 "JMSDest", "destjndi", "extDestJndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSDestinationAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My JMSDestination", "JNDIName", "extJNDIName", "category=myCategory,
description='My description', providerType=myProviderType, type=QUEUE")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSDestinationAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My JMSDestination", "JNDIName", "extJNDIName", [['category', 'myCategory'],
['description', 'My description'], ['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'],
['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value',50000]]]]], ['providerType', 'myProviderType'], ['type', 'QUEUE']] )
```

createGenericJMSDestinationUsingTemplateAtScope

This script uses a template to create a generic JMS destination in your configuration at the scope you specify. The script returns the configuration ID of the created JMS destination in the respective cell.

To run the script, specify the scope, JMS provider name, template ID, generic JMS destination name, JNDI name, and external JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 231. Arguments for the createGenericJMSDestinationUsingTemplateAtScope script. Run the script to create a generic JMS destination.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider.
templateID	Specifies the configuration ID of the template to use.
genericJMSDestination	Specifies the name to assign to the new generic JMS destination.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
extJndiName	Specifies the JNDI name that is used to bind the queue into the application server name space. As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource. This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 232. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType] [type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`

Table 232. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'QUEUE']`

Syntax

```
AdminJMS.createGenericJMSDestinationUsingTemplateAtScope(scope,
 jmsProvider, templateID,
 genericJMSDestination, jndiName, extJndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createGenericJMSDestinationUsingTemplateAtScope("myScope",
 "JMSTest",
 "Example.JMS.Generic.Win.Topic(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_2)",
 "JMSDest", "destjndi", "extDestJndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createGenericJMSDestinationUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example.JMS.Generic.Win.Queue
(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_1)",
"My JMSDestination", "JNDIName", "extJNDIName", "category=myCategory,
description='My description', providerType=myProviderType, type=QUEUE")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createGenericJMSDestinationUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example.JMS.Generic.Win.Queue
(templates/system|JMS-resource-provider-templates.xml#GenericJMSDestination_1)",
"My JMSDestination", "JNDIName", "extJNDIName", [['category', 'myCategory'], ['description', 'My description'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myProviderType'], ['type', 'QUEUE']] )
```

createJMSProvider

This script creates a JMS provider in your configuration. The script returns the configuration ID of the created JMS provider in the respective cell.

To run the script, specify the node, server, JMS provider name, external initial contextual factory name, and external provider URL arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 233. Arguments for the createJMSProvider script. Run the script to create a JMS provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name to assign to the new JMS provider.
extContextFactory	Specifies the Java class name of the initial context factory for the JMS provider.
extProviderURL	Specifies the JMS provider URL for external JNDI lookups.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 234. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`'-classpath', '${EXTERNAL_JMSPROVIDER_CLASSPATH}/extJms.jar'`
description	Specifies a description of the JMS Provider.	`['description', 'My description']`
isolatedClassLoader	If set to true, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'true']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`'-nativePath', '${EXTERNAL_JMSPROVIDER_NATIVEPATH}'`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JMS provider type that this JMS provider uses.	`['providerType', 'myJMSProviderType']`
supportsASF	If set to `true`, specifies that the JMS provider supports Application Server Facilities (ASF), which provides concurrency and transactional support for applications.	`['supportsASF', 'true']`

Syntax

```
AdminJMS.createJMSProvider(nodeName, serverName,
 jmsProvider, extContextFactory, extProviderURL,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createJMSProvider("myNode", "myServer", "JMSTest1",
 "extInitCF", "extPURL")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createJMSProvider("IBM-F4A849C57A0Node01", "server1", "MyJMSProvider", "extInitCF", "extPURL",
"classpath='${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar',
description='My JMSProvider description',isolatedClassLoader=true,
nativepath=${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}, providerType=myJMSProviderType, supportsASF=true")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createJMSProvider("IBM-F4A849C57A0Node01", "server1", "MyJMSProvider", "extInitCF", "extPURL",
[['classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'], ['description', 'My JMSProvider description'],
['isolatedClassLoader', 'true'], ['nativepath', '${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]], ['providerType', 'myJMSProviderType'],
['supportsASF', 'true']] )
```

createJMSProviderUsingTemplate

This script uses a template to create a JMS provider in your configuration. The script returns the configuration ID of the created JMS provider using a template in the respective cell.

To run the script, specify the node, server, configuration ID of the JMS provider template, name to assign to the new JMS provider, external initial context factory, and external provider URL arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 235. Arguments for the createJMSProviderUsingTemplate script. Run the script to create a JMS provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
templateID	Specifies the configuration ID of the JMS provider template to use.
jmsProvider	Specifies the name to assign to the new JMS provider.
extContextFactory	Specifies the Java class name of the initial context factory for the JMS provider.
extProviderURL	Specifies the JMS provider URL for external JNDI lookups.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 236. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`'-classpath', '${EXTERNAL_JMSPROVIDER_CLASSPATH}/extJms.jar'`
description	Specifies a description of the JMS Provider.	`['description', 'My description']`
isolatedClassLoader	If set to true, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'true']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`'-nativePath', '${EXTERNAL_JMSPROVIDER_NATIVEPATH}'`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JMS provider type that this JMS provider uses.	`['providerType', 'myJMSProviderType']`
supportsASF	If set to `true`, specifies that the JMS provider supports Application Server Facilities (ASF), which provides concurrency and transactional support for applications.	`['supportsASF', 'true']`

Syntax

```
AdminJMS.createJMSProviderUsingTemplate(nodeName,
 serverName, templateID, jmsProvider,
 extContextFactory, extProviderURL, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createJMSProviderUsingTemplate("myNode", "myServer", "WebSphere JMS
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#builtin_jmsprovider)",
 "JMSTest", "extInitCF", "extPURL")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createJMSProviderUsingTemplate("IBM-F4A849C57A0Node01", "server1",
"WebSphere JMSProvider(templates/servertypes/APPLICATION_SERVER/servers/default|resources.xml#builtin_jmsprovider)",
"MyJMSProvider", "extInitCF", "extPURL",
"classpath='${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar', description='My JMSProvider description',
isolatedClassLoader=true, nativepath=${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH},
providerType=myJMSProviderType,supportsASF=true")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createJMSProviderUsingTemplate("IBM-F4A849C57A0Node01", "server1",
"WebSphere JMSProvider(templates/servertypes/APPLICATION_SERVER/servers/default|resources.xml#builtin_jmsprovider)",
"MyJMSProvider", "extInitCF", "extPURL",
[['classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'],
['description', 'My JMSProvider description'], ['isolatedClassLoader', 'true'],
['nativepath', '${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']],
[['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myJMSProviderType'], ['supportsASF', 'true']])
```

createJMSProviderAtScope

This script creates a JMS provider in your configuration at the scope that you specify. The script returns the configuration ID of the created JMS provider in the respective cell.

To run the script, specify the scope, JMS provider name, external initial contextual factory name, and external provider URL arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 237. Arguments for the createJMSProviderAtScope script. Run the script to create a JMS provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name to assign to the new JMS provider.
extContextFactory	Specifies the Java class name of the initial context factory for the JMS provider.
extProviderURL	Specifies the JMS provider URL for external JNDI lookups.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 238. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`'-classpath', '${EXTERNAL_JMSPROVIDER_CLASSPATH}/extJms.jar'`
description	Specifies a description of the JMS Provider.	`['description', 'My description']`

Table 238. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
isolatedClassLoader	If set to true, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'true']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`'-nativePath', '${EXTERNAL_JMSPROVIDER_NATIVEPATH}'`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JMS provider type that this JMS provider uses.	`['providerType', 'myJMSProviderType']`
supportsASF	If set to `true`, specifies that the JMS provider supports Application Server Facilities (ASF), which provides concurrency and transactional support for applications.	`['supportsASF', 'true']`

Syntax

```
AdminJMS.createJMSProviderAtScope(scope,
 jmsProvider, extContextFactory, extProviderURL,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createJMSProvider("myScope", "JMSTest1",
 "extInitCF", "extPURL")
```

The following example script includes optional attributes in a string format:

```
AdminJMS. createJMSProviderAtScope ("Cell= IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"MyJMSProvider", "extInitCF", "extPURL",
"classpath='${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar', description='My JMSProvider description',
isolatedClassLoader=true, nativepath=${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH},
providerType=myJMSProviderType,supportsASF=true")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createJMSProviderAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"MyJMSProvider", "extInitCF", "extPURL", [['classpath',
'${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'], ['description', 'My JMSProvider description'],
['isolatedClassLoader', 'true'], ['nativepath', '${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'],
['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]], ['providerType', 'myJMSProviderType'],
['supportsASF', 'true']] )
```

createJMSProviderUsingTemplateAtScope

This script uses a template to create a JMS provider at the scope that you specify. The script returns the configuration ID of the created JMS provider using a template in the respective cell.

To run the script, specify the scope, configuration ID of the JMS provider template, name to assign to the new JMS provider, external initial context factory, and external provider URL arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 239. Arguments for the createJMSProviderUsingTemplateAtScope script. Run the script to create a JMS provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
templateID	Specifies the configuration ID of the JMS provider template to use.
jmsProvider	Specifies the name to assign to the new JMS provider.
extContextFactory	Specifies the Java class name of the initial context factory for the JMS provider.
extProviderURL	Specifies the JMS provider URL for external JNDI lookups.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 240. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.	`'-classpath', '${EXTERNAL_JMSPROVIDER_CLASSPATH}/extJms.jar'`
description	Specifies a description of the JMS Provider.	`['description', 'My description']`
isolatedClassLoader	If set to true, specifies that the resource provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.	`['isolatedClassLoader', 'true']`
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Native path entries are separated by a semicolon (;).	`'-nativePath', '${EXTERNAL_JMSPROVIDER_NATIVEPATH}'`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]].. [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
providerType	Specifies the JMS provider type that this JMS provider uses.	`['providerType', 'myJMSProviderType']`
supportsASF	If set to `true`, specifies that the JMS provider supports Application Server Facilities (ASF), which provides concurrency and transactional support for applications.	`['supportsASF', 'true']`

Syntax

```
AdminJMS.createJMSProviderUsingTemplateAtScope(scope,
 templateID, jmsProvider,
 extContextFactory, extProviderURL, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createJMSProviderUsingTemplateAtScope("myScope", "WebSphere JMS
 Provider(templates/servertypes/APPLICATION_SERVER/servers/DeveloperServer|resources.xml#builtin_jmsprovider)",
 "JMSTest", "extInitCF", "extPURL")
```

The following example script includes optional attributes in a string format:

```
AdminJMS. createJMSProviderUsingTemplateAtScope
("Cell= IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"WebSphere JMSProvider(templates/servertypes/APPLICATION_SERVER/servers/default|resources.xml#builtin_jmsprovider)",
"MyJMSProvider", "extInitCF", "extPURL",
"classpath='${DB2_JCC_DRIVER_PATH}/db2jcc4.jar;${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar', description='My JMSProvider description',
isolatedClassLoader=true, nativepath=${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH},
providerType=myJMSProviderType,supportsASF=true")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createJMSProviderUsingTemplateAtScope("Cell=IBM-F4A849C57A0Cell01, Node=IBM-F4A849C57A0Node01",
"WebSphere JMS Provider(templates/servertypes/APPLICATION_SERVER/servers/default|resources.xml#builtin_jmsprovider)",
"MyJMSProvider", "extInitCF", "extPURL",
[['classpath', '${DB2_JCC_DRIVER_PATH}/db2jcc4.jar; ${UNIVERSAL_JDBC_DRIVER}/db2jcc_license_cu.jar;
${DB2_JCC_DRIVER_PATH}/db2jcc_license_cisuz.jar'], ['description', 'My JMSProvider description'],
['isolatedClassLoader', 'true'], ['nativepath', '${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'],
['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value',50000]]]]], ['providerType', 'myJMSProviderType'],
['supportsASF', 'true']] )
```

createWASQueue

This script creates a WebSphere Application server queue in your configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue in the respective cell.

To run the script, specify the node, server, JMS provider name, name to assign to the queue, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 241. Arguments for the createWASQueue script. Run the script to create a queue.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
queueName	Specifies the name to assign to the new queue.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 242. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`

Table 242. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are APPLICATION_DEFINE, SPECIFIED, and UNLIMITED. Do not put either of the values in quotes for the string format of the command.	['expiry', 'SPECIFIED']
persistence	Specifies the level of persistence. Valid values are APPLICATION_DEFINED, NONPERSISTENT, and PERSISTENT.	['persistence', 'NONPERSISTENT']
priority	Specifies the level of priority. Valid values are APPLICATION_DEFINED and SPECIFIED.	['priority', 'SPECIFIED']
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myProviderType']
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	['specifiedExpiry', '1000']
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	['specifiedPriority', '0']

Syntax

```
AdminJMS.createWASQueue(nodeName, serverName,
 jmsProvider, queueName, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueue("myNode", "myServer", "JMSTest",
 "WASQueueTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueue("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example.JMS.WAS.Q1(templates/system|JMS-resource-provider-templates.xml#WASQueue_1)",
"My WASQueue Name", "JNDIName", "category =myCategory, description='My description',
expiry=SPECIFIED, persistence=NONPERSISTENT, priority=SPECIFIED, specifiedExpiry=1000,
specifiedPriority=0")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueue("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1", "My WASQueue Name",
"JNDIName", [['category ', 'myCategory'], ['description', 'My description'], ['expiry', 'SPECIFIED'],
['persistence', 'NONPERSISTENT'], ['priority', 'SPECIFIED'], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createWASQueueUsingTemplate

This script uses a template to create a WebSphere Application Server queue in your configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue using a template in the respective cell.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the queue, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 243. Arguments for the createWASQueueUsingTemplate script. Run the script to create a queue.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server queue template to use.
queueName	Specifies the name to assign to the new queue.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 244. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[[name databaseName][type string]` `[value mys]]` `[[name driverType][type integer][value 4]]` `[[name serverName][type string]` `[value localhost]]` `[[name portNumber][type integer]` `[value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`

Table 244. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	`['specifiedExpiry', '1000']`
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	`['specifiedPriority', '0']`

Syntax

```
AdminJMS.createWASQueueUsingTemplate(nodeName,
 serverName, jmsProvider, templateID, queueName,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueUsingTemplate("myNode", "myServer", "JMSTest",
 "WASQueueTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example.JMS.WAS.Q1(templates/system|JMS-resource-provider-templates.xml#WASQueue_1)", "My WASQueue Name",
"JNDIName", "category =myCategory, description='My description', expiry=SPECIFIED,
providerType=myProviderType, specifiedExpiry=1000, specifiedPriority=0")
```

The following example script includes optional attributes in a list format:

```
Usage: AdminJMS.createWASQueueUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example.JMS.WAS.Q1(templates/system|JMS-resource-provider-templates.xml#WASQueue_1)", "My WASQueue Name",
"JNDIName", [['category ', 'myCategory'], ['description', 'My description'], ['expiry', 'SPECIFIED'],
['persistence', 'NONPERSISTENT'], ['priority', 'SPECIFIED'], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createWASQueueAtScope

This script creates a WebSphere Application Server queue in your configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue in the respective cell.

To run the script, specify the scope, JMS provider name, name to assign to the queue, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 245. Arguments for the createWASQueueAtScope script. Run the script to create a queue.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider of interest.
queueName	Specifies the name to assign to the new queue.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 246. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[[name databaseName][type string]` `[value mys]]` `[[name driverType][type integer][value 4]]` `[[name serverName][type string]` `[value localhost]]` `[[name portNumber][type integer]` `[value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	`['specifiedExpiry', '1000']`
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	`['specifiedPriority', '0']`

Syntax

```
AdminJMS.createWASQueueAtScope(scope,
 jmsProvider, queueName, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueAtScope("scope", "JMSTest",
 "WASQueueTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASQueue Name", "JNDIName", "category =myCategory, description='My description',
expiry=SPECIFIED, providerType=myProviderType, specifiedExpiry=1000, specifiedPriority=0")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueueAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASQueue Name", "JNDIName", [['category ', 'myCategory'], ['description', 'My description'],
['expiry', 'SPECIFIED'], ['persistence', 'NONPERSISTENT'], ['priority', 'SPECIFIED'], ['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]], [
'providerType', 'myProviderType'], ['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createWASQueueUsingTemplateAtScope

This script uses a template to create a WebSphere Application Server queue in your configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue using a template in the respective cell.

To run the script, specify the scope, JMS provider name, configuration ID of the template, name to assign to the queue, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 247. Arguments for the createWASQueueUsingTemplateAtScope script. Run the script to create a queue.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server queue template to use.
queueName	Specifies the name to assign to the new queue.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 248. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[[name databaseName][type string]` `[value mys]]` `[[name driverType][type integer][value 4]]` `[[name serverName][type string]` `[value localhost]]` `[[name portNumber][type integer]` `[value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`

Table 248. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	`['specifiedExpiry', '1000']`
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	`['specifiedPriority', '0']`

Syntax

```
AdminJMS.createWASQueueUsingTemplateAtScope(scope,
 jmsProvider, templateID, queueName,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueUsingTemplateAtScope("myScope", "JMSTest",
 "WASQueueTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueUsingTemplateAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example.JMS.WAS.Q1(templates/system|JMS-resource-provider-templates.xml#WASQueue_1)",
"My WASQueue Name", "JNDIName", "category =myCategory, description='My description', expiry=SPECIFIED,
providerType=myProviderType, specifiedExpiry=1000, specifiedPriority=0")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueueUsingTemplateAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example.JMS.WAS.Q1(templates/system|JMS-resource-provider-templates.xml#WASQueue_1)",
"My WASQueue Name", "JNDIName", [['category ', 'myCategory'], ['description', 'My description'], ['expiry', 'SPECIFIED'],
['persistence', 'NONPERSISTENT'], ['priority', 'SPECIFIED'], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createSIBJMSQueue

This script creates a JMS queue in your configuration at the scope that you specify. The script returns the configuration ID of the created SIB JMS queue.

To run the script, specify the scope, JMS queue name, name to assign to the queue, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 249. Arguments for the createSIBJMSQueue script. Run the script to create a JMS queue.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the name of the JMS queue of interest.
queueName	Specifies the name of the service integration bus destination to which the JMS queue maps.
jndiName	Specifies the JNDI name that the system to bind the queue into the application server namespace.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 250. Optional attributes. Additional attributes for the script.

Attributes	Description
description	Specifies a description of the SIB JMS queue.
DeliveryMode	Specifies the delivery mode for messages. Valid values are `Application`, `NonPersistent`, and `Persistent`.

Table 250. Optional attributes (continued). Additional attributes for the script.

Attributes	Description
timeToLive	Specifies the time in milliseconds before a message expires.
priority	Specifies the priority for messages. Valid values are integers from 0 to 9, inclusive.
readAhead	Specifies the read-ahead value. Valid values are `Default`, `AlwaysOn`, and `AlwaysOff`.
busName	Specifies the name of the bus on which the queue resides.
ScopeToLocalQP	Specifies whether the SIB queue destination is scoped to a local queue point when addressed using this JMS queue.
ProducerBind	Specifies how JMS producers bind to queue points of the clustered queue. A value of `TRUE` indicates that a queue point is chosen when the session is opened and never changes. A value of `FALSE` indicates that a queue point is chosen every time a message is sent.
ProducerPreferLocal	Specifies whether queue points local to the producer are used.
GatherMessages	Specifies whether JMS consumers and browsers are given messages from any queue points, rather than from the single queue point to which they are attached.

Syntax

```
AdminJMS.createSIBJMSQueue(scope,
 name, queueName,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSQueue("myScope", "myName",
 "SIBJMSQueueTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createSIBJMSQueue
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName7", "myQueueName7", "readAhead=AlwaysOff,timeToLive=9")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createSIBJMSQueue
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName7", "myQueueName7", [['readAhead', 'AlwaysOff'], ['timeToLive', '9']])
```

createWMQQueue

This script creates queue type destination for the WebSphere MQ messaging provider at the scope that you specify. The script returns the configuration ID of the created WebSphere MQ queue.

To run the script, specify the scope, name of the queue type destination, name to assign to the queue, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 251. Arguments for the createWMQQueue script. Run the script to create a queue type destination.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the WebSphere MQ messaging provider.
name	Specifies the name of the WebSphere MQ messaging provider queue type destination.
queueName	Specifies the name of the WebSphere MQ queue to store messages for the queue type destination definition of the WebSphere MQ messaging provider.
jndiName	Specifies the name used to bind this object into WebSphere Application Server JNDI.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 252. Optional attributes. Additional attributes available for the script.

Attributes	Description
ccsid	Specifies the coded character set identifier.
decimalEncoding	Specifies the decimal encoding setting for this queue. Valid values are `Normal` and `Reversed`.
description	Specifies an administrative description to associate with this WebSphere MQ JMS queue type destination.
expiry	Specifies the amount of time after which messages, sent to this destination, expire and are dealt with based on their disposition options. Valid values are `APP`, `UNLIM`, or any positive integer.
floatingPointEncoding	Specifies the floating point encoding setting for this queue. Valid values are `IEEENormal`, `IEEEReversed` and `z/OS`.
integerEncoding	Specifies the integer encoding setting for this queue. Valid values are `Normal` and `Reversed`.
persistence	Specifies the level of persistence used to store messages sent to this destination. Valid values are `APP`, `QDEF`, `PERS`, `NON` or `HIGH`.
priority	Specifies the priority level to assign to messages sent to this destination. Valid values are `APP`, `QDEF`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`.
qmgr	Specifies the queue manager on which the WebSphere MQ queue resides.
readAheadClose	Specifies the behavior that occurs when closing a message consumer that is receiving messages asynchronously. The message consumer uses a message listener from a destination that has the readAhead parameter set to `true`. When a value of `DELIVERALL` is specified, the close method invocation waits until all read-ahead messages are delivered to the consumer before closing it. When a value of `DELIVERCURRENT` is specified, then the close() method only waits for any in-progress delivery to end before closing the consumer. Valid values are `DELIVERALL` and `DELIVERCURRENT`.
readAhead	Specifies whether messages for non-persistent consumers can be read ahead and cached. Valid values are `YES`, `NO` or `QDEF`.
sendAsync	Specifies whether messages can be sent to this destination without requiring that the queue manager acknowledges they have arrived. Valid values are `YES`, `NO` or `QDEF`.
useRFH2	Specifies whether an RFH version 2 header is appended to messages sent to this destination. Valid values are `true` or `false`.
useNativeEncoding	Specifies the native encoding settings for this queue.

Syntax

```
AdminJMS.createWMQQueue(scope,
 name, queueName,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWMQQueue("myScope", "myName",
 "WMQQueueTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQQueue
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName8", "myQueueName8", "readAhead=YES,description=myDescription")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWMQQueue
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName8", "myQueueName8", [['readAhead', 'YES'], ['description', 'myDescription']])
```

createWASQueueConnectionFactory

This script creates a WebSphere Application Server queue connection factory in your configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue connection factory in the respective cell.

To run the script, specify the node, server, JMS provider name, name to assign to the queue connection factory, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 253. Arguments for the createWASQueueConnectionFactory script. Run the script to create a queue connection factory.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
queueConnFactoryName	Specifies the name to assign to the new WebSphere Application Server queue connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 254. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`

Table 254. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name, nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Syntax

```
AdminJMS.createWASQueueConnectionFactory(nodeName,
 serverName, jmsProvider, queueConnFactoryName,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueConnectionFactory("myNode", "myServer",
 "JMSTest", "queueCFTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueConnectionFactory("IBM-F4A849C57A0Node01", "server1",
"My JMS Provider Name1", "My WASQueue Name", "JNDIName", "XAEnabled=false,
authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD,  category=myCategory,
description='My description', diagnoseConnectionUsage=false, logMissingTransactionContext=true,
manageCachedHandles=true, providerType=myProviderType, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueueConnectionFactory("IBM-F4A849C57A0Node01", "server1",
"My JMS Provider Name1", "My WASQueue Name", "JNDIName", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ' '], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeCreationInterval', '10'], ['surgeThreshold', '10'],
['testConnection', 'true'], ['testConnectionInterval', '10'], ['unusedTimeout', '10000']]],
['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['sessionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeThreshold', '10'], ['testConnection', 'true']]],
['xaRecoveryAuthAlias', 'myCellManager01/a1']] ))
```

createWASQueueConnectionFactoryUsingTemplate

This script uses a template to create a WebSphere Application Server queue connection factory in your configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue connection factory using a template in the respective cell.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the queue connection factory, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 255. Arguments for the createWASQueueConnectionFactoryUsingTemplate script. Run the script to create a queue connection factory.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server queue connection factory template to use.
queueConnFactoryName	Specifies the name to assign to the new WebSphere Application Server queue connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 256. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`

Table 256. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`

Table 256. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Syntax

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplate(nodeName,
 serverName, jmsProvider, templateID,
 queueConnFactoryName, jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "Example WAS
 QueueConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
 "queueCFTest", "queuecfjndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example WAS QueueConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
"My WASQueue Name", "JNDIName", "XAEnabled=false, authDataAlias=myAuthDataAlias,
authMechanismPreference=BASIC_PASSWORD,  category=myCategory, description='My description',
diagnoseConnectionUsage=false, logMissingTransactionContext=true, manageCachedHandles=true,
providerType=myProviderType, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplate("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"Example WAS QueueConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
"My WASQueue Name", "JNDIName", [['XAEnabled', 'false'],['authDataAlias', 'myAuthDataAlias'],
['authMechanismPreference', 'BASIC_PASSWORD'], ['category', 'myCategory'], ['connectionPool',
[['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10],
['maxConnections', '12'], ['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
```

```
['validationExpression', ' '], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'],['surgeCreationInterval', '10'], ['surgeThreshold', '10'],
['testConnection', 'true'], ['testConnectionInterval', '10'], ['unusedTimeout', '10000']]],
['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myProviderType'], ['sessionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeThreshold', '10'], ['testConnection', 'true']]],
['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createWASQueueConnectionFactoryAtScope

This script creates a WebSphere Application Server queue connection factory in your configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue connection factory in the respective cell.

To run the script, specify the scope, JMS provider name, name to assign to the queue connection factory, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 257. Arguments for the createWASQueueConnectionFactoryAtScope script. Run the script to create a queue connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider of interest.
queueConnFactoryName	Specifies the name to assign to the new WebSphere Application Server queue connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 258. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`

Table 258. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]
description	Specifies a description of the resource adapter.	['description', 'My description']
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	['diagnoseConnectionUsage', 'false']
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	['logMissingTransactionContext', 'true']
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	['manageCachedHandles', 'true']
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']

Table 258. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Syntax

```
AdminJMS.createWASQueueConnectionFactoryAtScope(scope,
 jmsProvider, queueConnFactoryName,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueConnectionFactoryAtScope("myScope",
 "JMSTest", "queueCFTest", "queuejndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueConnectionFactoryAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASQueue Name", "JNDIName", "XAEnabled=false, authDataAlias=myAuthDataAlias,
authMechanismPreference=BASIC_PASSWORD,  category=myCategory, description='My description', diagnoseConnectionUsage=false,
logMissingTransactionContext=true, manageCachedHandles=true, providerType=myProviderType,
 xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueueConnectionFactoryAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASQueue Name", "JNDIName", [['XAEnabled', 'false'], ['authDataAlias', 'myAuthDataAlias'],
['authMechanismPreference', 'BASIC_PASSWORD'], ['category', 'myCategory'], ['connectionPool', [['agedTimeout', '100'],
['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'], ['validationExpression', ' '],
['value', 'myValue']]]], ['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeCreationInterval', '10'],
['surgeThreshold', '10'], ['testConnection', 'true'], ['testConnectionInterval', '10'], ['unusedTimeout', '10000']]],
['description', 'My description'], ['diagnoseConnectionUsage', 'false'], ['logMissingTransactionContext', 'true'],
['manageCachedHandles', 'true'], ['mapping', [['authDataAlias', 'authDataAliasValue'],
['mappingConfigAlias', 'mappingConfigAliasValue']]], ['preTestConfig', [['preTestConnection', 'true'],
['retryInterval', '12343'],
['retryLimit', '4']]], ['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'],
['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]]], ['providerType', 'myProviderType'], ['sessionPool',
[['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10], ['maxConnections', '12'],
['minConnections', '5'], ['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'], ['name', 'myName'],
['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'], ['reapTime', '10000'],
['surgeThreshold', '10'], ['testConnection', 'true']]], ['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createWASQueueConnectionFactoryUsingTemplateAtScope

This script uses a template to create a WebSphere Application Server queue connection factory in your configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version

7.0 cell. The script returns the configuration ID of the created WebSphere Application Server queue connection factory using a template in the respective cell.

To run the script, specify the scope, JMS provider name, configuration ID of the template, name to assign to the queue connection factory, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 259. Arguments for the createWASQueueConnectionFactoryUsingTemplateAtScope script. Run the script to create a queue connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server queue connection factory template to use.
queueConnFactoryName	Specifies the name to assign to the new WebSphere Application Server queue connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 260. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`

Table 260. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name, nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Syntax

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplateAtScope(scope,
 jmsProvider, templateID,
 queueConnFactoryName, jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplateAtScope("myScope",
 "JMSTest", "Example WAS
 QueueConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
 "queueCFTest", "queuecfjndi")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example WAS QueueConnectionFactory
(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
"My WASQueue Name", "JNDIName", "XAEnabled=false, authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD,
category=myCategory, description='My description', diagnoseConnectionUsage=false, logMissingTransactionContext=true,
manageCachedHandles=true, providerType=myProviderType, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASQueueConnectionFactoryUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example WAS QueueConnectionFactory
(templates/system|JMS-resource-provider-templates.xml#WASQueueConnectionFactory_1)",
"My WASQueue Name", "JNDIName", [['XAEnabled', 'false'], ['authDataAlias', 'myAuthDataAlias'],
['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ' '], ['value', 'myValue']]]],
 ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeCreationInterval', '10'], ['surgeThreshold', '10'], ['testConnection', 'true'],
['testConnectionInterval', '10'], ['unusedTimeout', '10000']]], ['description', 'My description'],
 ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext','true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'],
['mappingConfigAlias', 'mappingConfigAliasValue']]], ['preTestConfig', [['preTestConnection', 'true'],
['retryInterval', '12343'],
['retryLimit', '4']]], ['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties', [[['name', 'databaseName'],
['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'],
['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']],
[['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]]], ['providerType', 'myProviderType'], ['sessionPool',
[['agedTimeout', '100'],
['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''],
['value', 'myValue']]]], ['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeThreshold', '10'],
 ['testConnection', 'true']]],
['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createWASTopic

This script creates a WebSphere Application Server topic in your JMS configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server topic in the respective cell.

To run the script, specify the node, server, JMS provider name, name to assign to the topic, JNDI name, and the topic arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 261. Arguments for the createWASTopic script. Run the script to create a topic.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
topicName	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
topic	Specifies the name of the topic (as a qualifier in the topic space) that this topic is to use.

Table 261. Arguments for the createWASTopic script (continued). Run the script to create a topic.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 262. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[[name databaseName][type string]` `[value mys]]` `[[name driverType][type integer][value 4]]` `[[name serverName][type string]` `[value localhost]]` `[[name portNumber][type integer]` `[value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	`['specifiedExpiry', '1000']`
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	`['specifiedPriority', '0']`

Syntax

```
AdminJMS.createWASTopic(nodeName, serverName,
 jmsProvider, topicName, jndiName, topic,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopic("myNode", "myServer", "JMSTest",
 "TopicTest", "topicjndi", "mytopic")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopic("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My WASTopic", "JNDIName", "A WAS Topic"," category=myCategory, description='My description',
expiry=SPECIFIED, persistence=PERSISTENT, priority=APPLICATION_DEFINED")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopic("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1", "My WASTopic",
"JNDIName", "A WAS Topic", [['category', 'myCategory'], ['description', 'My description'],
['expiry', 'SPECIFIED'], ['persistence', 'PERSISTENT'], ['priority', 'APPLICATION_DEFINED'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]], ['providerType', 'myProviderType'],
['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createWASTopicUsingTemplate

This script uses a template to create a WebSphere Application Server topic in your JMS configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere topic in the respective cell.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the topic, JNDI name, and the topic arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 263. Arguments for the createWASTopicUsingTemplate script. Run the script to create a topic.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server topic template to use.
topicName	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
topic	Specifies the name of the topic (as a qualifier in the topic space) that this topic is to use.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 264. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`

Table 264. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	`['specifiedExpiry', '1000']`
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	`['specifiedPriority', '0']`

Syntax

```
AdminJMS.createWASTopicUsingTemplate(nodeName,
 serverName, jmsProvider, templateID, topicName,
 jndiName, topic, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopicUsingTemplate("myNode", "myServer", "JMSTest",
 "Example.JMS.WAS.T1(templates/system|JMS-resource-provider-templates.xml#WASTopic_1)",
"TopicTest",
 "topicjndi", "mytopic")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopic("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My WASTopic", "JNDIName", "A WAS Topic"," category=myCategory, description='My description',
expiry=SPECIFIED, persistence=PERSISTENT, priority=APPLICATION_DEFINED")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopic("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1", "My WASTopic",
"JNDIName", "A WAS Topic", [['category', 'myCategory'], ['description', 'My description'], [
'expiry', 'SPECIFIED'], ['persistence', 'PERSISTENT'], ['priority', 'APPLICATION_DEFINED'],
['propertySet', [['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'],
['value', 'myDbName']], [['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]],
[['name', 'serverName'], ['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]], ['providerType', 'myProviderType'],
['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createWASTopicAtScope

This script creates a WebSphere Application Server topic in your JMS configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server topic in the respective cell.

To run the script, specify the scope, JMS provider name, name to assign to the topic, JNDI name, and the topic arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 265. Arguments for the createWASTopicAtScope script. Run the script to create a topic.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider of interest.
topicName	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
topic	Specifies the name of the topic (as a qualifier in the topic space) that this topic is to use.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 266. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties` `[[[name databaseName][type string]` `[value mys]]` `[[name driverType][type integer][value 4]]` `[[name serverName][type string]` `[value localhost]]` `[[name portNumber][type integer]` `[value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myProviderType']`
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	`['specifiedExpiry', '1000']`
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	`['specifiedPriority', '0']`

Syntax

```
AdminJMS.createWASTopicAtScope(scope,
 jmsProvider, topicName, jndiName, topic,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopic("myScope", "JMSTest",
 "TopicTest", "topicjndi", "mytopic")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopicAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASTopic", "JNDIName", "A WAS Topic", " category=myCategory, description='My description',
expiry=SPECIFIED, persistence=PERSISTENT, priority=APPLICATION_DEFINED")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopicAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASTopic", "JNDIName", "A WAS Topic", [['category', 'myCategory'],
['description', 'My description'],
['expiry', 'SPECIFIED'], ['persistence', 'PERSISTENT'], ['priority', 'APPLICATION_DEFINED'], ['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'],
['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myProviderType'], ['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createWASTopicUsingTemplateAtScope

This script uses a template to create a WebSphere Application Server topic in your JMS configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere topic in the respective cell.

To run the script, specify the scope, JMS provider name, configuration ID of the template, name to assign to the topic, JNDI name, and the topic arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 267. Arguments for the createWASTopicUsingTemplateAtScope script. Run the script to create a topic.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProvider	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server topic template to use.
topicName	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
topic	Specifies the name of the topic (as a qualifier in the topic space) that this topic is to use.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 268. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
expiry	Specifies the length of time after which messages that are sent to this destination expire and are dealt with according to their disposition options. Valid values are `APPLICATION_DEFINE`, `SPECIFIED`, and `UNLIMITED`. Do not put either of the values in quotes for the string format of the command.	`['expiry', 'SPECIFIED']`
persistence	Specifies the level of persistence. Valid values are `APPLICATION_DEFINED`, `NONPERSISTENT`, and `PERSISTENT`.	`['persistence', 'NONPERSISTENT']`
priority	Specifies the level of priority. Valid values are `APPLICATION_DEFINED` and `SPECIFIED`.	`['priority', 'SPECIFIED']`

Table 268. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string] [value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string] [value localhost]] [[name portNumber][type integer] [value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a specific vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myProviderType']
specifiedExpiry	Specifies the time in milliseconds after which messages on this queue are discarded.	['specifiedExpiry', '1000']
specifiedPriority	Specifies the priority from 0 to 9 of the WASQueue WebSphere queue.	['specifiedPriority', '0']

Syntax

```
AdminJMS.createWASTopicUsingTemplateAtScope(scope,
 jmsProvider, templateID, topicName,
 jndiName, topic, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopicUsingTemplateAtScope("myScope", "JMSTest",
 "Example.JMS.WAS.T1(templates/system|JMS-resource-provider-templates.xml#WASTopic_1)",
"TopicTest",
 "topicjndi", "mytopic")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopicUsingTemplateAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example.JMS.WAS.T1(templates/system|JMS-resource-provider-templates.xml#WASTopic_1)",
"My WASTopic", "JNDIName", "A WAS Topic", " category=myCategory, description='My description', expiry=SPECIFIED,
persistence=PERSISTENT, priority=APPLICATION_DEFINED")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopicUsingTemplateAtScope("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "Example.JMS.WAS.T1(templates/system|JMS-resource-provider-templates.xml#WASTopic_1)",
"My WASTopic", "JNDIName", "A WAS Topic", [['category', 'myCategory'], ['description', 'My description'],
['expiry', 'SPECIFIED'], ['persistence', 'PERSISTENT'], ['priority', 'APPLICATION_DEFINED'], ['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['specifiedExpiry', '1000'], ['specifiedPriority', '0']] )
```

createSIBJMSTopic

This script create a new JMS topic for the default messaging provider at the scope that you specify. The script returns the configuration ID of the created SIB JMS topic.

To run the script, specify the scope, name to assign to the topic, and the JNDI name. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 269. Arguments for the createSIBJMSTopic script. Run the script to create a JMS topic.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the default messaging provider.
name	Specifies the name to assign to the new topic.

Table 269. Arguments for the createSIBJMSTopic script (continued). Run the script to create a JMS topic.

Argument	Description
jndiName	Specifies the JNDI name that the system uses to bind this object into WebSphere Application Server JNDI.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 270. Optional attributes. Additional attributes available for the script.

Attributes	Description
description	Specifies a description of the SIB JMS topic.
topicSpace	Specifies the name of the underlying SIB topic space to which the topic points.
topicName	Specifies the topic used inside the topic space, for example, stock/IBM.
deliveryMode	Specifies the delivery mode for messages. Valid values are `Application`, `NonPersistent`, and `Persistent`.
timeToLive	Specifies the time in milliseconds for message expiration.
priority	Specifies the priority level to assign to messages sent to this destination. Valid values are `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`.
readAhead	Specifies the read-ahead value. Valid values are `AsConnection`, `AlwaysOn`, and `AlwaysOff`.
busName	The name of the bus on which the topic resides.

Syntax

```
AdminJMS.createSIBJMSTopic(scope,
 name,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSTopic("myScope", "myName", "myJNDIName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createSIBJMSTopic
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)", "myName" ,
"myJndiName9", "readAhead=AlwaysOff,timeToLive=6")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createSIBJMSTopic
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)", "myName" ,
"myJndiName9", [['readAhead', 'AlwaysOff'], ['timeToLive', '6']])
```

createWMQTopic

This script creates a JMS topic destination for the WebSphere MQ messaging provider at the scope that you specify. The script returns the configuration ID of the created WebSphere MQ topic.

To run the script, specify the scope, name to assign to the topic, and the JNDI name. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 271. Arguments for the createWMQTopic script. Run the script to create a JMS topic destination.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the WebSphere MQ messaging provider.
name	Specifies the name to assign to the new topic.
jndiName	Specifies the JNDI name that the system uses to bind this object into WebSphere Application Server JNDI.

Table 271. Arguments for the createWMQTopic script (continued). Run the script to create a JMS topic destination.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 272. Optional attributes. Additional attributes available for the script.

Attributes	Description
brokerCCDurSubQueue	Specifies the name of the queue from which a connection consumer receives durable subscription messages.
brokerDurSubQueue	Specifies the name of the queue from which a connection consumer receives non-durable subscription messages.
brokerPubQmgr	Specifies the name of the queue manager on which the broker is running.
brokerPubQueue	Specifies the name of the queue to which publication messages are sent.
brokerVersion	Specifies the level of functionality required for publish and subscribe operations.
ccsid	Specifies the coded character set identifier.
decimalEncoding	Specifies the decimal encoding setting for this topic. Valid values are `Normal` and `Reversed`.
description	Specifies an administrative description to associate with this WebSphere MQ JMS topic type destination.
expiry	Specifies the amount of time after which messages, sent to this destination, expire and are dealt with based on their disposition options. Valid values are `APP`, `UNLIM`, or any positive integer.
floatingPointEncoding	Specifies the floating point encoding setting for this topic. Valid values are `IEEENormal`, `IEEEReversed` and `z/OS`.
integerEncoding	Specifies the integer encoding setting for this topic. Valid values are `Normal` and `Reversed`.
persistence	Specifies the level of persistence used to store messages sent to this destination. Valid values are `APP`, `TDEF`, `PERS`, `NON` or `HIGH`.
priority	Specifies the priority level to assign to messages sent to this destination. Valid values are `APP`, `TDEF`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`.
readAheadClose	Specifies the behavior that occurs when closing a message consumer that is receiving messages asynchronously. The message consumer uses a message listener from a destination that has the readAhead parameter set to `true`. When a value of `DELIVERALL` is specified, the close method invocation waits until all read-ahead messages are delivered to the consumer before closing it. When a value of `DELIVERCURRENT` is specified, then the close() method only waits for any in-progress delivery to end before closing the consumer. Valid values are `DELIVERALL` and `DELIVERCURRENT`.
readAhead	Specifies whether messages for non-persistent consumers can be read ahead and cached. Valid values are `YES`, `NO` or `TDEF`.
sendAsync	Specifies whether messages can be sent to this destination without requiring that the queue manager acknowledges they have arrived. Valid values are `YES`, `NO` or `TDEF`.
useRFH2	Specifies whether an RFH version 2 header is appended to messages sent to this destination. Valid values are `true` or `false`.
useNativeEncoding	Specifies the native encoding settings for this topic.
wildcardFormat	Specifies which sets of characters are interpreted as topic wild cards.

Syntax

```
AdminJMS.createWMQTopic(scope,
 name,
 jndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWMQTopic("myScope", "myName", "myJNDIName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQTopic
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName10", "myTopicName10", "readAhead=YES,sendAsync=NO")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWMQTopic
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName10", "myTopicName10", [['readAhead', 'YES'], ['sendAsync', 'NO']])
```

createWASTopicConnectionFactory

This script creates a WebSphere Application Server topic connection factory in your configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server topic connection factory in the respective cell.

To run the script, specify the node, server, JMS provider name, name to assign to the connection factory, JNDI name, and the port arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 273. Arguments for the createWASTopicConnectionFactory script. Run the script to create a topic connection factory.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProviderName	Specifies the name of the JMS provider of interest.
topicConnFactoryName	Specifies the name to assign to the new connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
port	Specify the port of interest.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 274. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`

Table 274. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myJMSProviderType']`
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]`
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`['xaRecoveryAuthAlias', 'myCellManager01/a1']`

Table 275. Optional attributes, continued. Additional attributes available for the script.

Attributes	Description	Example
clientID	Specifies the client ID.	`['clientID', 'myClientID']`
cloneSupport	Specifies whether the WebSphere topic connection factory is supported across clones.	`['cloneSupport', 'true']`

Syntax

```
AdminJMS.createWASTopicConnectionFactory(nodeName,
 serverName, jmsProviderName, topicConnFactoryName,
 jndiName, port, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopicConnectionFactory("myNode", "myServer",
 "JMSTest", "TopicCFTest", "topiccfjndi", "DIRECT")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopicConnectionFactory("IBM-F4A849C57A0Node01", "server1",
"My JMS Provider Name1", "My WASTopicConnectionFactory", "JNDIName", "DIRECT", "XAEnabled=false,
authDataAlias=myAuthDataAlias, authMechanismPreference=BASIC_PASSWORD, category=myCategory,
clientID=myClientID, cloneSupport=true, description='My description', diagnoseConnectionUsage=false,
logMissingTransactionContext=true, manageCachedHandles=true, providerType=myProviderType,
xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopicConnectionFactory("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"My WASTopicConnectionFactory", "JNDIName", "DIRECT", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['clientID', 'myClientID'], ['cloneSupport', 'true'],
['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeThreshold', '10'],
['testConnection', 'true']]], ['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'],
['type', 'java.lang.String'], ['value', 'localhost']], [['name', 'portNumber'],
['type', 'java.lang.Integer'], ['value', 50000]]]]], ['providerType', 'myProviderType'], ['sessionPool',
[['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10],
['maxConnections', '12'], ['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeThreshold', '10'], ['testConnection', 'true']]],
['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createWASTopicConnectionFactoryUsingTemplate

This script uses a template to create a WebSphere Application Server topic connection factory in your configuration. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMS Server in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere topic using a template in the respective cell.

To run the script, specify the node, server, JMS provider name, configuration ID of the template, name to assign to the connection factory, JNDI name, and the port arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 276. Arguments for the createWASTopicConnectionFactoryUsingTemplate script. Run the script to create a topic connection factory.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
jmsProviderName	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server topic connection factory to use.
topicConnFactoryName	Specifies the name to assign to the new connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
port	Specifies the port of interest.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 277. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`

Table 277. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name, nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]]]]]]
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	['provider', 'myJMSProvider']
providerType	Specifies the JMS provider type used by this JMS provider.	['providerType', 'myJMSProviderType']
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	['xaRecoveryAuthAlias', 'myCellManager01/a1']

Table 278. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	['type', 'TOPIC']

Syntax

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate(nodeName,
 serverName, jmsProviderName, templateID,
 topicConnFactoryName, jndiName, port, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate("myNode", "myServer",
 "JMSTest", "First Example WAS
 TopicConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
 "TopicCFTest", "topiccfjndi", "DIRECT")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate
("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"First Example WAS TopicConnectionFactory
(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
"My WASTopicConnectionFactory", "JNDIName", "DIRECT", "XAEnabled=false, authDataAlias=myAuthDataAlias,
authMechanismPreference=BASIC_PASSWORD, category=myCategory, clientID=myClientID, cloneSupport=true,
description='My description', diagnoseConnectionUsage=false, logMissingTransactionContext=true,
manageCachedHandles=true, providerType=myProviderType, xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplate
("IBM-F4A849C57A0Node01", "server1", "My JMS Provider Name1",
"First Example WAS TopicConnectionFactory
(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
"My WASTopicConnectionFactory", "JNDIName", "'DIRECT'", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference', 'BASIC_PASSWORD'],
['category', 'myCategory'], ['clientID', 'myClientID'], ['cloneSupport', 'true'],
['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeThreshold', '10'],
['testConnection', 'true']]], ['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'], ['manageCachedHandles', 'true'], ['mapping',
[['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['propertySet', [['resourceProperties',
[[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']], [['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]]],
['providerType', 'myProviderType'], ['sessionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeThreshold', '10'],
['testConnection', 'true']]], ['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createWASTopicConnectionFactoryAtScope

This script creates a WebSphere Application Server topic connection factory in your configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMSServer in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere Application Server topic connection factory in the respective cell.

To run the script, specify the scope, JMS provider name, name to assign to the connection factory, JNDI name, and the port arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 279. Arguments for the createWASTopicConnectionFactoryAtScope script. Run the script to create a topic connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProviderName	Specifies the name of the JMS provider of interest.
topicConnFactoryName	Specifies the name to assign to the new connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
port	Specify the port of interest.

Table 279. Arguments for the createWASTopicConnectionFactoryAtScope script (continued). Run the script to create a topic connection factory.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 280. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'TOPIC']`

Syntax

```
AdminJMS.createWASTopicConnectionFactoryAtScope(scope,
 jmsProviderName, topicConnFactoryName,
 jndiName, port, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopicConnectionFactoryAtScope("myScope",
 "JMSTest", "TopicCFTest", "topiccfjndi", "DIRECT")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopicConnectionFactoryAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASTopicConnectionFactory", "JNDIName", "DIRECT", "XAEnabled=false,
authDataAlias=myAuthDataAlias,
authMechanismPreference=BASIC_PASSWORD, category=myCategory, clientID=myClientID, cloneSupport=true,
description='My description',
diagnoseConnectionUsage=false, logMissingTransactionContext=true, manageCachedHandles=true,
providerType=myProviderType,
xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopicConnectionFactoryAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "My WASTopicConnectionFactory", "JNDIName", "'DIRECT'", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'], ['authMechanismPreference','BASIC_PASSWORD'], ['category', 'myCategory'],
['clientID', 'myClientID'], ['cloneSupport', 'true'], ['connectionPool', [['agedTimeout', '100'],
['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeThreshold', '10'], ['testConnection', 'true']]],
['description', 'My description'],
['diagnoseConnectionUsage', 'false'], ['logMissingTransactionContext', 'true'],
['manageCachedHandles', 'true'],
['mapping', [['authDataAlias', 'authDataAliasValue'], ['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig',
[['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]], ['properties',
[[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]], ['propertySet',
[['resourceProperties', [[['name','databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'], ['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'],
['type', 'java.lang.String'],
['value', 'localhost']], [['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]],
['providerType', 'myProviderType'], ['sessionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'],
['freePoolDistributionTableSize', 10], ['maxConnections', '12'], ['minConnections', '5'],
['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'], ['numberOfUnsharedPoolPartitions', '3'], ['properties',
[[['description', 'My description'],
```

```
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]],
['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeThreshold', '10'], ['testConnection', 'true']]],
['xaRecoveryAuthAlias','myCellManager01/a1']] )
```

createWASTopicConnectionFactoryUsingTemplateAtScope

This script uses a template to create a WebSphere Application Server topic connection factory in your configuration at the scope that you specify. You should only use JMS resources for applications that perform messaging with a WebSphere Application Server version 5.1 embedded JMS Server in a Version 7.0 cell. The script returns the configuration ID of the created WebSphere topic using a template in the respective cell.

To run the script, specify the scope, JMS provider name, configuration ID of the template, name to assign to the connection factory, JNDI name, and the port arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 281. Arguments for the createWASTopicConnectionFactoryUsingTemplateAtScope script. Run the script to create a topic connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
jmsProviderName	Specifies the name of the JMS provider of interest.
templateID	Specifies the configuration ID of the WebSphere Application Server topic connection factory to use.
topicConnFactoryName	Specifies the name to assign to the new connection factory.
jndiName	Specifies the JNDI name that the system uses to bind the connection factory into the name space.
port	Specifies the port of interest.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 282. Optional attributes. Additional attributes available for the script.

Attributes	Description	Example
XAEnabled	Specifies whether XA recovery processing is enabled.	`['XAEnabled', 'false']`
authDataAlias	Specifies the alias used for database authentication at runtime.	`['authDataAlias', 'myAuthDataAlias']`
authMechanismPreference	Specifies the authentication mechanism. Valid values are `BASIC_PASSWORD` for basic authentication and `KERBEROS` for Kerberos authentication. Do not put either of the values in quotes for the string format of the command.	`['authMechanismPreference', 'BASIC_PASSWORD']`
category	Specifies the category that can be used to classify or group the resource.	`['category', 'myCategory']`
connectionPool	Specifies the JMS connection pooling properties for the parent JMS connection factory instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`['connectionPool', [['agedTimeout','100'],['connectionTimeout','1000'], ['freePoolDistributionTableSize',10],['maxConnections','12'], ['minConnections','5'],['numberOfFreePoolPartitions','3'], ['numberOfSharedPoolPartitions','6'], ['numberOfUnsharedPoolPartitions','3'],['properties', [['description','My description'],['name', 'myName'], ['required','false'],['type','String'], ['validationExpression',''],['value','myValue']]], ['purgePolicy','EntirePool'],['reapTime','10000'], ['struckThreshold','3'],['struckTime','10'], ['struckTimerTime','10'],['surgeCreationInterval','10'], ['surgeThreshold','10'],['testConnection','true'], ['testConnectionInterval','10'],['unusedTimeout','10000']]]`

Table 282. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
description	Specifies a description of the resource adapter.	`['description', 'My description']`
diagnoseConnectionUsage	Specifies whether connection usage diagnosis is enabled.	`['diagnoseConnectionUsage', 'false']`
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.	`['logMissingTransactionContext', 'true']`
manageCachedHandles	Specifies whether this data source is used for container-managed persistence of enterprise beans. The default value is `true`.	`['manageCachedHandles', 'true']`
mapping	Specifies the mapping of the configuration login to a specified authentication alias name. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["mapping",[["authDataAlias","authDataAliasValue"], ["mappingConfigAlias","mappingConfigAliasValue"]]]`
preTestConfig	Specifies the pretest connection configuration settings. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["preTestConfig",[["preTestConnection", "true"], ["retryInterval", "12343"],["retryLimit", "4"]]]`
properties	Specifies either a typed property type or a descriptive property type. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["properties",[["description","My description"], ["name","myName"],["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]]`
propertySet	Optionally specifies resource properties in the following format: [propertySet[[resourceProperties[[[name$_1$ nameValue$_1$][type$_1$ typeValue$_1$][value$_1$ valueValue$_1$]]... [[name$_n$ nameValue$_n$][type$_n$ typeValue$_n$][value$_n$ valueValue$_n$]]]]]] When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`[propertySet [[resourceProperties [[[name databaseName][type string][value mys]] [[name driverType][type integer][value 4]] [[name serverName][type string][value localhost]] [[name portNumber][type integer][value 50000]] ]]]]`
provider	Specifies the JMS driver implementation class for access to a vendor database. To create a pool of connections to that database, associate a data source with the JMS provider.	`['provider', 'myJMSProvider']`
providerType	Specifies the JMS provider type used by this JMS provider.	`['providerType', 'myJMSProviderType']`
sessionPool	Specifies the JMS session pooling properties for the parent JMS connection instance. When you use this attribute in a script, use the list format. The string format does not work because this attribute is a configuration object type.	`["sessionPool\", [["agedTimeout","100"],["connectionTimeout","1000"], ["freePoolDistributionTableSize",10],["maxConnections","12"], ["minConnections","5"],["numberOfFreePoolPartitions","3"], ["numberOfSharedPoolPartitions","6"], ["numberOfUnsharedPoolPartitions","3"],["properties", [["description","My description"],["name", "myName"], ["required","false"],["type","String"], ["validationExpression",""],["value","myValue"]]], ["purgePolicy",'EntirePool'],["reapTime","10000"], ["struckThreshold","3"],["struckTime","10"], ["struckTimerTime","10"],["surgeCreationInterval","10"], ["surgeThreshold","10"],["testConnection","true"], ["testConnectionInterval","10"],["unusedTimeout","10000"]]]`

Table 282. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description	Example
xaRecoveryAuthAlias	Specifies the database authentication alias used during XA recovery processing. When this property is specified, the default value is the alias for application authentication.	`['xaRecoveryAuthAlias', 'myCellManager01/a1']`

Table 283. Optional attributes, continued. Additional attribute available for the script.

Attributes	Description	Example
type	Specifies `QUEUE` for queues, `TOPIC` for topics, and `UNIFIED` for both queues and topics. Do not put either of the values in quotes for the string format of the command.	`['type', 'TOPIC']`

Syntax

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplateAtScope(scope,
 jmsProviderName, templateID,
 topicConnFactoryName, jndiName,  port, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplateAtScope("myScope",
 "JMSTest", "First Example WAS
 TopicConnectionFactory(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
 "TopicCFTest", "topiccfjndi", "DIRECT")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "First Example WAS TopicConnectionFactory
(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
"My WASTopicConnectionFactory", "JNDIName", "DIRECT", "XAEnabled=false, authDataAlias=myAuthDataAlias,
 authMechanismPreference=BASIC_PASSWORD,
category=myCategory, clientID=myClientID, cloneSupport=true, description='My description',
 diagnoseConnectionUsage=false,
logMissingTransactionContext=true, manageCachedHandles=true, providerType=myProviderType,
xaRecoveryAuthAlias=myCellManager01/a1")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWASTopicConnectionFactoryUsingTemplateAtScope
("Cell=IBM-F4A849C57A0Cell01,Node=IBM-F4A849C57A0Node01,Server=server1",
"My JMS Provider Name1", "First Example WAS TopicConnectionFactory
(templates/system|JMS-resource-provider-templates.xml#WASTopicConnectionFactory_1)",
"My WASTopicConnectionFactory", "JNDIName", "'DIRECT'", [['XAEnabled', 'false'],
['authDataAlias', 'myAuthDataAlias'],
['authMechanismPreference', 'BASIC_PASSWORD'], ['category', 'myCategory'], ['clientID', 'myClientID'],
['cloneSupport', 'true'],
['connectionPool', [['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10],
['maxConnections', '12'], ['minConnections', '5'], ['numberOfFreePoolPartitions', '3'],
['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'], ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'],
['validationExpression', ''], ['value', 'myValue']]]], ['purgePolicy', 'EntirePool'],
['reapTime', '10000'], ['surgeThreshold', '10'],
['testConnection', 'true']]], ['description', 'My description'], ['diagnoseConnectionUsage', 'false'],
['logMissingTransactionContext', 'true'],
['manageCachedHandles', 'true'], ['mapping', [['authDataAlias', 'authDataAliasValue'],
['mappingConfigAlias', 'mappingConfigAliasValue']]],
['preTestConfig', [['preTestConnection', 'true'], ['retryInterval', '12343'], ['retryLimit', '4']]],
 ['properties', [[['description', 'My description'],
['name', 'myName'], ['required', 'false'], ['validationExpression', ''], ['value', 'myValue']]]],
['propertySet',
[['resourceProperties', [[['name', 'databaseName'], ['type', 'java.lang.String'], ['value', 'myDbName']],
[['name', 'driverType'],
['type', 'java.lang.Integer'], ['value', 4]], [['name', 'serverName'], ['type', 'java.lang.String'],
['value', 'localhost']],
[['name', 'portNumber'], ['type', 'java.lang.Integer'], ['value', 50000]]]]], ['providerType', 'myProviderType'],
['sessionPool',
[['agedTimeout', '100'], ['connectionTimeout', '1000'], ['freePoolDistributionTableSize', 10],
```

```
['maxConnections', '12'],
['minConnections', '5'], ['numberOfFreePoolPartitions', '3'], ['numberOfSharedPoolPartitions', '6'],
['numberOfUnsharedPoolPartitions', '3'],
['properties', [[['description', 'My description'], ['name', 'myName'], ['required', 'false'],
 ['validationExpression', ''],
['value', 'myValue']]]], ['purgePolicy', 'EntirePool'], ['reapTime', '10000'], ['surgeThreshold', '10'],
 ['testConnection', 'true']]],
['xaRecoveryAuthAlias', 'myCellManager01/a1']] )
```

createSIBJMSConnectionFactory

The script creates a new SIB JMS connection factory for the default messaging provider at the scope that you specify. The script returns the configuration ID of the created SIB JMS connection factory.

To run the script, specify the scope, name, JNDI name, and bus name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 284. Arguments for the createSIBJMSConnectionFactory script. Run the script to create a JMS connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the administrative name assigned to this connection factory.
jndiName	Specifies the JNDI name that is specified in the bindings for message-driven beans associated with this connection factory.
busName	Specifies the name of the service integration bus to which connections are made.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 285. Optional attributes. Additional attributes available for the script.

Attributes	Description
authDataAlias	Specifies a user ID and password to be used to authenticate connections to the JMS provider for application-managed authentication.
containerAuthAlias	Specifies a container managed authentication alias, from which security credentials are used to establish a connection to the JMS provider.
mappingAlias	Specifies the Java Authentication and Authorization Service (JAAS) mapping alias to use when determining the security related credentials. The security related credentials are used when establishing a connection to the JMS provider.
xaRecoveryAuthAlias	Specifies the authentication alias used during XA recovery processing.
category	Specifies the category that can be used to classify or group the connection factory
description	Specifies a description of the connection factory.
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.
manageCachedHandles	Specifies whether cached handles , which are handles held in instance variables in a bean, are tracked by the container
clientID	Specifies the client ID which is required only for durable subscriptions.
userName	Specifies the user name that is used to create connections from the connection factory.
password	Specifies the password that is used to create connections from the connection factory.
nonPersistentMapping	Specifies a non-persistent mapping value. Valid values are `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination`, and `None`.
persistentMapping	Specifies a persistent mapping value. Valid values are `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination`and `None`.
durableSubscriptionHome	Specifies the durable subscription home value.
readAhead	Specifies the read-ahead value. Valid values are `Default`, `AlwaysOn`, and `AlwaysOff`.
target	Specifies the name of a target that resolves to a group of messaging engines.
targetType	Specifies the type of the name in the target parameter. Valid values are `BusMember`, `Custom`, and `ME`.
targetSignificance	Specifies the significance of the target group. Valid values are `Preferred` and `Required`.

Table 285. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
targetTransportChain	Specifies the name of the protocol that to connect to a remote messaging engine.
providerEndPoints	Specifies a comma-separated list of endpoint triplets of the form *host:port:chain*.
connectionProximity	Specifies the proximity of acceptable messaging engines. Valid values are `Bus`, `Host`, `Cluster`, and `Server`.
tempQueueNamePrefix	Specifies a temporary queue name prefix.
tempTopicNamePrefix	Specifies a temporary topic name prefix.
shareDataSourceWithCMP	Specifies how to control data sources that are shared.
shareDurableSubscriptions	Specifies how to control durable subscriptions that are shared. Valid values are `InCluster`, `AlwaysShared`, and `NeverShared`.
consumerDoesNotModifyPayloadAfterGet	Specifies that when a message consuming application receives object or byte messages, the system serializes the message data only when necessary. The application is connected to the bus using this connection factory. Applications that obtain the data from these messages must treat the data as read-only data. Valid values are `true` and `false`. The default value is `false`.
producerDoesNotModifyPayloadAfterSet	Specifies that when a message consuming application sends object or byte messages, the data is not copied and the system serializes the data only when necessary. The application is connected to the bus using this connection factory. Applications sending such messages must not modify the data after it has been set in a message. Valid values are `true` and `false`. The default value is `false`.

Syntax

```
AdminJMS.createSIBJMSConnectionFactory(scope,
 name, jndiName,
 busName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSConnectionFactory("myScope", "myName", "myJNDIName",
"MyBusName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName2", "readAhead=AlwaysOff,description=my description")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createSIBJMSConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)", "myName"
, "myJndiName1", "myBusName", [['readAhead', 'AlwaysOff'], ['description', 'my description']])
```

createWMQConnectionFactory

The script creates a connection factory for the WebSphere MQ messaging provider at the scope that you specify. The script returns the configuration ID of the created WebSphere MQ connection factory.

To run the script, specify the scope, name, and JNDI name, arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 286. Arguments for the createWMQConnectionFactory script. Run the script to create a connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the administrative name assigned to this WebSphere MQ messaging provider connection factory.
jndiName	Specifies the name and location used to bind this object into WebSphere Application Server JNDI.

Table 286. Arguments for the createWMQConnectionFactory script (continued). Run the script to create a connection factory.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 287. Optional attributes. Additional attributes available for the script.

Attributes	Description
maxBatchSize	Specifies the maximum number of messages to take from a queue in one packet when using asynchronous message delivery.
brokerCCSubQueue	Specifies the name of the queue from which non-durable subscription messages are retrieved for a connection consumer. This parameter is valid only for topic connection factories.
brokerCtrlQueue	Specifies the broker control queue to use if this connection factory is to subscribe to a topic. This parameter is valid only for topic connection factories.
brokerQmgr	Specifies the name of the queue manager on which the queue manager is running. This parameter is valid only for topic connection factories.
brokerSubQueue	Specifies the queue for obtaining subscription messages if this connection factory subscribes to a topic. This parameter is valid only for topic connection factories.
brokerVersion	Specifies the level of functionality required for publish and subscribe operations. This parameter is valid only for topic connection factories.
brokerPubQueue	Specifies the queue to send publication messages to when using queue based brokering. This parameter is valid only for topic connection factories.
ccdtQmgrName	Specifies a queue manager name that is used to select one or more entries from a client channel definition table.
ccdtUrl	Specifies a URL to a client channel definition table. Use this attribute for this connection factory, when contacting the WebSphere MQ messaging provider. Connection factories created using this attribute are ccdtURL connection factories.
ccsid	Specifies the coded character set ID to use on connections.
cleanupInterval	Specifies the interval between background executions of the publish and subscribe cleanup utility. This parameter is valid only for topic connection factories.
cleanupLevel	Specifies the cleanup Level for BROKER or MIGRATE subscription stores. This parameter is valid only for topic connection factories.
clientId	Specifies the client identifier used for connections started using this connection factory.
clonedSubs	Specifies whether two or more instances of the same durable topic subscriber can run simultaneously. This parameter is valid only for topic connection factories.
compressHeaders	Determines if message headers are compressed or not.
compressPayload	Determines if message payloads are compressed or not.
containerAuthAlias	Specifies a container managed authentication alias that has security credentials that are used for establishing a connection to the WebSphere MQ messaging provider.
description	Specifies a description of the connection factory.
failIfQuiescing	Specifies the behavior of certain calls to the queue manager when the queue manager is put into a quiescent state.
localAddress	Specifies either or both of the following items: • The local network interface to be used. • The local port, or range of local ports, to be used.
mappingAlias	Specifies the JAAS mapping alias to use when determining the security related credentials. The security related credentials are used when establishing a connection to the WebSphere MQ.
modelQueue	Specifies the WebSphere MQ model queue definition to use as a basis when creating JMS temporary destinations. This parameter is valid only for queue connection factories.
msgRetention	Specifies whether the connection consumer keeps unwanted messages on the input queue. A value of `true` means that it does. A value of `false` means that the messages are disposed of based on their disposition options. This parameter is valid only for queue connection factories.
msgSelection	Specifies where message selection occurs. This parameter is valid only for topic connection factories.

Table 287. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
pollingInterval	Specifies in milliseconds the maximum time that elapses during a polling interval. If a message listener within a session has no suitable message on its queue, the message listener uses the polling interval to determine how often to poll its queue for a message. Increase the value for this property if sessions frequently do not have a suitable message available. This attribute is applicable only in the client container.
providerVersion	Specifies the minimum version and capabilities of the queue manager.
pubAckInterval	Specifies the number of publications to send to a queue based broker before sending a publication which solicits an acknowledgement. This attribute is valid only for topic connection factories.
qmgrHostname	Specifies the hostname that this connection factory uses when attempting a client mode connection to WebSphere MQ.
qmgrName	Specifies the queue manager name that this connection factory uses when contacting the WebSphere MQ messaging provider. Connection factories created using this parameter are user-defined connection factories.
qmgrPortNumber	Specifies the port number that this connection factory uses when attempting a client mode connection to WebSphere MQ.
qmgrSvrconnChannel	Specifies the SVRCONN channel to use when connecting to WebSphere MQ. Connection factories created using this parameter are user-defined connection factories.
rcvExitInitData	Specifies initialization data to pass to the receive exit.
rcvExit	Specifies a comma separated list of receive exit class names.
replyWithRFH2	Specifies whether, when replying to a message, an RFH version 2 header is included in the reply message. This parameter is valid only for queue connection factories.
rescanInterval	Specifies in milliseconds the maximum time that elapses during a scanning interval. When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, the WebSphere MQ JMS client searches the WebSphere MQ queue for suitable messages in the sequence determined by the MsgDeliverySequence attribute of the queue. When the client finds a suitable message and delivers it to the consumer, the client resumes the search for the next suitable message from its current position in the queue. The client continues to search the queue until it reaches the end of the queue, or until the interval of time specified by this property has expired. In each case, the client returns to the beginning of the queue to continue its search, and a new time interval starts. This parameter is only valid for queue connection factories.
secExitInitData	Specifies initialization data to pass to the security exit.
secExit	Specifies a comma separated list of security exit class names.
sendExitInitData	Specifies initialization data to pass to the send exit.
sendExit	Specifies a comma separated list of send exit class names.
sparseSubs	Specifies the message retrieval policy of a TopicSubscriber object. This parameter is only valid for topic connection factories.
sslConfiguration	Specifies a specific Secure Sockets Layer (SSL) configuration to secure network connections to the queue manager.
sslCrl	Specifies a list of LDAP servers which can be used to provide certificate revocation information if this connection factory establishes an SSL connection to WebSphere MQ.
sslPeerName	Specifies a peer name to match against the distinguished name in the peer certificate. This attribute is used when the connection factory establishes an SSL connection to the queue manager.
sslResetCount	Specifies the number of bytes to transfer before resetting the symmetric encryption key used for the SSL session. This attribute is used when the connection factory establishes an SSL connection to the queue manager.
sslType	Specifies the configuration, if any, when the connection factory establishes an SSL connection to the queue manager.
stateRefreshInt	Specifies in milliseconds the maximum time that elapses between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if subStore attribute has the value `QUEUE`. This attribute is valid only for topic connection factories.
subStore	Specifies that WebSphere MQ JMS stores persistent data relating to active subscriptions. This attribute is valid only for topic connection factories.
support2PCProtocol	Specifies whether the connection factory acts as a resource which is capable of participating in distributed two phase commit processing.
tempQueuePrefix	Specifies the prefix to apply to WebSphere MQ temporary queues. These temporary queues represent JMS temporary queue type destinations. This attribute is valid only for queue connection factories.
tempTopicPrefix	Specifies the prefix to apply to WebSphere MQ temporary topics. These temporary topics represent JMS temporary topic type destinations. This attribute is valid only for topic connection factories.
wildcardFormat	Specifies which sets of characters are interpreted as topic wild cards. This attribute is valid only for topic connection factories.

Table 287. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
wmqTransportType	Specifies how this connection factory connects to WebSphere MQ. Connection factories created using this attribute are user-defined. Valid values are `BINDINGS`, `BINDINGS_THEN_CLIENT`, and `CLIENT`.
xaRecoveryAuthAlias	Specifies the authentication alias to connect to WebSphere MQ for XA recovery.

Syntax

```
AdminJMS.createWMQConnectionFactory(scope,
 name, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWMQConnectionFactory("myScope", "myName", "myJNDIName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName2", "maxBatchSize=15,description=my description")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWMQConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName2", [['maxBatchSize', '15'], ['description', 'my description']])
```

createSIBJMSQueueConnectionFactory

The script creates a new SIB JMS queue connection factory for the default messaging provider at the scope that you specify. The script returns the configuration ID of the created SIB JMS queue connection factory.

To run the script, specify the scope, name, JNDI name, and bus name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 288. Arguments for the createSIBJMSQueueConnectionFactory script. Run the script to create a JMS queue connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the administrative name assigned to this connection factory.
jndiName	Specifies the JNDI name that is specified in the bindings for message-driven beans associated with this connection factory.
busName	Specifies the name of the service integration bus to which connections are made.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 289. Optional attributes. Additional attributes available for the script.

Attributes	Description
authDataAlias	Specifies a user ID and password to be used to authenticate connections to the JMS provider for application-managed authentication.
containerAuthAlias	Specifies a container managed authentication alias, from which security credentials are used to establish a connection to the JMS provider.
mappingAlias	Specifies the Java Authentication and Authorization Service (JAAS) mapping alias to use when determining the security related credentials. The security related credentials are used when establishing a connection to the JMS provider.

Table 289. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
xaRecoveryAuthAlias	Specifies the authentication alias used during XA recovery processing.
category	Specifies the category that can be used to classify or group the connection factory
description	Specifies a description of the connection factory.
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.
manageCachedHandles	Specifies whether cached handles , which are handles held in instance variables in a bean, are tracked by the container
clientID	Specifies the client ID which is required only for durable subscriptions.
userName	Specifies the user name that is used to create connections from the connection factory.
password	Specifies the password that is used to create connections from the connection factory.
nonPersistentMapping	Specifies a non-persistent mapping value. Valid values are `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination`, and `None`.
persistentMapping	Specifies a persistent mapping value. Valid values are `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination`and `None`.
durableSubscriptionHome	Specifies the durable subscription home value.
readAhead	Specifies the read-ahead value. Valid values are `Default`, `AlwaysOn`, and `AlwaysOff`.
target	Specifies the name of a target that resolves to a group of messaging engines.
targetType	Specifies the type of the name in the target parameter. Valid values are `BusMember`, `Custom`, and `ME`.
targetSignificance	Specifies the significance of the target group. Valid values are `Preferred` and `Required`.
targetTransportChain	Specifies the name of the protocol that to connect to a remote messaging engine.
providerEndPoints	Specifies a comma-separated list of endpoint triplets of the form *host:port:chain*.
connectionProximity	Specifies the proximity of acceptable messaging engines. Valid values are `Bus`, `Host`, `Cluster`, and `Server`.
tempQueueNamePrefix	Specifies a temporary queue name prefix.
tempTopicNamePrefix	Specifies a temporary topic name prefix.
shareDataSourceWithCMP	Specifies how to control data sources that are shared.
shareDurableSubscriptions	Specifies how to control durable subscriptions that are shared. Valid values are `InCluster`, `AlwaysShared`, and `NeverShared`.
consumerDoesNotModifyPayloadAfterGet	Specifies that when a message consuming application receives object or byte messages, the system serializes the message data only when necessary. The application is connected to the bus using this connection factory. Applications that obtain the data from these messages must treat the data as read-only data. Valid values are `true` and `false`. The default value is `false`.
producerDoesNotModifyPayloadAfterSet	Specifies that when a message consuming application sends object or byte messages, the data is not copied and the system serializes the data only when necessary. The application is connected to the bus using this connection factory. Applications sending such messages must not modify the data after it has been set in a message. Valid values are `true` and `false`. The default value is `false`.

Syntax

```
AdminJMS.createSIBJMSQueueConnectionFactory(scope,
 name, jndiName,
 busName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSQueueConnectionFactory("myScope", "myName", "myJNDIName",
"MyBusName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createSIBJMSQueueConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)", "myName" ,
"myJndiName3", "myBusName", "readAhead=AlwaysOff,description=my description")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createSIBJMSQueueConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)", "myName" ,
"myJndiName3", "myBusName", [['readAhead', 'AlwaysOff'], ['description', 'my description']])
```

createWMQQueueConnectionFactory

The script creates a queue connection factory for the WebSphere MQ messaging provider at the scope that you specify. The script returns the configuration ID of the created WebSphere MQ queue connection factory.

To run the script, specify the scope, name, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 290. Arguments for the createWMQQueueConnectionFactory script. Run the script to create a queue connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the administrative name assigned to this WebSphere MQ messaging provider connection factory.
jndiName	Specifies the name and location used to bind this object into WebSphere Application Server JNDI.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 291. Optional attributes. Additional attributes available for the script.

Attributes	Description
maxBatchSize	Specifies the maximum number of messages to take from a queue in one packet when using asynchronous message delivery.
brokerCCSubQueue	Specifies the name of the queue from which non-durable subscription messages are retrieved for a connection consumer. This parameter is valid only for topic connection factories.
brokerCtrlQueue	Specifies the broker control queue to use if this connection factory is to subscribe to a topic. This parameter is valid only for topic connection factories.
brokerQmgr	Specifies the name of the queue manager on which the queue manager is running. This parameter is valid only for topic connection factories.
brokerSubQueue	Specifies the queue for obtaining subscription messages if this connection factory subscribes to a topic. This parameter is valid only for topic connection factories.
brokerVersion	Specifies the level of functionality required for publish and subscribe operations. This parameter is valid only for topic connection factories.
brokerPubQueue	Specifies the queue to send publication messages to when using queue based brokering. This parameter is valid only for topic connection factories.
ccdtQmgrName	Specifies a queue manager name that is used to select one or more entries from a client channel definition table.
ccdtUrl	Specifies a URL to a client channel definition table. Use this attribute for this connection factory, when contacting the WebSphere MQ messaging provider. Connection factories created using this attribute are ccdtURL connection factories.
ccsid	Specifies the coded character set ID to use on connections.
cleanupInterval	Specifies the interval between background executions of the publish and subscribe cleanup utility. This parameter is valid only for topic connection factories.
cleanupLevel	Specifies the cleanup Level for BROKER or MIGRATE subscription stores. This parameter is valid only for topic connection factories.
clientId	Specifies the client identifier used for connections started using this connection factory.
clonedSubs	Specifies whether two or more instances of the same durable topic subscriber can run simultaneously. This parameter is valid only for topic connection factories.
compressHeaders	Determines if message headers are compressed or not.
compressPayload	Determines if message payloads are compressed or not.
containerAuthAlias	Specifies a container managed authentication alias that has security credentials that are used for establishing a connection to the WebSphere MQ messaging provider.
description	Specifies a description of the connection factory.

Table 291. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
failIfQuiescing	Specifies the behavior of certain calls to the queue manager when the queue manager is put into a quiescent state.
localAddress	Specifies either or both of the following items: • The local network interface to be used. • The local port, or range of local ports, to be used.
mappingAlias	Specifies the JAAS mapping alias to use when determining the security related credentials. The security related credentials are used when establishing a connection to the WebSphere MQ.
modelQueue	Specifies the WebSphere MQ model queue definition to use as a basis when creating JMS temporary destinations. This parameter is valid only for queue connection factories.
msgRetention	Specifies whether the connection consumer keeps unwanted messages on the input queue. A value of `true` means that it does. A value of `false` means that the messages are disposed of based on their disposition options. This parameter is valid only for queue connection factories.
msgSelection	Specifies where message selection occurs. This parameter is valid only for topic connection factories.
pollingInterval	Specifies in milliseconds the maximum time that elapses during a polling interval. If a message listener within a session has no suitable message on its queue, the message listener uses the polling interval to determine how often to poll its queue for a message. Increase the value for this property if sessions frequently do not have a suitable message available. This attribute is applicable only in the client container.
providerVersion	Specifies the minimum version and capabilities of the queue manager.
pubAckInterval	Specifies the number of publications to send to a queue based broker before sending a publication which solicits an acknowledgement. This attribute is valid only for topic connection factories.
qmgrHostname	Specifies the hostname that this connection factory uses when attempting a client mode connection to WebSphere MQ.
qmgrName	Specifies the queue manager name that this connection factory uses when contacting the WebSphere MQ messaging provider. Connection factories created using this parameter are user-defined connection factories.
qmgrPortNumber	Specifies the port number that this connection factory uses when attempting a client mode connection to WebSphere MQ.
qmgrSvrconnChannel	Specifies the SVRCONN channel to use when connecting to WebSphere MQ. Connection factories created using this parameter are user-defined connection factories.
rcvExitInitData	Specifies initialization data to pass to the receive exit.
rcvExit	Specifies a comma separated list of receive exit class names.
replyWithRFH2	Specifies whether, when replying to a message, an RFH version 2 header is included in the reply message. This parameter is valid only for queue connection factories.
rescanInterval	Specifies in milliseconds the maximum time that elapses during a scanning interval. When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, the WebSphere MQ JMS client searches the WebSphere MQ queue for suitable messages in the sequence determined by the MsgDeliverySequence attribute of the queue. When the client finds a suitable message and delivers it to the consumer, the client resumes the search for the next suitable message from its current position in the queue. The client continues to search the queue until it reaches the end of the queue, or until the interval of time specified by this property has expired. In each case, the client returns to the beginning of the queue to continue its search, and a new time interval starts. This parameter is only valid for queue connection factories.
secExitInitData	Specifies initialization data to pass to the security exit.
secExit	Specifies a comma separated list of security exit class names.
sendExitInitData	Specifies initialization data to pass to the send exit.
sendExit	Specifies a comma separated list of send exit class names.
sparseSubs	Specifies the message retrieval policy of a TopicSubscriber object. This parameter is only valid for topic connection factories.
sslConfiguration	Specifies a specific Secure Sockets Layer (SSL) configuration to secure network connections to the queue manager.
sslCrl	Specifies a list of LDAP servers which can be used to provide certificate revocation information if this connection factory establishes an SSL connection to WebSphere MQ.
sslPeerName	Specifies a peer name to match against the distinguished name in the peer certificate. This attribute is used when the connection factory establishes an SSL connection to the queue manager.
sslResetCount	Specifies the number of bytes to transfer before resetting the symmetric encryption key used for the SSL session. This attribute is used when the connection factory establishes an SSL connection to the queue manager.
sslType	Specifies the configuration, if any, when the connection factory establishes an SSL connection to the queue manager.

Table 291. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
stateRefreshInt	Specifies in milliseconds the maximum time that elapses between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if subStore attribute has the value `QUEUE`. This attribute is valid only for topic connection factories.
subStore	Specifies that WebSphere MQ JMS stores persistent data relating to active subscriptions. This attribute is valid only for topic connection factories.
support2PCProtocol	Specifies whether the connection factory acts as a resource which is capable of participating in distributed two phase commit processing.
tempQueuePrefix	Specifies the prefix to apply to WebSphere MQ temporary queues. These temporary queues represent JMS temporary queue type destinations. This attribute is valid only for queue connection factories.
tempTopicPrefix	Specifies the prefix to apply to WebSphere MQ temporary topics. These temporary topics represent JMS temporary topic type destinations. This attribute is valid only for topic connection factories.
wildcardFormat	Specifies which sets of characters are interpreted as topic wild cards. This attribute is valid only for topic connection factories.
wmqTransportType	Specifies how this connection factory connects to WebSphere MQ. Connection factories created using this attribute are user-defined. Valid values are `BINDINGS`, `BINDINGS_THEN_CLIENT`, and `CLIENT`.
xaRecoveryAuthAlias	Specifies the authentication alias to connect to WebSphere MQ for XA recovery.

Syntax

```
AdminJMS.createWMQQueueConnectionFactory(scope,
 name, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWMQQueueConnectionFactory("myScope", "myName", "myJNDIName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQQueueConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)", "myName
" , "myJndiName4", "maxBatchSize=15,description=my description")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWMQQueueConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName4", [['maxBatchSize', '15'], ['description', 'my description']])
```

createSIBJMSTopicConnectionFactory

The script creates a new SIB JMS topic connection factory for the default messaging provider at the scope that you specify. The script returns the configuration ID of the created SIB JMS topic connection factory.

To run the script, specify the scope, name, JNDI name, and bus name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 292. Arguments for the createSIBJMSTopicConnectionFactory script. Run the script to create a SIB JMS topic connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the administrative name assigned to this connection factory.
jndiName	Specifies the JNDI name that is specified in the bindings for message-driven beans associated with this connection factory.
busName	Specifies the name of the service integration bus to which connections are made.

Table 292. Arguments for the createSIBJMSTopicConnectionFactory script (continued). Run the script to create a SIB JMS topic connection factory.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 293. Optional attributes. Additional attributes available for the script.

Attributes	Description
authDataAlias	Specifies a user ID and password to be used to authenticate connections to the JMS provider for application-managed authentication.
containerAuthAlias	Specifies a container managed authentication alias, from which security credentials are used to establish a connection to the JMS provider.
mappingAlias	Specifies the Java Authentication and Authorization Service (JAAS) mapping alias to use when determining the security related credentials. The security related credentials are used when establishing a connection to the JMS provider.
xaRecoveryAuthAlias	Specifies the authentication alias used during XA recovery processing.
category	Specifies the category that can be used to classify or group the connection factory
description	Specifies a description of the connection factory.
logMissingTransactionContext	Specifies whether missing transaction context logging is enabled.
manageCachedHandles	Specifies whether cached handles , which are handles held in instance variables in a bean, are tracked by the container
clientID	Specifies the client ID which is required only for durable subscriptions.
userName	Specifies the user name that is used to create connections from the connection factory.
password	Specifies the password that is used to create connections from the connection factory.
nonPersistentMapping	Specifies a non-persistent mapping value. Valid values are `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination`, and `None`.
persistentMapping	Specifies a persistent mapping value. Valid values are `BestEffortNonPersistent`, `ExpressNonPersistent`, `ReliableNonPersistent`, `ReliablePersistent`, `AssuredPersistent`, `AsSIBDestination`and `None`.
durableSubscriptionHome	Specifies the durable subscription home value.
readAhead	Specifies the read-ahead value. Valid values are `Default`, `AlwaysOn`, and `AlwaysOff`.
target	Specifies the name of a target that resolves to a group of messaging engines.
targetType	Specifies the type of the name in the target parameter. Valid values are `BusMember`, `Custom`, and `ME`.
targetSignificance	Specifies the significance of the target group. Valid values are `Preferred` and `Required`.
targetTransportChain	Specifies the name of the protocol that to connect to a remote messaging engine.
providerEndPoints	Specifies a comma-separated list of endpoint triplets of the form *host:port:chain*.
connectionProximity	Specifies the proximity of acceptable messaging engines. Valid values are `Bus`, `Host`, `Cluster`, and `Server`.
tempQueueNamePrefix	Specifies a temporary queue name prefix.
tempTopicNamePrefix	Specifies a temporary topic name prefix.
shareDataSourceWithCMP	Specifies how to control data sources that are shared.
shareDurableSubscriptions	Specifies how to control durable subscriptions that are shared. Valid values are `InCluster`, `AlwaysShared`, and `NeverShared`.
consumerDoesNotModifyPayloadAfterGet	Specifies that when a message consuming application receives object or byte messages, the system serializes the message data only when necessary. The application is connected to the bus using this connection factory. Applications that obtain the data from these messages must treat the data as read-only data. Valid values are `true` and `false`. The default value is `false`.
producerDoesNotModifyPayloadAfterSet	Specifies that when a message consuming application sends object or byte messages, the data is not copied and the system serializes the data only when necessary. The application is connected to the bus using this connection factory. Applications sending such messages must not modify the data after it has been set in a message. Valid values are `true` and `false`. The default value is `false`.

Syntax

```
AdminJMS.createSIBJMSTopicConnectionFactory(scope,
 name, jndiName,
 busName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSTopicConnectionFactory("myScope", "myName", "myJNDIName",
"MyBusName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createSIBJMSTopicConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName5", "myBusName", "readAhead=AlwaysOff,description=my description")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createSIBJMSTopicConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName5", "myBusName", [['readAhead', 'AlwaysOff'], ['description', 'my description']])
```

createWMQTopicConnectionFactory

The script creates a topic connection factory for the WebSphere MQ messaging provider at the scope that you specify. The script returns the configuration ID of the created WebSphere MQ topic connection factory.

To run the script, specify the scope, name, and JNDI name, arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 294. Arguments for the createWMQTopicConnectionFactory script. Run the script to create a topic connection factory.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
name	Specifies the administrative name assigned to this WebSphere MQ messaging provider connection factory.
jndiName	Specifies the name and location used to bind this object into WebSphere Application Server JNDI.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 295. Optional attributes. Additional attributes available for the script.

Attributes	Description
maxBatchSize	Specifies the maximum number of messages to take from a queue in one packet when using asynchronous message delivery.
brokerCCSubQueue	Specifies the name of the queue from which non-durable subscription messages are retrieved for a connection consumer. This parameter is valid only for topic connection factories.
brokerCtrlQueue	Specifies the broker control queue to use if this connection factory is to subscribe to a topic. This parameter is valid only for topic connection factories.
brokerQmgr	Specifies the name of the queue manager on which the queue manager is running. This parameter is valid only for topic connection factories.
brokerSubQueue	Specifies the queue for obtaining subscription messages if this connection factory subscribes to a topic. This parameter is valid only for topic connection factories.
brokerVersion	Specifies the level of functionality required for publish and subscribe operations. This parameter is valid only for topic connection factories.
brokerPubQueue	Specifies the queue to send publication messages to when using queue based brokering. This parameter is valid only for topic connection factories.
ccdtQmgrName	Specifies a queue manager name that is used to select one or more entries from a client channel definition table.

Table 295. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
ccdtUrl	Specifies a URL to a client channel definition table. Use this attribute for this connection factory, when contacting the WebSphere MQ messaging provider. Connection factories created using this attribute are ccdtURL connection factories.
ccsid	Specifies the coded character set ID to use on connections.
cleanupInterval	Specifies the interval between background executions of the publish and subscribe cleanup utility. This parameter is valid only for topic connection factories.
cleanupLevel	Specifies the cleanup Level for BROKER or MIGRATE subscription stores. This parameter is valid only for topic connection factories.
clientId	Specifies the client identifier used for connections started using this connection factory.
clonedSubs	Specifies whether two or more instances of the same durable topic subscriber can run simultaneously. This parameter is valid only for topic connection factories.
compressHeaders	Determines if message headers are compressed or not.
compressPayload	Determines if message payloads are compressed or not.
containerAuthAlias	Specifies a container managed authentication alias that has security credentials that are used for establishing a connection to the WebSphere MQ messaging provider.
description	Specifies a description of the connection factory.
failIfQuiescing	Specifies the behavior of certain calls to the queue manager when the queue manager is put into a quiescent state.
localAddress	Specifies either or both of the following items: • The local network interface to be used. • The local port, or range of local ports, to be used.
mappingAlias	Specifies the JAAS mapping alias to use when determining the security related credentials. The security related credentials are used when establishing a connection to the WebSphere MQ.
modelQueue	Specifies the WebSphere MQ model queue definition to use as a basis when creating JMS temporary destinations. This parameter is valid only for queue connection factories.
msgRetention	Specifies whether the connection consumer keeps unwanted messages on the input queue. A value of `true` means that it does. A value of `false` means that the messages are disposed of based on their disposition options. This parameter is valid only for queue connection factories.
msgSelection	Specifies where message selection occurs. This parameter is valid only for topic connection factories.
pollingInterval	Specifies in milliseconds the maximum time that elapses during a polling interval. If a message listener within a session has no suitable message on its queue, the message listener uses the polling interval to determine how often to poll its queue for a message. Increase the value for this property if sessions frequently do not have a suitable message available. This attribute is applicable only in the client container.
providerVersion	Specifies the minimum version and capabilities of the queue manager.
pubAckInterval	Specifies the number of publications to send to a queue based broker before sending a publication which solicits an acknowledgement. This attribute is valid only for topic connection factories.
qmgrHostname	Specifies the hostname that this connection factory uses when attempting a client mode connection to WebSphere MQ.
qmgrName	Specifies the queue manager name that this connection factory uses when contacting the WebSphere MQ messaging provider. Connection factories created using this parameter are user-defined connection factories.
qmgrPortNumber	Specifies the port number that this connection factory uses when attempting a client mode connection to WebSphere MQ.
qmgrSvrconnChannel	Specifies the SVRCONN channel to use when connecting to WebSphere MQ. Connection factories created using this parameter are user-defined connection factories.
rcvExitInitData	Specifies initialization data to pass to the receive exit.
rcvExit	Specifies a comma separated list of receive exit class names.
replyWithRFH2	Specifies whether, when replying to a message, an RFH version 2 header is included in the reply message. This parameter is valid only for queue connection factories.
rescanInterval	Specifies in milliseconds the maximum time that elapses during a scanning interval. When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, the WebSphere MQ JMS client searches the WebSphere MQ queue for suitable messages in the sequence determined by the MsgDeliverySequence attribute of the queue. When the client finds a suitable message and delivers it to the consumer, the client resumes the search for the next suitable message from its current position in the queue. The client continues to search the queue until it reaches the end of the queue, or until the interval of time specified by this property has expired. In each case, the client returns to the beginning of the queue to continue its search, and a new time interval starts. This parameter is only valid for queue connection factories.
secExitInitData	Specifies initialization data to pass to the security exit.

Table 295. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
secExit	Specifies a comma separated list of security exit class names.
sendExitInitData	Specifies initialization data to pass to the send exit.
sendExit	Specifies a comma separated list of send exit class names.
sparseSubs	Specifies the message retrieval policy of a TopicSubscriber object. This parameter is only valid for topic connection factories.
sslConfiguration	Specifies a specific Secure Sockets Layer (SSL) configuration to secure network connections to the queue manager.
sslCrl	Specifies a list of LDAP servers which can be used to provide certificate revocation information if this connection factory establishes an SSL connection to WebSphere MQ.
sslPeerName	Specifies a peer name to match against the distinguished name in the peer certificate. This attribute is used when the connection factory establishes an SSL connection to the queue manager.
sslResetCount	Specifies the number of bytes to transfer before resetting the symmetric encryption key used for the SSL session. This attribute is used when the connection factory establishes an SSL connection to the queue manager.
sslType	Specifies the configuration, if any, when the connection factory establishes an SSL connection to the queue manager.
stateRefreshInt	Specifies in milliseconds the maximum time that elapses between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if subStore attribute has the value `QUEUE`. This attribute is valid only for topic connection factories.
subStore	Specifies that WebSphere MQ JMS stores persistent data relating to active subscriptions. This attribute is valid only for topic connection factories.
support2PCProtocol	Specifies whether the connection factory acts as a resource which is capable of participating in distributed two phase commit processing.
tempQueuePrefix	Specifies the prefix to apply to WebSphere MQ temporary queues. These temporary queues represent JMS temporary queue type destinations. This attribute is valid only for queue connection factories.
tempTopicPrefix	Specifies the prefix to apply to WebSphere MQ temporary topics. These temporary topics represent JMS temporary topic type destinations. This attribute is valid only for topic connection factories.
wildcardFormat	Specifies which sets of characters are interpreted as topic wild cards. This attribute is valid only for topic connection factories.
wmqTransportType	Specifies how this connection factory connects to WebSphere MQ. Connection factories created using this attribute are user-defined. Valid values are `BINDINGS`, `BINDINGS_THEN_CLIENT`, and `CLIENT`.
xaRecoveryAuthAlias	Specifies the authentication alias to connect to WebSphere MQ for XA recovery.

Syntax

```
AdminJMS.createWMQTopicConnectionFactory(scope,
 name, jndiName,
 attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createWMQTopicConnectionFactory("myScope", "myName", "myJNDIName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQTopicConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName6", "maxBatchSize=15,description=my description")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWMQTopicConnectionFactory
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName6", [['maxBatchSize', '15'], ['description', 'my description']])
```

createSIBJMSActivationSpec

This script creates a new JMS activation specification for the default messaging provider at the scope that you specify. The script returns the configuration ID of the created SIB JMS activation specification.

To run the script, specify the scope, activation specification name, JNDI name, and the JNDI name destination arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 296. Arguments for the createSIBJMSActivationSpec script. Run the script to create a JMS activation specification.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the default messaging provider at which the JMS activation specification is to be created.
name	Specifies the name assigned to this activation specification.
jndiName	Specifies the JNDI name that is specified in the bindings for message-driven beans associated with this activation specification.
destinationJndiName	Specifies the JNDI name of the destination JMS queue or topic that the message-driven bean uses.
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 297. Optional attributes. Additional attributes available for the script.

Attributes	Description
description	Specifies the JMS activation specification that the default messaging provider uses to validate the activation-configuration properties for a JMS message-driven bean (MDB).
acknowledgeMode	Specifies how the session acknowledges any messages it receives.
authenticationAlias	Specifies the name of a J2C authentication alias used for component-managed authentication of connections to the service integration bus.
busName	Name of the service integration bus to which connections are made.
clientId	Specifies the JMS client identifier. The client identifier is required for durable topic subscriptions.
destinationType	Specifies whether the message-driven bean uses a queue or topic destination.
durableSubscriptionHome	Specifies the name of the durable subscription home. This attribute identifies the messaging engine where all durable subscriptions accessed through this activation specification are managed.
maxBatchSize	Specifies the maximum number of messages received from the messaging engine in a single batch.
maxConcurrency	Specifies the maximum number of endpoints to which messages are delivered concurrently.
messageSelector	Specifies the JMS message selector used to determine which messages the message-driven bean (MDB) receives.
password	Specifies the password for Java 2 connector security to use.
subscriptionDurability	Specifies whether a JMS topic subscription is durable or nondurable.
subscriptionName	Specifies the subscription name needed for durable topic subscriptions.
shareDurableSubscriptions	Specifies how durable subscriptions are shared.
userName	Specifies the user identify for the Java 2 connector security to use.
readAhead	Specifies the read-ahead value. Valid values are `Default`, `AlwaysOn`, and `AlwaysOff`.
target	Specifies the new target value of the SIB JMS activation specification.
targetType	Specifies the new target value of the SIB JMS activation specification. Valid values are `BusMember`, `Custom`, and `ME`.
targetSignificance	Specifies the significance of the target group.
targetTransportChain	Specifies the name of the protocol that used to connect to a remote messaging engine.
providerEndPoints	Specifies a comma-separated list of endpoint triplets of the form *host:port:chain*.
shareDataSourceWithCMP	Specifies how data sources are shared.
consumerDoesNotModifyPayloadAfterGet	Specifies, when enabled, that object messages received through this activation specification only have their message data serialized by the system when necessary. Applications that obtain data from these messages must be treated as read-only. Valid values are `true` and `false`. The default is `false`.
forwarderDoesNotModifyPayloadAfterSet	Specifies, when enabled, that object or byte messages forwarded through this activation specification that have their payload modified will not have the data copied when the data is sent in the message. The system serializes the message data only when necessary. Applications sending such messages must not modify the data after they send it in the message. Valid values are `true` and `false`. The default is `false`.

Table 297. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
alwaysActivateAllMDBs	Specifies the MDB server-selection rule. The rule determines which servers can drive MDBs deployed to them. Valid values are `true` and `false`. The default is `false`. Specify `true` to activate MDBs in all servers. Otherwise, only servers with a running messaging engine are used.
retryInterval	Specifies the delay in seconds between attempts to connect to a messaging engine, both for the initial connection and for any subsequent attempts to establish a better connection. The default is `30`. The delay must be greater than zero.
autoStopSequentialMessageFailure	Specifies that the endpoint is stopped when the number of sequentially failing messages reaches the configured limit. Due to processing dependencies in the MDB, the actual number of messages processed might exceed this value.
failingMessageDelay	Specifies the period of time that passes before a message can be retried. A message is retried when the MDB fails to process it, but the message has not reached its maximum failed delivery limit. Other messages can be tried during this period, unless the sequential failure threshold and the maximum concurrency are set to 1.

Syntax

```
AdminJMS.createSIBJMSActivationSpec(scope,
 name, jndiName,
 destinationJndiName, attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSActivationSpec("myScope", "myName",
 "myJNDIName", "myDestinationName")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createSIBJMSActivationSpec(
"server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName11", "myDestinationJndiName11", "readAhead=AlwaysOff,maxBatchSize=54")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createSIBJMSActivationSpec
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName11", "myDestinationJndiName11", [['readAhead', 'AlwaysOff'], ['maxBatchSize', '54']])
```

createWMQActivationSpec

This script creates a new activation specification for the WebSphere MQ messaging provider at the scope that you specify. The script returns the configuration ID of the created WebSphere MQ activation specification.

To run the script, specify the scope, activation specification name, JNDI name, and the JNDI name destination arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 298. Arguments for the createWMQActivationSpec script. Run the script to create an activation specification.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the activation specification of the WebSphere MQ messaging provider.
name	Specifies the name assigned to this activation specification.
jndiName	Specifies the name and location used to bind this object into WebSphere Application Server JNDI.
destinationJndiName	Specifies the JNDI name of a WebSphere MQ messaging provider queue or topic type destination. When an MDB is deployed with this activation specification, messages for the MDB are consumed from this destination.
destinationType	Specifies the type of the destination. Valid values are `javax.jms.Queue` and `javax.jms.Topic`. The argument has no default value.

Table 298. Arguments for the createWMQActivationSpec script (continued). Run the script to create an activation specification.

Argument	Description
attributes	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 299. Optional attributes. Additional attributes available for the script.

Attributes	Description
authAlias	Specifies the authentication alias used to obtain the credentials that this activation specification needs to establish a connection to WebSphere MQ.
brokerCCDurSubQueue	Specifies the name of the queue from which a connection consumer receives durable subscription messages.
brokerCCSubQueue	Specifies the name of the queue from which non-durable subscription messages are retrieved for a connection consumer.
brokerCtrlQueue	Specifies the broker control queue to use when this activation specification subscribes to a topic.
brokerQmgr	Specifies the name of the queue manager on which the broker is running.
brokerSubQueue	Specifies the queue for obtaining subscription messages if this activation subscribes to a topic.
brokerVersion	Specifies the level of functionality required for publish and subscribe operations.
ccdtQmgrName	Specifies the name of the queue manager that selects one or more entries from a client channel definition table.
ccdtUrl	Specifies the URL to client channel definition table. Use this attribute for this activation specification when contacting WebSphere MQ. Activation specifications created using this attribute are ccdtURL activation specifications.
ccsid	Specifies the coded character set ID used on connections.
cleanupInterval	Specifies the interval between background executions of the publish and subscribe cleanup utility.
cleanupLevel	Specifies the cleanup level for broker or migrate subscription stores.
clientId	Specifies the client identifier for connections started with this activation specification.
clonedSubs	Specifies whether two or more instances of the same durable topic subscriber can run simultaneously.
compressHeaders	Specifies whether message headers are compressed.
compressPayload	Specifies whether message payloads are compressed.
description	Specifies an administrative description assigned to the activation specification.
failIfQuiescing	Specifies the behavior of certain calls to the queue manager when the queue manager is put into a quiescent state.
failureDeliveryCount	Specifies the number of sequential delivery failures that are allowed before the endpoint is suspended.
maxPoolSize	Specifies the maximum number of server sessions in the server session pool that the connection consumer uses.
messageSelector	Specifies which messages are delivered.
msgRetention	Specifies whether the connection consumer keeps unwanted messages on the input queue. A value of `true` means that it does. A value of `false` means that the messages are disposed of based on the disposition options.
msgSelection	Specifies where message selection occurs.
poolTimeout	Specifies the period of time, in milliseconds, that an unused server session is held open in the server session pool before being closed due to inactivity.
providerVersion	Specifies the minimum version and capabilities of the queue manager.
qmgrHostname	Specifies the hostname which used for this activation specification, when attempting a client mode connection to WebSphere MQ.
qmgrName	Specifies the name of the queue manager for this activation specification, when contacting WebSphere MQ. Activation specifications created using this attribute are user defined.
qmgrPortNumber	Specifies the port number for this activation specification, when attempting a client mode connection to WebSphere MQ.
qmgrSvrconnChannel	Specifies the SVRCONN channel to use when connecting to WebSphere MQ. Activation specifications created using this attribute are user defined.
rcvExitInitData	Specifies initialization data to pass to the receive exit.

Table 299. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
rcvExit	Specifies a comma separated list of receive exit class names.
rescanInterval	Specifies in milliseconds the maximum time that elapses during a scanning interval. When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, the WebSphere MQ JMS client searches the WebSphere MQ queue for suitable messages in the sequence determined by the MsgDeliverySequence attribute of the queue. When the client finds a suitable message and delivers it to the consumer, the client resumes the search for the next suitable message from its current position in the queue. The client continues to search the queue until it reaches the end of the queue, or until the interval of time specified by this property has expired. In each case, the client returns to the beginning of the queue to continue its search, and a new time interval starts.
secExitInitData	Specifies initialization data to pass to the security exit.
secExit	Specifies a security exit class name.
sendExitInitData	Specifies initialization data to pass to the send exit.
sendExit	Specifies a comma separated list of class names for the send exit.
sparseSubs	Specifies the message retrieval policy of a topic subscriber object.
sslConfiguration	Specifies the SSL configuration used to secure network connections to the queue manager.
sslCrl	Specifies a list of LDAP servers which can be used to provide certificate revocation information if this activation specification establishes an SSL connection to WebSphere MQ.
sslPeerName	Specifies a value to match against the distinguished name in the peer certificate. This attribute is used when the activation specification establishes an SSL connection to the queue manager.
sslResetCount	Specifies how many bytes to transfer before resetting the symmetric encryption key for the SSL session. This attribute is used when the activation specification establishes an SSL connection to the queue manager.
sslType	Specifies the SSL configuration for the network connection to the queue manager.
startTimeout	Specifies the period of time in milliseconds within which delivery of a message to an MDB must start after the work to deliver the message has been scheduled. If this period of time elapses, the message is returned to the queue.
stateRefreshInt	Specifies the interval in milliseconds between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This attribute is relevant only if the subStore attribute has the value `QUEUE`.
stopEndpointIfDeliveryFails	Specifies whether the endpoint is stopped if message delivery fails the number of times specified by the failureDeliveryCount attribute.
subscriptionDurability	Specifies whether a durable or nondurable subscription is used to deliver messages to an MDB subscribing to the topic.
subscriptionName	Specifies the name of the durable subscription.
subStore	Specifies where WebSphere MQ JMS stores persistent data relating to active subscriptions.
wildcardFormat	Specifies which sets of characters are interpreted as topic wild cards.
wmqTransportType	Specifies the manner in which, for this activation specification, a connection is established with WebSphere MQ. Activation specifications created using this attribute are user defined. Valid values are `BINDINGS`, `BINDINGS_THEN_CLIENT` and `CLIENT`.
localAddress	Specifies either or both of the following options: • The local network interface to be used • The local port, or range of local ports, to be used

Syntax

```
AdminJMS.createSIBJMSActivationSpec(scope,name, jndiName,
 destinationJndiName, destinationType,attributes)
```

Example usage

The following example script contains required attributes only:

```
AdminJMS.createSIBJMSActivationSpec("myScope", "myName",
 "myJNDIName", "myDestinationName", "myDestinationType")
```

The following example script includes optional attributes in a string format:

```
AdminJMS.createWMQActivationSpec
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName12", "myDestinationJndiName12", "javax.jms.Topic", "poolTimeout=2222,startTimeout=9999")
```

The following example script includes optional attributes in a list format:

```
AdminJMS.createWMQActivationSpec
("server1(cells/avmoghe01Cell02/nodes/avmoghe01Node02/servers/server1|server.xml#Server_1237476439906)",
"myName" , "myJndiName12", "myDestinationJndiName12", "javax.jms.Topic", [['poolTimeout', '2222'],
['startTimeout', '9999']])
```

startListenerPort

This script starts a listener port in your environment. The script returns a value of `1` if the system successfully starts the listener port or a value of `-1` if the system does not start the listener port.

To run the script, specify the node and server name arguments, as defined in the following table:

Table 300. Arguments for the startListenerPort script. Run the script to start a listener port.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Syntax

```
AdminJMS.startListenerPort(nodeName,
 serverName)
```

Example usage

```
AdminJMS.startListenerPort("myNode", "myServer")
```

JMS query scripts

The scripting library provides many script procedures to manage your Java Messaging Service (JMS) configurations. This topic provides usage information for scripts that retrieve configuration IDs from your JMS configuration. You can run each script individually or combine many procedures to create custom automation scripts for your environment.

Each JMS management script procedure is located in the *app_server_root*/scriptLibraries/resources/JMS/V70 directory.

Beginning with Version 7, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/`scriptLibraries` directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Fast path: Beginning with Fix Pack 5, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the cell, node, server, or cluster scope. Resource providers include mail providers, URL providers, and resource environment providers. You do not have to write custom scripts to configure resources at a particular scope.

Use the following script procedures to query your JMS configurations:

listGenericJMSConnectionFactories

This script displays a list of configuration IDs for the generic JMS connection factories configured in your environment.

The script does not require any input parameters.

Table 301. Argument for the listGenericJMSConnectionFactories script. Run the script to list generic JMS connection factories.

Argument	Description
connFactoryName	Optionally specifies the name of the generic JMS connection factory of interest.

Syntax

```
AdminJMS.listGenericJMSConnectionFactories(connFactoryName)
```

Example usage

```
AdminJMS.listGenericJMSConnectionFactories()
AdminJMS.listGenericJMSConnectionFactories("JMSCFTest")
```

listGenericJMSConnectionFactoryTemplates

This script displays a list of generic JMS connection factory template configuration ids.

The script does not require any input parameters. However, to return a specific generic JMS connection factory template, specify the template ID argument, as defined in the following table:

Table 302. Argument for the listGenericJMSConnectionFactoryTemplates script. Run the script to list generic JMS connection factory templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listGenericJMSConnectionFactoryTemplates(templateName)
```

Example usage

```
AdminJMS.listGenericJMSConnectionFactoryTemplates()
AdminJMS.listGenericJMSConnectionFactoryTemplates("Generic QueueConnectionFactory for Windows")
```

listGenericJMSDestinations

This script displays a list of configuration IDs for the generic JMS destinations configured in your environment. The script does not require any input parameters. However, to return a specific generic JMS destination, specify the generic JMS destination name.

The script does not require any input parameters. However, to return a specific generic JMS destination, specify the generic JMS destination name, as defined in the following table:

Table 303. Argument for the listGenericJMSDestinations script. Run the script to list generic JMS destinations.

Argument	Description
destinationName	Optionally specifies the name of the generic JMS destination of interest.

Syntax

```
AdminJMS.listGenericJMSDestinations(destinationName)
```

Example usage

```
AdminJMS.listGenericJMSDestinations()
AdminJMS.listGenericJMSDestinations("JMSDestination")
```

listGenericJMSDestinationTemplates

This script displays a list of generic JMS destination template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Table 304. Argument for the listGenericJMSDestinationTemplates script. Run the script to list generic JMS destination templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listGenericJMSDestinationTemplates(templateName)
```

Example usage

```
AdminJMS.listGenericJMSDestinationTemplates()
AdminJMS.listGenericJMSDestinationTemplates("Example.JMS.Generic.Win.Topic")
```

listJMSProviders

This script displays a list of configuration IDs for the JMS providers that are configured in your environment.

The script does not require any input parameters. However, to return a specific JMS provider, specify the JMS provider name, as defined in the following table:

Table 305. Argument for the listJMSProviders script. Run the script to list JMS providers.

Argument	Description
jmsProviderName	Optionally specifies the name of the generic JMS connection factory of interest.

Syntax

```
AdminJMS.listJMSProviders(jmsProviderName)
```

Example usage

```
AdminJMS.listJMSProviders()
AdminJMS.listJMSProviders("JMSTest")
```

listJMSProviderTemplates

This script displays a list of JMS provider template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Table 306. Argument for the listJMSProviderTemplates script. Run the script to list JMS provider templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listJMSProviderTemplates(templateName)
```

Example usage

```
AdminJMS.listJMSProviderTemplates()
AdminJMS.listJMSProviderTemplates("WebSphere JMS Provider")
```

listWASQueueConnectionFactoryTemplates

This script displays a list of JMS queue connection factory template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Table 307. Argument for the listWASQueueConnectionFactoryTemplates script. Run the script to list JMS queue connection factory templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASQueueConnectionFactoryTemplates(templateName)
```

Example usage

```
AdminJMS.listWASQueueConnectionFactoryTemplates()
AdminJMS.listWASQueueConnectionFactoryTemplates("Example WAS QueueConnectionFactory")
```

listWASQueueTemplates

This script displays a list of JMS queue template configuration ids.

The script does not require any input parameters. However, to return a specific generic template, specify the template name, as defined in the following table:

Table 308. Argument for the listWASQueueTemplates script. Run the script with the template name argument.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASQueueTemplates(templateName)
```

Example usage

```
AdminJMS.listWASQueueTemplates()
AdminJMS.listWASQueueTemplates("Example.JMS.WAS.Q1")
```

listWASTopicConnectionFactoryTemplates

This script displays a list of JMS topic connection factory template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Table 309. Argument for the listWASTopicConnectionFactoryTemplates script. Run the script to list JMS topic connection factory templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASTopicConnectionFactoryTemplates(templateName)
```

Example usage

```
AdminJMS.listWASTopicConnectionFactoryTemplates()
AdminJMS.listWASTopicConnectionFactoryTemplates("First Example WAS TopicConnectionFactory")
```

listWASQueueConnectionFactories

This script displays a list of configuration IDs for the JMS queue connection factories configured in your environment.

The script does not require any input parameters. However, to return a specific JMS queue connection factory, specify the connection factory name, as defined in the following table:

Table 310. Argument for the listWASQueueConnectionFactories script. Run the script to list JMS queue connection factories.

Argument	Description
connFactoryName	Optionally specifies the name of the JMS connection factory of interest.

Syntax

```
AdminJMS.listWASQueueConnectionFactories(connFactoryName)
```

Example usage

```
AdminJMS.listWASQueueConnectionFactories()
AdminJMS.listWASQueueConnectionFactories("queuecf")
```

listWASQueues

This script displays a list of JMS queues.

The script does not require any input parameters. However, to return a specific queue, specify the queue name, as defined in the following table:

Table 311. Argument for the listWASQueues script. Run the script to list JMS queues.

Argument	Description
queueName	Optionally specifies the name of the queue of interest.

Syntax

```
AdminJMS.listWASQueues(queueName)
```

Example usage

```
AdminJMS.listWASQueues()
AdminJMS.listWASQueues("WASQueueTest")
```

listWASTopicConnectionFactories

This script displays a list of configuration IDs for the JMS topic connection factories configured in your environment.

The script does not require any input parameters. However, to return a specific JMS topic connection factory, specify the connection factory name, as defined in the following table:

Table 312. Argument for the listWASTopicConnectionFactories script. Run the script to list JMS topic connection factories.

Argument	Description
connFactoryName	Optionally specifies the name of the JMS topic connection factory of interest.

Syntax

```
AdminJMS.listWASTopicConnectionFactories(connFactoryName)
```

Example usage

```
AdminJMS.listWASTopicConnectionFactories()
AdminJMS.listWASTopicConnectionFactories("TopicCFTest")
```

listWASTopics

This script displays a list of configuration IDs for the JMS topics configured in your environment.

The script does not require any input parameters. However, to return a specific topic, specify the topic name, as defined in the following table:

Table 313. Argument for the listWASTopics script. Run the script to list JMS topics.

Argument	Description
topicName	Optionally specifies the name of the topic of interest.

Syntax

```
AdminJMS.listWASTopics(topicName)
```

Example usage

```
AdminJMS.listWASTopics()
AdminJMS.listWASTopics("TopicTest")
```

listWASTopicTemplates

This script displays a list of JMS topic template configuration ids.

The script does not require any input parameters. However, to return a specific template, specify the template name, as defined in the following table:

Table 314. Argument for the listWASTopicTemplates script. Run the script to list JMS topic templates.

Argument	Description
templateName	Optionally specifies the name of the template of interest.

Syntax

```
AdminJMS.listWASTopicTemplates(templateName)
```

Example usage

```
AdminJMS.listWASTopicTemplates()
AdminJMS.listWASTopicTemplates("Example.JMS.WAS.T1")
```

Automating authorization group configurations using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the authorization groups scripts create, configure, remove and query your authorization group configuration.

Before you begin

Before you can complete this task, you must install an application server in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
      "myNode", "Server1")

  # Add a second member to the cluster
  AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

  # Install an application
  AdminApplication.installAppWithClusterOption("DefaultApplication",
      "..\installableApps\DefaultApplication.ear", "myCluster")

  # Start all servers and applications on the node
  AdminServerManagement.startAllServers("myNode")
  ```

  Save the custom script and run it from the command line, as the following syntax demonstrates:

  ```
  bin>wsadmin -language jython -f path/to/your/jython/file.py
  ```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The authorization group management procedures in scripting library are located in the *app_server_root*/`scriptLibraries/security/V70` subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/`scriptLibraries` directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminAuthorizations.py scripts to perform multiple combinations of authorization group administration functions. This topic provides one sample combination of procedures. Use the following

steps to create an authorization group, adds resources to the group, and assigns user roles.

Procedure

1. Optional: Start the wsadmin scripting tool.

   Use this step to launch the wsadmin tool and connect to a server. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts. Alternatively, you can run each script individually without launching the wsadmin tool.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
bin>wsadmin -lang jython
```

   When the wsadmin tool launches, the system loads each script from the scripting library.

2. Create an authorization group.

   Use the createAuthorizationGroup script to create a new authorization group in your configuration, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminAuthorizations.createAuthorizationGroup("myAuthGroup")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminAuthorizations.createAuthorizationGroup("myAuthGroup")
```

3. Add resources to the new authorization group.

   Use the addResourceToAuthorizationGroup script to add resources. You can create a file-grained administrative authorization groups by selecting administrative resources to be part of the authorization group, as the following example demonstrates:

```
bin>wsadmin -lang jython -c "AdminAuthorizations.addResourceToAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminAuthorizations.addResourceToAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")
```

4. Assign users to the administrative role for the authorization group.

   Use the mapUsersToAdminRole script to assign one or more users to the administrative role for the resources in the authorization group. You can assign users for the authorization group to the administrator, configurator, deployer, operator, monitor, adminsecuritymanager, and iscadmins administrative roles. The following example maps the `user01`, `user02`, and `user03` users as administrators for the resources in the authorization group:

```
bin>wsadmin -lang jython -c "AdminAuthorizations.mapUsersToAdminRole("myAuthGroup", "administrator", "user01 user02 user03")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminAuthorizations.mapUsersToAdminRole("myAuthGroup", "administrator", "user01 user02 user03")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/`scriptLibraries` directory.

Authorization group configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the scripts in this topic to create, configure, remove and query your security authorization group configuration. You can run each script individually or combine procedures to create custom automation scripts.

The AdminAuthorizations script procedures are located in the *app_server_root*/scriptLibraries/security/V70 directory.

Use the following script procedures to configure authorization groups:

- "addResourceToAuthorizationGroup"
- "createAuthorizationGroup" on page 281
- "mapGroupsToAdminRole" on page 281
- "mapUsersToAdminRole" on page 281

Use the following script procedures to remove users and groups from the security authorization settings:

- "deleteAuthorizationGroup" on page 282
- "removeGroupFromAllAdminRoles" on page 282
- "removeGroupsFromAdminRole" on page 282
- "removeResourceFromAuthorizationGroup" on page 283
- "removeUserFromAllAdminRoles" on page 283
- "removeUsersFromAdminRole" on page 283

Use the following script procedures to query your security authorization group configuration:

- "help" on page 283
- "listAuthorizationGroups" on page 284
- "listAuthorizationGroupsForUserID" on page 284
- "listAuthorizationGroupsForGroupID" on page 284
- "listAuthorizationGroupsOfResource" on page 284
- "listUserIDsOfAuthorizationGroup" on page 285
- "listGroupIDsOfAuthorizationGroup " on page 285
- "listResourcesOfAuthorizationGroup" on page 285
- "listResourcesForUserID " on page 286

addResourceToAuthorizationGroup

This script adds a resource to an existing authorization group in your configuration. You can create a fine-grained administrative authorization groups by selecting administrative resources to be part of the authorization group. You can assign users or groups to this new administrative authorization group and also give them access to the administrative resources contained within.

Table 315. addResourceToAuthorizationGroup argument descriptions. Run the script with the authorization group name and resource name to add a resource to an authorization group.

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
resource	Specifies the name of the resource to add to the authorization group of interest.

Syntax

```
AdminAuthorizations.addResourceToAuthorizationGroup(authGroupName, resource)
```

Example usage

```
AdminAuthorizations.addResourceToAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")
```

createAuthorizationGroup

This script creates a new authorization group in your configuration. Administrative authorization groups that specify users and groups that have certain authorities with the selected resources.

Table 316. createAuthorizationGroup argument description. Run the script with the authorization group name argument to create an authorization group.

Argument	Description
authGroupName	Specifies the name of the authorization group to create.

Syntax

```
AdminAuthorizations.createAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.createAuthorizationGroup("myAuthGroup")
```

mapGroupsToAdminRole

This script maps group IDs to one or more administrative roles in the authorization group. The name of the authorization group that you provide determines the authorization table. The group ID can be a short name or fully qualified domain name in case Lightweight Directory Access Protocol (LDAP) user registry is used.

Table 317. mapGroupsToAdminRole argument descriptions. Run the script with the authorization group name, administrative role, and group ID arguments.

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role to which the system maps the user IDs.
groupIDs	Specifies the group IDs to map to the role and authorization group.

Syntax

```
AdminAuthorizations.mapGroupsToAdminRole(authGroupName, adminRole, groupIDs)
```

Example usage

```
AdminAuthorizations.mapGroupsToAdminRole("myAuthGroup", "administrator", "group01 group02 group03")
```

mapUsersToAdminRole

This script maps user IDs to one or more administrative roles in the authorization group. The name of the authorization group that you provide determines the authorization table. The user ID can be a short name or fully qualified domain name in case LDAP user registry is used.

Table 318. mapUsersToAdminRole argument descriptions. Run the script with the authorization group name, administrative role, and user ID arguments.

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role to which the system maps the user IDs.
userIDs	Specifies the user IDs to map to the role and authorization group.

Syntax

```
AdminAuthorizations.mapUsersToAdminRole(authGroupName, adminRole, userIDs)
```

Example usage

```
AdminAuthorizations.mapUsersToAdminRole("myAuthGroup", "administrator", "user01 user02 user03")
```

deleteAuthorizationGroup

This script removes an authorization group from your security configuration.

Table 319. deleteAuthorizationGroup argument descriptions. Run the script with the authorization group argument.

Argument	Description
authGroupName	Specifies the name of the authorization group to delete.

Syntax

```
AdminAuthorizations.deleteAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.deleteAuthorizationGroup("myAuthGroup")
```

removeGroupFromAllAdminRoles

This script removes a specific group from an administrative role in each authorization group in your configuration.

Table 320. removeGroupFromAllAdminRoles argument description. Run the script with the group ID argument.

Argument	Description
groupID	Specifies the group ID to remove from the administrative role in each authorization group in your configuration.

Syntax

```
AdminAuthorizations.removeGroupFromAllAdminRoles(groupID)
```

Example usage

```
AdminAuthorizations.removeGroupFromAllAdminRoles("group01")
```

removeGroupsFromAdminRole

This script removes specific groups from an administrative role in the authorization group of interest.

Table 321. removeGroupsFromAdminRole argument descriptions. Run the script with the authorization group name, administrative role, and group ID arguments.

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role from which to remove the user IDs.
groupIDs	Specifies the group IDs to remove from the specific role in the authorization group.

Syntax

```
AdminAuthorizations.removeUsersFromAdminRole(authGroupName, adminRole, groupIDs)
```

Example usage

```
AdminAuthorizations.removeUsersFromAdminRole("myAuthGroup", "administrator", "group01 group02 group03")
```

removeResourceFromAuthorizationGroup

This script removes a specific resource from the authorization group of interest.

Table 322. removeResourceFromAuthorizationGroup argument descriptions. Run the script with the authorization group name and resource name arguments.

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
resource	Specifies the name of the resource to remove.

Syntax

```
AdminAuthorizations.removeResourceFromAuthorizationGroup(authGroupName, resource)
```

Example usage

```
AdminAuthorizations.removeResourceFromAuthorizationGroup("myAuthGroup", "Node=myNode:Server=myServer")
```

removeUserFromAllAdminRoles

This script removes a specific user from an administrative role in each authorization group in your configuration.

Table 323. removeUserFromAllAdminRoles argument description. Run the script with the user ID argument.

Argument	Description
userID	Specifies the user ID to remove from the administrative role in each authorization group in your configuration.

Syntax

```
AdminAuthorizations.removeUserFromAllAdminRoles(userID)
```

Example usage

```
AdminAuthorizations.removeUserFromAllAdminRoles("user01")
```

removeUsersFromAdminRole

This script removes specific users from an administrative role in the authorization group of interest.

Table 324. removeUsersFromAdminRole argument descriptions. Run the script to remove users from an administrative role.

Argument	Description
authGroupName	Specifies the name of the authorization group of interest.
adminRole	Specifies the name of the administrative role from which to remove the user IDs.
userIDs	Specifies the user IDs to remove from the specific role in the authorization group.

Syntax

```
AdminAuthorizations.removeUsersFromAdminRole(authGroupName, adminRole, userIDs)
```

Example usage

```
AdminAuthorizations.removeUsersFromAdminRole("myAuthGroup", "administrator", "user01 user02 user03")
```

help

Table 325. help argument description. Run the help script to display the script procedures that the AdminClusterManagement script library supports. Specify the name of the script of interest.

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminResources.help(script)
```

Example usage

```
AdminResources.help("listAuthorizationGroups")
```

listAuthorizationGroups

This script displays each authorization group in your security configuration. This script does not require arguments.

Syntax

```
AdminAuthorizations.listAuthorizationGroups()
```

Example usage

```
AdminAuthorizations.listAuthorizationGroups()
```

listAuthorizationGroupsForUserID

This script displays each authorization group to which a specific user ID has access.

Table 326. listAuthorizationGroupsForUserID argument description. Run the script with the user ID argument.

Argument	Description
userID	Specifies the user ID for which to display authorization groups.

Syntax

```
AdminAuthorizations.listAuthorizationGroupsForUserID(userID)
```

Example usage

```
AdminAuthorizations.listAuthorizationGroupsForUserID("user01")
```

listAuthorizationGroupsForGroupID

This script displays each authorization group to which a specific group ID has access.

Table 327. listAuthorizationGroupsForGroupID argument description. Run the script with the group ID argument.

Argument	Description
groupID	Specifies the group ID for which to display authorization groups.

Syntax

```
AdminAuthorizations.listAuthorizationGroupsForGroupID(groupID)
```

Example usage

```
AdminAuthorizations.listAuthorizationGroupsForGroupID("group01")
```

listAuthorizationGroupsOfResource

This script displays each authorization group to which a specific resource is mapped.

Table 328. listAuthorizationGroupsOfResource argument description. Run the script with the resource name argument.

Argument	Description
resource	Specifies the resource of interest.

Syntax

```
AdminAuthorizations.listAuthorizationGroupsOfResource(resource)
```

Example usage

```
AdminAuthorizations.listAuthorizationGroupsOfResource("Node=myNode:Server=myServer")
```

listUserIDsOfAuthorizationGroup

This script displays the user IDs and access level that are associated with a specific authorization group.

Table 329. listUserIDsOfAuthorizationGroup argument description. Run the script with the authorization group name argument.

Argument	Description
authGroupname	Specifies the name of the authorization group of interest.

Syntax

```
AdminAuthorizations.listUserIDsOfAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.listUserIDsOfAuthorizationGroup("myAuthGroup")
```

listGroupIDsOfAuthorizationGroup

This script displays the group IDs and access level that are associated with a specific authorization group.

Table 330. listGroupIDsOfAuthorizationGroup argument description. Run the script with the authorization group name argument.

Argument	Description
authGroupname	Specifies the name of the authorization group of interest.

Syntax

```
AdminAuthorizations.listGroupIDsOfAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.listGroupIDsOfAuthorizationGroup("myAuthGroup")
```

listResourcesOfAuthorizationGroup

This script displays the resources that are associated with a specific authorization group.

Table 331. listResourcesOfAuthorizationGroup argument description. Run the script with the authorization group name argument.

Argument	Description
authGroupname	Specifies the name of the authorization group of interest.

Syntax

```
AdminAuthorizations.listResourcesOfAuthorizationGroup(authGroupName)
```

Example usage

```
AdminAuthorizations.listResourcesOfAuthorizationGroup("myAuthGroup")
```

listResourcesForUserID

This script displays the resources that a specific user ID can access.

Table 332. listResourcesForUserID argument description. Run the script with the user ID argument.

Argument	Description
userID	Specifies the user ID of interest.

Syntax

```
AdminAuthorizations.listResourcesForUserID(userID)
```

Example usage

```
AdminAuthorizations.listResourcesForUserID("user01")
```

listResourcesForGroupID

This script displays the resources that a specific group ID can access.

Table 333. listResourcesForGroupID argument description. Run the script with the group ID argument.

Argument	Description
groupID	Specifies the group ID of interest.

Syntax

```
AdminAuthorizations.listResourcesForGroupID(groupID)
```

Example usage

```
AdminAuthorizations.listResourcesForGroupID("group01")
```

Automating resource configurations using wsadmin scripting

The scripting library provides Jython script procedures to assist in automating your environment. Use the scripts in the AdminResources script library to configure mail, URL, and resource settings.

Before you begin

Before you can complete this task, you must install an application server in your environment.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
      "myNode", "Server1")

  # Add a second member to the cluster
  ```

```
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The resource management procedures in scripting library are located in the *app_server_root*/scriptLibraries/resource/V70 subdirectory. Each script from the directory automatically loads when you launch the wsadmin tool. To automatically load your own Jython scripts (*.py) when the wsadmin tool starts, create a new subdirectory and save existing automation scripts under the *app_server_root*/scriptLibraries directory.

Note: To create custom scripts using the scripting library procedures, save the modified scripts to a new subdirectory to avoid overwriting the library. Do not edit the script procedures in the scripting library.

You can use the AdminResources.py scripts to perform multiple combinations of administration functions. This topic provides one sample combination of procedures. See the documentation for the resource configuration scripts for additional scripts, argument descriptions, and syntax examples.

The example script in this topic configures a custom mail provider and session. A mail provider encapsulates a collection of protocol providers like SMTP, IMAP and POP3, while mail sessions authenticate users and controls users' access to messaging systems. Configure your own mail providers and sessions to customize how JavaMail is handled.

Procedure

1. Optional: Launch the wsadmin tool.

   Use this step to launch the wsadmin tool and connect to a server, job manager, or administrative agent profile, or run the tool in local mode. If you launch the wsadmin tool, use the interactive mode examples in this topic to run scripts.

   - Enter the following command from the bin directory to launch the wsadmin tool and connect to a server:

```
wsadmin -lang jython
```

   - Enter the following command from the bin directory to launch the wsadmin tool in local mode and using the Jython scripting language:

```
wsadmin -conntype none -lang jython
```

   When the wsadmin tool launches, the system loads all scripts from the scripting library.

2. Create a mail provider.

   Run the createMailProvider script from the AdminResources script library, specifying the node name, server name, and new mail provider name, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminResources.createMailProvider(myNode, myServer, newMailProvider)"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminResources.createMailProvider(nodeName, serverName, mailProviderName)
```

3. Define the protocol provider for the mail provider.

You can also configure custom properties, classes, JNDI name, and other mail settings with this script. See the documentation for the resource configuration scripts for argument descriptions and syntax examples. Run the configMailProvider script from the AdminResources script library to define the protocol provider, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminResources.configMailProvider(myNode, myServer, newMailProvider, "", "", "SOAP", "", "", "", "", "", "")"
```

You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminResources.configMailProvider(myNode, myServer, newMailProvider, "", "", "SOAP", "", "", "", "", "", "")
```

4. Create the mail session.

   Run the createMailSession script from the AdminResources script library, specifying the node name, server name, mail provider name, mail session name, and Java Naming and Directory Interface (JNDI) name arguments, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminResources.createMailSession("myNode", "myServer", "newMailProvider", "myMailSession", "myMailSession/jndi")"
```

   You can also use interactive mode to run the script procedure, as the following example demonstrates:

```
wsadmin>AdminResources.createMailSession("myNode", "myServer", "newMailProvider", "myMailSession", "myMailSession/jndi")
```

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

6. Synchronize the node.

   To propagate the configuration changes to the node, run the syncNode script procedure from the AdminNodeManagement script library, specifying the node of interest, as the following example demonstrates:

```
wsadmin -lang jython -c "AdminNodeManagement.syncNode("myNode")"
```

   You can also use interactive mode to run the script procedure, as the following example displays:

```
wsadmin>AdminNodeManagement.syncNode("myNode")
```

Results

The wsadmin script libraries return the same output as the associated wsadmin commands. For example, the AdminServerManagement.listServers() script returns a list of available servers. The AdminClusterManagement.checkIfClusterExists() script returns a value of `true` if the cluster exists, or `false` if the cluster does not exist. If the command does not return the expected output, the script libraries return a 1 value when the script successfully runs. If the script fails, the script libraries return a -1 value and an error message with the exception.

By default, the system disables failonerror option. To enable this option, specify `true` as the last argument for the script procedure, as the following example displays:

```
wsadmin>AdminApplication.startApplicationOnCluster("myApplication","myCluster","true")
```

What to do next

Create custom scripts to automate your environment by combining script procedures from the scripting library. Save custom scripts to a new subdirectory of the *app_server_root*/scriptLibraries directory.

Resource configuration scripts

The scripting library provides multiple script procedures to automate your application server configurations. Use the mail, URL, and resource environment configuration scripts to create and configure resources in your environment. You can run each script individually or combine procedures to create custom automation scripts.

The mail, URL, and resource management script procedures are located in the *app_server_root*/scriptLibraries/resources/V70 directory.

Beginning with Version 7, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the server scope. You can write your own custom scripts to configure resources at the cell, node, or cluster level.

Note: Do not edit the script procedures in the script library. To write custom script library procedures, use the scripts in the *app_server_root*/`scriptLibraries` directory as Jython syntax samples. Save the custom scripts to a new subdirectory to avoid overwriting the library.

Fast path: Beginning with Fix Pack 5, the Jython script library provides script functions for JDBC providers, JMS resources, and resource providers at the cell, node, server, or cluster scope. Resource providers include mail providers, URL providers, and resource environment providers. You do not have to write custom scripts to configure resources at a particular scope.

Attention: The example usage scripts and the script syntax are split on multiple lines for printing purposes.

Use the following script procedures to configure your mail settings:

Use the following script procedures to configure your resource environment settings:

Use the following script procedures to configure your URL provider settings:

Use the following script procedures to configure additional Java Enterprise Edition (JEE) resources:

Format for the scope argument

The scope format applies to the scripts in the script library that have the scope argument.

A cell is optional on node, server, and cluster scopes. A node is required on the server scope.

You can delimit the type by using a comma (,) or a colon (:). You can use lower case for the type (cell=, node=, server=, or cluster=.)

The examples in the following table are split on multiple lines for publishing purposes.

Table 334. Examples of the containment path, configuration ID, and type for a particular scope. The scope can be Cell, Node, Server, or Cluster.

Scope	Containment path	Configuration ID	Type
Cell	/Cell:*myCell*/	*myCell(cells/myCell|cell.xml#Cell_1)*	Cell=*myCell* or cell=*myCell*
Node	/Cell:myCell/Node:*myNode*/ or /Node:*myNode*/	*myNode(cells/myCell/nodes/myNode|node.xml#Node_1)*	`Cell=myCell, Node=myNode` or `Cell=myCell: Node=myNode` or `cell=myCell, node=myNode`
Server	`/Cell:myCell/Node: myNode/ Server:myServer/` or `/Node:myNode/Server: myServer/`	*myServer(cells/myCell/nodes/myNode/servers/myServer|server.xml#Server_1)*	`Cell=myCell, Node=myNode, Server=myServer` or `Node=myNode: Server=myServer` or `cell=myCell, Node=myNode, Server=myServer`
Cluster	`/Cell:myCell/ ServerCluster: myCluster/` or `/ServerCluster: myCluster/`	*myCluster(cells/myCell/clusters/myCluster|cluster.xml#ServerCluster_1)*	`Cell=myCell, Cluster=myCluster` or `Cell=myCell: Cluster=myCluster` or `cell=myCell, Cluster=myCluster`

createCompleteMailProvider

This script configures additional configuration attributes for your mail provider. A mail provider encapsulates a collection of protocol providers like SMTP, IMAP and POP3, while mail sessions authenticate users and

controls user access to messaging systems. Configure your own mail providers and sessions to customize how JavaMail is handled. The script returns the configuration ID of the created mail provider.

To run the script, specify the following arguments:

Table 335. createCompleteMailProvider arguments. Run the script to configure a mail provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mailProviderName	Specifies the mail provider that the application server uses for this mail session.
propName	Specifies the name of the custom property.
propValue	Specifies the value of the custom property.
protocolName	Specifies the name of the protocol provider. The application server contains protocol providers for SMTP, IMAP and POP3.
className	Specifies the implementation class name of the protocol provider.
mailSessionName	Specifies the administrative name of the JavaMail session object.
JNDIName	Specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts. This name provides the link between the platform binding information for resources defined in the client application deployment descriptor and the actual resources bound into JNDI by the platform.
mailStoreHost	Specifies the server that is accessed when receiving the mail. This setting, combined with the mail store user ID and password, represents a valid mail account. For example, if the mail account is `john_william@my.company.com`, then the mail store host is `my.company.com`.
mailStoreUser	Specifies the user ID for the given mail account. For example, if the mail account is `john_william@my.company.com` then the user is `john_william`.
mailStorePassword	Specifies the password for the given mail account . For example, if the mail account is `john_william@my.company.com` then enter the password for ID `john_william`.

Syntax

```
AdminResources.createCompleteMailProvider(nodeName,
 serverName, mailProviderName, propName, propValue,
 protocolName, className, mailSessionName, JNDIName,
 mailStoreHost, mailStoreUser, mailStorePassword)
```

Example usage

```
AdminResources.createCompleteMailProvider("myNode",
 "myServer", "myMailProvider", "myProp", "myPropValue", "myMailProtocol",
 "com.ibm.mail.myMailProtocol.myMailStore", "myMailSession", "myMailSession/jndi", "server1",
 "mailuser", "password")
```

createCompleteMailProviderAtScope

This script configures additional configuration attributes for your mail provider for the scope that you specify. A mail provider encapsulates a collection of protocol providers like SMTP, IMAP and POP3, while mail sessions authenticate users and controls user access to messaging systems. Configure your own mail providers and sessions to customize how JavaMail is handled. The script returns the configuration ID of the created mail provider for the specified scope.

To run the script, specify the scope, mail provider name, property name, property value, protocol name, class name, type, mail session name, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 336. createCompleteMailProviderAtScope arguments. Run the script to configure a mail provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
mailProviderName	Specifies the mail provider that the application server uses for this mail session.
propName	Specifies the name of the custom property.
propValue	Specifies the value of the custom property.

Table 336. createCompleteMailProviderAtScope arguments (continued). Run the script to configure a mail provider.

Argument	Description
protocolName	Specifies the name of the protocol provider. The application server contains protocol providers for SMTP, IMAP and POP3.
className	Specifies the implementation class name of the protocol provider.
type	Specifies the type of protocol provider. Valid options are `STORE` or `TRANSPORT`.
mailSessionName	Specifies the administrative name of the JavaMail session object.
jndiName	Specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts. This name provides the link between the platform binding information for resources defined in the client application deployment descriptor and the actual resources bound into JNDI by the platform.
otherAttributesList, *mailProviderAttributesList*, *mailSessionAttributesList*	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

The following table contains optional attributes for a mail provider:

Table 337. Optional attributes. Additional attributes available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.
description	Specifies a description of the mail provider.
isolatedClassLoader	If set to true, specifies that the mail provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.
providerType	Specifies the mail provider type that this mail provider uses.

The following table contains optional attributes for a mail session:

Table 338. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
debug	Specifies whether debug mode is used for this mail session. The default value is `off`.
description	Specifies a description of the mail provider.
mailFrom	Specifies the internet email address. If set to true, this mail provider is loaded in its own class loader. **CAUTION:** **A provider cannot be isolated when a native library path is specified.**
mailStoreHost	Specifies the server to connect to when receiving mail.
mailStorePort	Specifies the port to connect to when receiving mail.
mailStoreUser	Specifies the user of a mail account when an incoming mail server requires authentication.
mailStorePassword	Specifies the password of a mail account when an incoming mail server requires authentication.
mailStoreProtocol	Specifies the protocol to use when receiving mail.
mailTransportHost	Specifies the server to connect to when sending mail.
mailTransportPort	Specifies the port to connect to when sending mail.
mailTransportUser	Specifies the user of a mail account when an outgoing mail server requires authentication.
mailTransportPassword	Specifies the password of a mail account when an outgoing mail server requires authentication.
mailTransportProtocol	Specifies the protocol to use when sending mail.
strict	Specifies whether the recipient addresses is parsed in compliance with RFC 822. The default value is `true`.

Syntax

```
AdminResources.createCompleteMailProviderAtScope(scope,
  mailProviderName, propName, propValue,
 protocolName, className, type, mailSessionName, JNDIName, otherAttributesList,
mailProviderAttributesList, mailSessionAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createCompleteMailProviderAtScope("myScope",
 "myMailProvider", "myProp", "myPropValue", "myMailProtocol",
 "com.ibm.mail.myMailProtocol.myMailStore", "myMailSession", "myMailSession/jndi", "server1", "STORE")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createCompleteMailProviderAtScope("Node=AMYLIN4Node09, server=server1", "myMailProvider", "myProp",
"myPropValue", "myMailProtocol", "com.ibm.mail.myMailProtocol.myMailStore", "STORE", "myMailSession", "myMailSession/jndi",
"classpath=c:/temp, description='this is my mail', nativepath=c:/temp/nativepath, isolatedClassLoader=true",
"category=myCategory, debug=true, description='this is my mailsession', mailStoreUser=user1, mailStorePassword=password,
mailStoreHost=user1, mailStorePort=1000, mailTransportUser=user2, mailTransportPassword=password, mailTransportHost=test2,
mailTransportPort=1001, strict=true, mailFrom=test1@gmail.com")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createCompleteMailProviderAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myMailProvider", "myProp",
"myPropValue", "myMailProtocol", "com.ibm.mail.myMailProtocol.myMailStore", "STORE", "myMailSession", "myMailSession/jndi",
[['classpath', 'c:/temp'], ['description', 'this is my mail'], ['nativepath', 'c:/temp/nativepath'],
['isolatedClassLoader', 'true']]
[['category', 'myCategory'], ['debug', 'true'], ['description', 'this is my mailsession'], ['mailStoreUser', 'user1'],
['mailStorePassword', 'password'], ['mailStoreHost', 'user1'], ['mailStorePort', 1000], ['mailTransportUser', 'user2'],
['mailTransportPassword', 'password'], ['mailTransportHost', 'test2'], ['mailTransportPort', 1001], ['strict', 'true'],
['mailFrom', 'test1@gmail.com']])
```

createMailProvider

This script creates a mail provider in your environment. The application server includes a default mail provider called the built-in provider. If you use the default mail provider you only have to configure the mail session. To use the customized mail provider you must first create the mail provider and session. The script returns the configuration ID of the created mail provider.

To run the script, specify the node, server, and mail provider names as defined in the following table:

Table 339. createMailProvider arguments. Run the script to create a mail provider.

Argument	Description
nodeName	Specifies the name of the node on which to create the mail provider.
serverName	Specifies the name of the server for which to create the mail provider.
mailProviderName	Specifies the name to assign to the new mail provider.

Syntax

```
AdminResources.createMailProvider(nodeName, serverName,
 mailProviderName)
```

Example usage

```
AdminResources.createMailProvider("myNode", "myServer",
 "myMailProvider")
```

createMailProviderAtScope

This script creates a mail provider in your environment at the scope that you specify. The application server includes a default mail provider called the built-in provider. If you use the default mail provider you only have to configure the mail session. To use the customized mail provider you must first create the mail provider and session. The script returns the configuration ID of the created mail provider for the specified scope.

To run the script, specify the scope and mail provider name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 340. createMailProviderAtScope arguments. Run the script to create a mail provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
mailProviderName	Specifies the name to assign to the new mail provider.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 341. Optional attributes. Additional attributes available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.
description	Specifies a description of the mail provider.
isolatedClassLoader	If set to true, specifies that the mail provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.
providerType	Specifies the mail provider type that this mail provider uses.

Syntax

```
AdminResources.createMailProviderAtScope(scope, mailProviderName, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createMailProviderAtScope("myScope", "myMailProvider")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createMailProviderAtScope("Node=AMYLIN4Node09, server=server1", "myMailProvider", "classpath=c:/temp,
description='this is my mail', nativepath=c:/temp/nativepath, isolatedClassLoader=true")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createMailProviderAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myMailProvider",
[['classpath', 'c:/temp'],
['description', 'this ismy mail'], ['nativepath', 'c:/temp/nativepath'], ['isolatedClassLoader', 'true']])
```

createMailSession

This script creates a new mail session for your mail provider. Mail sessions are represented by the javax.mail.Session class. A mail session object authenticates users, and controls user access to messaging systems. The script returns the configuration ID of the created mail session.

To run the script, specify the node name, server name, mail provider name, mail session name, and Java Naming and Directory Interface (JNDI) name arguments, as defined in the following table:

Table 342. createMailSession arguments. Run the script to create a mail session.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mailProviderName	Specifies the mail provider that the application server uses for this mail session.
mailSessionName	Specifies the administrative name of the JavaMail session object.

Table 342. createMailSession arguments (continued). Run the script to create a mail session.

Argument	Description
JNDIName	Specifies the JNDI name for the resource, including any naming subcontexts. This name provides the link between the platform binding information for resources defined in the client application deployment descriptor and the actual resources bound into JNDI by the platform.

Syntax

```
AdminResources.createMailSession(nodeName, serverName,
 mailProviderName, mailSessionName, JNDIName)
```

Example usage

```
AdminResources.createMailSession("myNode", "myServer", "myMailProvider",
 "myMailSession", "myMailSession/jndi")
```

createMailSessionAtScope

This script creates a new mail session for your mail provider at the scope that you specify. Mail sessions are represented by the javax.mail.Session class. A mail session object authenticates users, and controls user access to messaging systems. The script returns the configuration ID of the created mail session for the specified scope.

To run the script, specify the scope, mail provider name, mail session name, and Java Naming and Directory Interface (JNDI) name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 343. createMailSessionAtScope arguments. Run the script to create a mail session.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
mailProviderName	Specifies the mail provider that the application server uses for this mail session.
mailSessionName	Specifies the administrative name of the JavaMail session object.
JNDIName	Specifies the JNDI name for the resource, including any naming subcontexts. This name provides the link between the platform binding information for resources defined in the client application deployment descriptor and the actual resources bound into JNDI by the platform.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 344. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
debug	Specifies whether debug mode is used for this mail session. The default value is `off`.
description	Specifies a description of the mail provider.
mailFrom	Specifies the internet email address. If set to true, this mail provider is loaded in its own class loader. **CAUTION:** **A provider cannot be isolated when a native library path is specified.**
mailStoreHost	Specifies the server to connect to when receiving mail.
mailStorePort	Specifies the port to connect to when receiving mail.
mailStoreUser	Specifies the user of a mail account when an incoming mail server requires authentication.
mailStorePassword	Specifies the password of a mail account when an incoming mail server requires authentication.
mailStoreProtocol	Specifies the protocol to use when receiving mail.
mailTransportHost	Specifies the server to connect to when sending mail.
mailTransportPort	Specifies the port to connect to when sending mail.

Table 344. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
mailTransportUser	Specifies the user of a mail account when an outgoing mail server requires authentication.
mailTransportPassword	Specifies the password of a mail account when an outgoing mail server requires authentication.
mailTransportProtocol	Specifies the protocol to use when sending mail.
strict	Specifies whether the recipient addresses is parsed in compliance with RFC 822. The default value is `true`.

Syntax

```
AdminResources.createMailSessionAtScope(scope,
 mailProviderName, mailSessionName, JNDIName, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createMailSessionAtScope("myScope", "myMailProvider",
 "myMailSession", "myMailSession/jndi", attributes")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createMailSessionAtScope("Node=AMYLIN4Node09, server=server1", "myMailProvider", "myMailSession",
"myMailSession/jndi", "category=myCategory, debug=true, description='this is my mailsession', mailStoreUser=user1,
mailStorePassword=password, mailStoreHost=user1, mailStorePort=1000, mailTransportUser=user2, mailTransportPassword=password,
mailTransportHost=test2, mailTransportPort=1001, strict=true, mailFrom=test1@gmail.com")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createMailSessionAtScope("/Cell:AMYLIN4Cell01/ServerClust
er:c1/", "myMailProvider", "myMailSession", "myMailSession/jndi",
[['category','myCategory'], ['description', 'this is mailsession'], ['debug', 'true'],
['mailTransportHost', 'test1'], ['mailTransportUser', 'user2'], ['mailTransportPassword', 'password'],
['mailStoreUser', 'user1'], ['mailStorePassword', 'password'], ['mailStoreHost', 'test2'], ['strict', 'true'],
['mailFrom', 'tester@mail.com']])
```

createProtocolProvider

This script creates a protocol provider in your configuration, which provides the implementation class for a specific protocol to support communication between your JavaMail application and mail servers. The application server contains protocol providers for SMTP, IMAP and POP3. If you require custom providers for different protocols, install them in your application serving environment before configuring the providers. See the JavaMail API design specification for guidelines. After configuring your protocol providers, return to the mail provider page to find the link for configuring mail sessions. The script returns the configuration ID of the created protocol provider.

To run the script, specify the following arguments:

Table 345. createProtocolProvider arguments. Run the script to create a protocol provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
mailProviderName	Specifies the name of the mail provider that the application server uses in association with the protocol provider.
protocolName	Specifies the name of the protocol provider. The application server contains protocol providers for SMTP, IMAP and POP3.
className	Specifies the implementation class name of the protocol provider.
type	Specifies the type of protocol provider. Valid options are `STORE` or `TRANSPORT`.

Syntax

```
AdminResources.createProtocolProvider(nodeName,
 serverName, mailProviderName, protocolName,
 className, type)
```

Example usage

```
AdminResources.createProtocolProvider("myNode", "myServer", "myMailProvider",
   "myMailProtocol", "com.ibm.mail.myMailProtocol.myMailStore",
   "STORE")
```

createProtocolProviderAtScope

This script creates a protocol provider in your configuration at the scope that you specify. The protocol provider provides the implementation class for a specific protocol to support communication between your JavaMail application and mail servers at the scope that you specify. The application server contains protocol providers for SMTP, IMAP and POP3. If you require custom providers for different protocols, install them in your application serving environment before configuring the providers. See the JavaMail API design specification for guidelines. After configuring your protocol providers, return to the mail provider page to find the link for configuring mail sessions. The script returns the configuration ID of the created protocol provider for the specified scope.

To run the script, specify the scope mail provider name, protocol name, class name, and type arguments. You can optionally specify the classpath attribute. The arguments and attributes are defined in the following tables:

Table 346. createProtocolProviderAtScope arguments. Run the script to create a protocol provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
mailProviderName	Specifies the name of the mail provider that the application server uses in association with the protocol provider.
protocolName	Specifies the name of the protocol provider. The application server contains protocol providers for SMTP, IMAP and POP3.
className	Specifies the implementation class name of the protocol provider.
type	Specifies the type of protocol provider. Valid options are `STORE` or `TRANSPORT`.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 347. Optional attributes. The classpath attribute is also available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.

Syntax

```
AdminResources.createProtocolProviderAtScope(nodeName,
 serverName, mailProviderName, protocolName,
 className, type, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createProtocolProviderAtScope("myScope", "myMailProvider",
   "myMailProtocol", "com.ibm.mail.myMailProtocol.myMailStore",
   "STORE")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createProtocolProviderAtScope("Node=AMYLIN4Node09, server=server1", "myMailProvider", "myMailProtocol",
"com.ibm.mail.myMailProtocol.myMailStore", "STORE", "classpath=c:/temp")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createProtocolProviderAtScope("/Cell:AMYLIN4Cell01/Server
Cluster:c1/", "myMailProvider", "myMailProtocol", "com.ibm.mail.myMailProtocol.m
yMailStore", "STORE", [['classpath', 'c:/temp']])
```

createCompleteResourceEnvProvider

This script configures a resource environment provider, which encapsulate the referenceables that convert resource environment entry data into resource objects in your configuration. The script returns the configuration ID of the created resource environment provider.

To run the script, specify the following arguments:

Table 348. createCompleteResourceEnvProvider arguments. Run the script to configure a resource environment provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the name to assign to the resource environment provider.
propName	Specifies the name of a custom property to set.
propValue	Specifies the value of the custom property.
factoryClass	Specifies the factory that converts resource environment entry data into a class instance for a physical resource.
className	Specifies the class name for the referenceable.
resourceEnvEntryName	Specifies the name of the resource environment entry.
JNDIName	Specifies the JNDI name for the resource environment entry, including any naming subcontexts. This name is used as the linkage between the platform binding information for resources defined by a module deployment descriptor and actual resources bound into JNDI by the platform.

Syntax

```
AdminResources.createCompleteResourceEnvProvider(nodeName,
 serverName, resourceEnvProviderName, propName,
 propValue, factoryClass, className,
 resourceEnvEntryName, JNDIName)
```

Example usage

```
AdminResources.createCompleteResourceEnvProvider("myNode", "myServer",
 "myResEnvProvider", "myProp", "myPropValue", "com.ibm.resource.res1", "java.lang.String",
 "myResEnvEntry", "res1/myResEnv")
```

createCompleteResourceEnvProviderAtScope

This script configures a resource environment provider at the scope that you specify. The resource environment provider encapsulates the referenceables that convert resource environment entry data into resource objects in your configuration. The script returns the configuration ID of the created resource environment provider for the specified scope.

To run the script, specify the scope, resource environment provider, custom property name, custom property value, factory class, class name, resoure environment entry, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 349. createCompleteResourceEnvProviderAtScope arguments. Run the script to configure a resource environment provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
resourceEnvProviderName	Specifies the name to assign to the resource environment provider.
propName	Specifies the name of a custom property to set.
propValue	Specifies the value of the custom property.

Table 349. createCompleteResourceEnvProviderAtScope arguments (continued). Run the script to configure a resource environment provider.

Argument	Description
factoryClass	Specifies the factory that converts resource environment entry data into a class instance for a physical resource.
className	Specifies the class name for the referenceable.
resourceEnvEntryName	Specifies the name of the resource environment entry.
jndiName	Specifies the JNDI name for the resource environment entry, including any naming subcontexts. This name is used as the linkage between the platform binding information for resources defined by a module deployment descriptor and actual resources bound into JNDI by the platform.
otherAttributesList, *resourceEnvProviderAttributesList*, *resourceEnvEntryAttributesList*	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

The following table contains optional attributes for the resource environment provider:

Table 350. Optional attributes. Additional attributes available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.
description	Specifies a description of the resource environment provider.
isolatedClassLoader	If set to true, specifies that this resource environment provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.
providerType	Specifies the resource provider type that this resource environment provider uses.

The following table contains optional attributes for the resource environment entry:

Table 351. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
description	Specifies a description of the resource provider.
providerType	Specifies the mail provider type that this resource environment provider uses.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.

Syntax

```
AdminResources.createCompleteResourceEnvProviderAtScope(scope,
  resourceEnvProviderName, propName,
 propValue, factoryClass, className,
 resourceEnvEntryName, JNDIName, otherAttributesList,
resourceEnvProviderAttributesList, resourceEnvProviderAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createCompleteResourceEnvProviderAtScope("myScope",
 "myResEnvProvider", "myProp", "myPropValue", "com.ibm.resource.res1", "java.lang.String",
 "myResEnvEntry", "res1/myResEnv")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createCompleteResourceEnvProviderAtScope("Node=AMYLIN4Node09, server=server1", "myResEnvProvider", "myProp",
"myPropValue", "com.ibm.resource.res1", "java.lang.String", "myResEnvEntry", "res1/myResEnv", "classpath=c:/temp,
description='this is my resource provider', nativepath=c:/temp/nativepath,
isolatedClassLoader=false", "category=myCategory, description='this is my resource entry',
referenceable=(cells/AMYLIN4Cell01/nodes/AMYLIN4Node09|resources.xml#Referenceable_1238331401156)")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createCompleteResourceEnvProviderAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myResEnvProvider",
"myProp", "myPropValue", "com.ibm.resource.res1", "java.lang.String", "myResEnvEntry", "res1/myResEnv", [['classpath', 'c:/temp'],
['description', 'this is my resource provider'], ['nativepath','c:/temp/nativepath'], ['isolatedClassLoader', 'false']]
[['category', 'myCategory'], ['description', 'this is my resource entry'],
['referenceable', '(cells/AMYLIN4Cell01/nodes/AMYLIN4Node09|resources.xml#Referenceable_1238331401156)']])
```

createResourceEnvEntries

This script creates a resource environment entry in your configuration. Within an application server name space, the data contained in a resource environment entry is converted into an object that represents a physical resource. This resource is frequently called an environment resource. The script returns the configuration ID of the created resource environment entry.

To run the script, specify the following arguments:

Table 352. createResourceEnvEntries arguments. Run the script to create a resource environment entry.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the resource environment provider for this entry. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.
resourceEnvEntry	Specifies a name for the resource environment entry to create.
JNDIName	Specifies the string to use to look up this environment resource using JNDI. This is the string to which you bind resource environment reference deployment descriptors.

Syntax

```
AdminResources.createResourceEnvEntries(nodeName,
 serverName, resourceEnvProviderName, referenceable,
 resourceEnvEntry, JNDIName)
```

Example usage

```
AdminResources.createResourceEnvEntries("myNode", "myServer",
 "myResEnvProvider", "com.ibm.resource.res1", "myResEnvEntry", "res1/myResEnv")
```

createResourceEnvEntriesAtScope

This script creates a resource environment entry in your configuration at the scope that you specify. Within an application server name space, the data contained in a resource environment entry is converted into an object that represents a physical resource. This resource is frequently called an environment resource. The script returns the configuration ID of the created resource environment entry for the specified scope.

To run the script, specify the scope the resource environment provider, resource environment entry, and JNDI name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 353. createResourceEnvEntriesAtScope arguments. Run the script to create a resource environment entry.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
resourceEnvProviderName	Specifies the resource environment provider for this entry. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.

Table 353. createResourceEnvEntriesAtScope arguments (continued). Run the script to create a resource environment entry.

Argument	Description
resourceEnvEntry	Specifies a name for the resource environment entry to create.
JNDIName	Specifies the string to use to look up this environment resource using JNDI. This is the string to which you bind resource environment reference deployment descriptors.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 354. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
description	Specifies a description of the resource provider.
providerType	Specifies the mail provider type that this resource environment provider uses.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.

Syntax

```
AdminResources.createResourceEnvEntriesAtScope(scope,
  resourceEnvProviderName,
 resourceEnvEntry, JNDIName, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createResourceEnvEntriesAtScope("myScope",
 "myResEnvProvider", "myResEnvEntry", "res1/myResEnv")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createResourceEnvEntriesAtScope("Node=AMYLIN4Node09, server=server1", "myResEnvProvider",
"myResEnvEntry", "res1/myResEnv", "category=myCategory, description='this is my resource entry',
referenceable=(cells/AMYLIN4Cell01/nodes/AMYLIN4Node09|resources.xml#Referenceable_1238331401156)")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createResourceEnvEntriesAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myResEnvProvider",
"myResEnvEntry", "res1/myResEnv", [['category', 'myCategory'], ['description', 'this is my resource entry'],
['referenceable', '(cells/AMYLIN4Cell01/nodes/AMYLIN4Node09|resources.xml#Referenceable_1238331401156)']])
```

createResourceEnvProvider

This script creates a resource environment provider in your configuration. The resource environment provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects. The script returns the configuration ID of the created Resource environment provider.

To run the script, specify the node name, server name, and resource environment provider name arguments, as defined in the following table:

Table 355. createResourceEnvProvider arguments. Run the script to create a resource environment provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the resource environment provider to create.

Syntax

```
AdminResources.createResourceEnvProvider(nodeName,
 serverName, resourceEnvProviderName)
```

Example usage

```
AdminResources.createResEnvProvider("myNode", "myServer",
 "myResEnvProvider")
```

createResourceEnvProviderAtScope

This script creates a resource environment provider in your configuration at the scope that you specify. The resource environment provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects. The script returns the configuration ID of the created Resource environment provider for the specified scope.

To run the script, specify the scope, and resource environment provider name arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 356. createResourceEnvProviderAtScope arguments. Run the script to create a resource environment provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
resourceEnvProviderName	Specifies the resource environment provider to create.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 357. Optional attributes. Additional attributes available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.
description	Specifies a description of the resource environment provider.
isolatedClassLoader	If set to true, specifies that this resource environment provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.
providerType	Specifies the resource provider type that this resource environment provider uses.

Syntax

```
AdminResources.createResourceEnvProviderAtScope(scope,
 resourceEnvProviderName, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createResEnvProviderAtScope("myScope",
 "myResEnvProvider")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createResourceEnvProviderAtScope("Node=AMYLIN4Node09, server=server1", "myResEnvProvider",
"classpath=c:/temp, description='this is my resource provider', nativepath=c:/temp/nativepath, isolatedClassLoader=false")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createResourceEnvProviderAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/",
"myResEnvProvider", [['classpath', 'c:/temp'],
['description','this is my resource provider'], ['nativepath', 'c:/temp/nativepath'], ['isolatedClassLoader', 'false']])
```

createResourceEnvProviderRef

This script creates a resource environment provider reference in your configuration. Resource environment references are different than resource references. Resource environment references allow your application client to use a logical name to look up a resource bound into the server JNDI namespace. A resource reference allows your application to use a logical name to look up a local JEE resource. The JEE specification does not specify a particular implementation of a resource. The script returns the configuration ID of the created resource environment provider reference ID.

To run the script, specify the following arguments:

Table 358. createResourceEnvProviderRef arguments. Run the script to create a resource environment provider reference.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
resourceEnvProviderName	Specifies the resource environment provider for this reference. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.
factoryClass	Specifies the class of the factory that converts resource environment entry data into a class instance for a physical resource.
className	Specifies the class name to associate with the referenceable.

Syntax

```
AdminResources.createResourceEnvProviderRef(nodeName,
 serverName, resourceEnvProviderName, factoryClass,
 className)
```

Example usage

```
AdminResources.createResourceEnvProviderRef("myNode", "myServer",
 "myResEnvProvider", "com.ibm.resource.res1", "java.lang.String")
```

createResourceEnvProviderRefAtScope

This script creates a resource environment provider reference in your configuration at the scope you specify. Resource environment references are different than resource references. Resource environment references allow your application client to use a logical name to look up a resource bound into the server JNDI namespace. A resource reference allows your application to use a logical name to look up a local JEE resource. The JEE specification does not specify a particular implementation of a resource. The script returns the configuration ID of the created resource environment provider reference ID for the specified scope.

To run the script, specify the following arguments:

Table 359. createResourceEnvProviderRefAtScope arguments. Run the script to create a resource environment provider reference.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
resourceEnvProviderName	Specifies the resource environment provider for this reference. The provider encapsulates the classes that, when implemented, convert resource environment entry data into resource objects.
factoryClass	Specifies the class of the factory that converts resource environment entry data into a class instance for a physical resource.
className	Specifies the class name to associate with the referenceable.

Syntax

```
AdminResources.createResourceEnvProviderRefAtScope(scope,
 resourceEnvProviderName, factoryClass,
 className, otherAttributesList)
```

Example usage

```
AdminResources.createResourceEnvProviderRefAtScope("/Node:AMYLIN4Node09/Server:server1/",
"myResEnvProvider", "com.ibm.resource.res1", "java.lang.String")
```

configURLProvider

This script configures a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs.

To run the script, specify the following arguments:

Table 360. configURLProvider arguments. Run the script to configure a URL provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.
propName	Specifies the name of the custom property to set for the URL provider.
propValue	Specifies the value of the custom property to set for the URL provider.
URLName	Specifies the name of a Uniform Resource Locator (URL) name that points to an Internet or intranet resource. For example: http://www.ibm.com.
JNDIName	Specifies the JNDI name. Do not assign duplicate JNDI names across different resource types, such as mail sessions versus URL configurations. Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.
URLSpec	Specifies the string from which to form a URL.

Syntax

```
AdminResources.configURLProvider(nodeName, serverName,
 URLProviderName, URLStreamHandlerClass, URLProtocol,
 propName, propValue, URLName, JNDIName,
 URLSpec)
```

Example usage

```
AdminResources.configURLProvider("myNode", "myServer", "myURLProvider",
 "com.ibm.resource.url1", "ftp", "myProp", "myPropValue", "myURL", "url1/myURL",
 "myURLSpec")
```

createCompleteURLProvider

This script creates a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs. The script returns the configuration ID of the created URL provider.

To run the script, specify the following arguments:

Table 361. createCompleteURLProvider arguments. Run the script to create a URL provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.

Table 361. createCompleteURLProvider arguments (continued). Run the script to create a URL provider.

Argument	Description
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.

Syntax

```
AdminResources.createCompleteURLProvider(nodeName,
 serverName, URLProviderName, URLStreamHandlerClass,
 URLProtocol)
```

Example usage

```
AdminResources.createCompleteURLProvider("myNode", "myServer",
 "myURLProvider", "com.ibm.resource.url1", "ftp")
```

createCompleteURLProviderAtScope

This script creates a URL provider at the scope specified. The URL provider supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs. The script returns the configuration ID of the created URL provider for the specified scope.

To run the script, specify the following arguments:

Table 362. createCompleteURLProviderAtScope arguments. Run the script to create a URL provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.
propName	Specifies the name of the custom property to set for the URL provider.
propValue	Specifies the value of the custom property to set for the URL provider.
URLName	Specifies the name of a Uniform Resource Locator (URL) name that points to an Internet or intranet resource. For example: http://www.ibm.com.
jndiName	Specifies the JNDI name. Do not assign duplicate JNDI names across different resource types, such as mail sessions versus URL configurations. Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.
URLSpec	Specifies the string from which to form a URL.
otherAttributesList, *urlProviderAttributesList*, *urlAttributesList*	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

The following table contains optional attributes for the URL provider:

Table 363. Optional attributes. Additional attributes available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.
description	Specifies a description of the resource environment provider.
isolatedClassLoader	If set to true, specifies that this URL provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.

Table 363. Optional attributes (continued). Additional attributes available for the script.

Attributes	Description
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.
providerType	Specifies the URL provider type that this resource environment provider uses.

The following table contains optional attributes for the URL:

Table 364. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
description	Specifies a description of the resource provider.
providerType	Specifies the mail provider type that this resource environment provider uses.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.

Syntax

```
AdminResources.createCompleteURLProviderAtScope(Scope,
  URLProviderName, URLStreamHandlerClass,
 URLProtocol, propName, propValue,
 URLName, JNDIName, URLSpec, otherAttributesList,
  urlProviderAttributesList,  urlAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createCompleteURLProviderAtScope("Scope",
 "myURLProvider", "com.ibm.resource.url1", "ftp", myProp", myPropValue",
 "myURL", "url1/myURL", "myURLSpec")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createCompleteURLProviderAtScope("Node=AMYLIN4Node09, server=server1", "myURLProvider",
"com.ibm.resource.url1",
"ftp","myProp", "myPropValue", "myURL", "url1/myURL", "myURLSpec", "classpath=c:/temp, description='this
 is my url provider',
nativepath=c:/temp/nativepath, isolatedClassLoader=true", "category=myCategory, description='this is my url'")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createCompleteURLProviderAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/",
"myURLProvider", "com.ibm.resource.url1",
"ftp","myProp", "myPropValue", "myURL", "url1/myURL", "myURLSpec", [['classpath', 'c:/temp'],
['description', 'this is my urlProvider'],
['nativepath', 'c:/temp/nativepath'], ['isolatedClassLoader', 'true']] [['category', 'myCategory'],
['description', 'this is my url']])
```

createURL

This script creates a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs. The script returns the configuration ID of the created URL.

To run the script, specify the following arguments:

Table 365. createURL arguments. Run the script to create a URL provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
URLProviderName	Specifies the name of the URL provider to assign the URL to.

Table 365. createURL arguments (continued). Run the script to create a URL provider.

Argument	Description
URLName	Specifies the name of the URL to create.
JNDIName	Specifies the JNDI name. Do not assign duplicate JNDI names across different resource types, such as mail sessions versus URL configurations. Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.
URLSpec	Specifies the string from which to form a URL.

Syntax

```
AdminResources.createURL(nodeName, serverName,
 URLProviderName, URLName, JNDIName,
 URLSpec)
```

Example usage

```
AdminResources.createURL("myNode", "myServer", "myURLProvider",
 "myURL", "url1/myURL", "myURLSpec")
```

createURLAtScope

This script creates a URL provider at the scope that you specify. The URL provider supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs. The script returns the configuration ID of the created URL for the specified scope.

To run the script, specify the scope, URL provider name, URL name, JNDI name, and URL specification arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 366. createURLAtScope arguments. Run the script to create a URL provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
URLProviderName	Specifies the name of the URL provider to assign the URL to.
URLName	Specifies the name of the URL to create.
JNDIName	Specifies the JNDI name. Do not assign duplicate JNDI names across different resource types, such as mail sessions versus URL configurations. Do not assign duplicate JNDI names for multiple resources of the same type in the same scope.
URLSpec	Specifies the string from which to form a URL.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 367. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
description	Specifies a description of the URL provider.
providerType	Specifies the mail provider type that this URL provider uses.

Syntax

```
AdminResources.createURLAtScope(scope,
 URLProviderName, URLName, JNDIName,
 URLSpec, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createURLAtScope("myScope", "myURLProvider",
 "myURL", "url1/myURL", "myURLSpec")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createURLAtScope("Node=AMYLIN4Node09, server=server1", "myURLProvider", "myURL", "url1/myURL",
"myURLSpec", "category=myCategory, description='this is my url'")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createURLAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/",
 "myURLProvider", "myURL", "url1/myURL", "myURLSpec", [['category', 'myCategory'], ['description', 'this is my url']])
```

createURLProvider

This script creates a URL provider, which supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs. The script returns the configuration ID of the created URL provider.

To run the script, specify the following arguments:

Table 368. createURLProvider arguments. Run the script to create a URL provider.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.

Syntax

```
AdminResources.createURLProvider(nodeName,
 serverName, URLProviderName, URLStreamHandlerClass,
 URLProtocol)
```

Example usage

```
AdminResources.createURLProvider("myNode", "myServer",
 "myURLProvider", "com.ibm.resource.url1", "ftp")
```

createURLProviderAtScope

This script creates a URL provider at the scope that you specify. The URL provider supplies the implementation classes that are necessary for the application server to access a URL through a specific protocol. The default URL provider provides connectivity through protocols that are supported by the IBM Developer Kit. These protocols include HTTP and File Transfer Protocol (FTP), which work for must URLs. The script returns the configuration ID of the created URL provider for the specified scope.

To run the script, specify the scope, URL provider name, URL stream handler class, and URL protocol arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 369. createURLProviderAtScope arguments. Run the script to create a URL provider.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
URLProviderName	Specifies the name of the URL provider to configure.
URLStreamHandlerClass	Specifies fully qualified name of a user-defined Java class that extends the java.net.URLStreamHandler class for a particular URL protocol, such as FTP.

Table 369. createURLProviderAtScope arguments (continued). Run the script to create a URL provider.

Argument	Description
URLProtocol	Specifies the protocol supported by this stream handler. For example, NNTP, SMTP, FTP.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 370. Optional attributes. Additional attributes available for the script.

Attributes	Description
classpath	Specifies a list of paths or Java archive (JAR) file names which together form the location for the resource provider classes. Use a semicolon (;) to separate class paths.
description	Specifies a description of the resource environment provider.
isolatedClassLoader	If set to true, specifies that this URL provider is loaded in its own class loader. **Attention:** A provider cannot be isolated when a native library path is specified.
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.
providerType	Specifies the URL provider type that this resource environment provider uses.

Syntax

```
AdminResources.createURLProviderAtScope(scope,
  URLProviderName, URLStreamHandlerClass,
 URLProtocol, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createURLProviderAtScope("myScope",
 "myURLProvider", "com.ibm.resource.url1", "ftp")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createURLProviderAtScope("Node=AMYLIN4Node09, server=server1", "myURLProvider", "com.ibm.resource.url1",
"ftp", "classpath=c:/temp, description='this is my url provider', nativepath=c:/temp/nativepath, isolatedClassLoader=true")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createURLProviderAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myURLProvider", "com.ibm.resource.url1",
"ftp", [['classpath', 'c:/temp'], ['description', 'this is my url provider'], ['nativepath', 'c:/temp/nativepath'],
['isolatedClassLoader', 'true']])
```

createJAASAuthenticationAlias

This script creates a Java Authentication and Authorization Service (JAAS) authentication alias. The alias identifies the authentication data entry. When configuring resource adapters or data sources, the administrator can specify which authentication data to choose using the corresponding alias. The script returns the configuration ID of the created JavaAuthentication and Authorization Service (JAAS) Authentication Alias.

To run the script, specify the following arguments:

Table 371. createJAASAuthenticationAlias arguments. Run the script to create a JAAS authentication alias.

Argument	Description
authAliasName	Specifies the name of the authentication alias to create.
authAliasID	A user identity of the intended security domain. For example, if a particular authentication data entry is used to open a new connection to DB2, this entry contains a DB2 user identity.
authAliasPW	Specifies the password of the user identity is encoded in the configuration repository.

Syntax

```
AdminResources.createJAASAuthenticationAlias(authAliasName,
 authAliasID, authAliasPW)
```

Example usage

```
AdminResources.createJAASAuthenticationAlias("myJAAS", "user01",
 "password")
```

createLibraryRef

This script creates a library reference, which defines how to use global libraries. The first step for making a library file available to multiple applications deployed on a server is to create a shared library for each library file that your applications need. When you create the shared libraries, set variables for the library files. The script returns the configuration ID of the created library reference.

To run the script, specify the following arguments:

Table 372. createLibraryRef arguments. Run the script to create a library reference.

Argument	Description
libraryRefName	Specifies the name of the library reference to create.
applicationName	Specifies the name of the application to associate with the library reference.

Syntax

```
AdminResources.createLibraryRef(libraryRefName,
 applicationName)
```

Example usage

```
AdminResources.createLibraryRef("myLibrary", "myApplication")
```

createSharedLibrary

This script creates a shared library in your configuration. The first step for making a library file available to multiple applications deployed on a server is to create a shared library for each library file that your applications need. When you create the shared libraries, set variables for the library files. The script returns the configuration ID of the created library.

To run the script, specify the following arguments:

Table 373. createSharedLibrary arguments. Run the script to create a shared library.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
sharedLibName	Specifies the name to assign to the shared library.
sharedLibClassPath	Specifies the file path where the product searches for classes and resources of the shared library. If a path in the list is a file, the product searches the contents of that `.jar` or compressed `.zip` file. If a path in the list is a directory, then the product searches the contents of `.jar` and `.zip` files in that directory. For performance reasons, the product searches the directory itself only if the directory contains subdirectories or files other than `.jar` or `.zip` files.

Syntax

```
AdminResources.createSharedLibrary(nodeName,
 serverName, sharedLibName, sharedLibClassPath)
```

Example usage

```
AdminResources.createSharedLibrary("myNode", "myServer", "myLibrary",
 "/myLibrary.zip")
```

createSharedLibraryAtScope

This script creates a shared library in your configuration at the scope that you specify. The first step for making a library file available to multiple applications deployed on a server is to create a shared library for each library file that your applications need. When you create the shared libraries, set variables for the library files. The script returns the configuration ID of the created library for the specified scope.

To run the script, specify the scope shared library name and shared library class path arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 374. createSharedLibraryAtScope arguments. Run the script to create a shared library.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the mail provider.
sharedLibName	Specifies the name to assign to the shared library.
sharedLibClassPath	Specifies the file path where the product searches for classes and resources of the shared library. If a path in the list is a file, the product searches the contents of that `.jar` or compressed `.zip` file. If a path in the list is a directory, then the product searches the contents of `.jar` and `.zip` files in that directory. For performance reasons, the product searches the directory itself only if the directory contains subdirectories or files other than `.jar` or `.zip` files.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 375. Optional attributes. Additional attributes available for the script.

Attributes	Description
description	Specifies a description of the shared library.
isolatedClassLoader	If set to true, specifies that the URL provider is loaded in its own class loader. The default value is `false`. **Attention:** A provider cannot be isolated when a native library path is specified.
nativepath	Specifies an optional path to any native libraries, such as *.dll and *.so. Use a semicolon (;) to separate native path entries.

Syntax

```
AdminResources.createSharedLibraryAtScope(scope,
 sharedLibName, sharedLibClassPath, otherAttributesList)
```

Example usage

The following example scripts contains required attributes only:

```
AdminResources.createSharedLibraryAtScope("myScope", "myLibrary",
 "/myLibrary.zip")
```

The following example script includes optional attributes in a string format:

```
 AdminResources.createSharedLibraryAtScope("Node=AMYLIN4Node09, server=server1", "myLibrary",
"c:/myLibrary.zip", "description='this is my library',
nativePath=/nativepath, isolatedClassLoader=true")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createSharedLibraryAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myLibrary",
"c:/myLibrary.zip", [['description', 'this is my library'],
['nativePath', '/nativepath'], ['isolatedClassLoader', 'true']])
```

createScheduler

This script creates a scheduler in your configuration. Schedulers are persistent and transactional timer services that can run business logic. Each scheduler runs tasks independently and has a programming

interface accessible from JEE applications using the Java Naming and Directory Interface (JNDI). You can also manage schedulers using a Java Management Extensions (JMX) MBean. See the scheduler documentation in the Information Center for details on how to configure and use schedulers. The script returns the configuration ID of the created scheduler.

To run the script, specify the following arguments:

Table 376. createScheduler arguments. Run the script to create a scheduler.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
schedulerName	Specifies the name by which this scheduler is known for administrative purposes.
JNDIName	Specifies the JNDI name that determines where this scheduler instance is bound in the name space. Clients can look this name up directly, although the use of resource references is recommended.
scheduleCategory	Specifies a string that can be used to classify or group this scheduler.
datasourceJNDI	Specifies the name of the data source where persistent tasks are stored. Any data source available in the name space can be used with a scheduler. Multiple schedulers can share a single data source while using different tables by specifying a table prefix.
tablePrefix	Specifies the string prefix to affix to the scheduler tables. Multiple independent schedulers can share the same database if each instance specifies a different prefix string.
pollInterval	Specifies the interval, in seconds, that a scheduler polls the database. The default value is appropriate for most applications. Each poll operation can be consuming. If the interval is extremely small and there are many scheduled tasks, polling can consume a large portion of system resources. The default value is `30`.
workMgmtJNDI	Specifies the JNDI name of the work manager, which is used to manage the number of tasks that can run concurrently with the scheduler. The work manager also can limit the amount of JEE context applied to the task.

Syntax

```
AdminResources.createScheduler(nodeName, serverName,
 schedulerName, JNDIName, scheduleCategory,
 datasourceJNDI, tablePrefix, pollInterval, workMgmtJNDI)
```

Example usage

```
AdminResources.createScheduler("myNode", "myServer", "myScheduler",
 "myScheduleJndi", "Default", "jdbc/MyDatasource", "sch1", "30",
 "myWorkManager")
```

createSchedulerAtScope

This script creates a scheduler in your configuration at the scope that you specify. Schedulers are persistent and transactional timer services that can run business logic. Each scheduler runs tasks independently and has a programming interface accessible from JEE applications using the Java Naming and Directory Interface (JNDI). You can also manage schedulers using a Java Management Extensions (JMX) MBean. See the scheduler documentation in the Information Center for details on how to configure and use schedulers. The script returns the configuration ID of the created scheduler for the specified scope.

To run the script, specify the scope, scheduler name, JNDI name, JNDI datasource, table prefix, poll interval, JNDI name of the work manager, and scheduler provider ID arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 377. createSchedulerAtScope arguments. Run the script to create a scheduler.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
schedulerName	Specifies the name by which this scheduler is known for administrative purposes.
JNDIName	Specifies the JNDI name that determines where this scheduler instance is bound in the name space. Clients can look this name up directly, although the use of resource references is recommended.

Table 377. createSchedulerAtScope arguments (continued). Run the script to create a scheduler.

Argument	Description
datasourceJNDI	Specifies the name of the data source where persistent tasks are stored. Any data source available in the name space can be used with a scheduler. Multiple schedulers can share a single data source while using different tables by specifying a table prefix.
tablePrefix	Specifies the string prefix to affix to the scheduler tables. Multiple independent schedulers can share the same database if each instance specifies a different prefix string.
pollInterval	Specifies the interval, in seconds, that a scheduler polls the database. The default value is appropriate for most applications. Each poll operation can be consuming. If the interval is extremely small and there are many scheduled tasks, polling can consume a large portion of system resources. The default value is `30`.
workMgmtJNDI	Specifies the JNDI name of the work manager, which is used to manage the number of tasks that can run concurrently with the scheduler. The work manager also can limit the amount of JEE context applied to the task.
schedulerProviderID	Specifies the provider ID of this scheduler.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 378. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
description	Specifies a description of the URL provider.
providerType	Specifies the resource provider type that this URL provider uses.
datasourceAlias	Specifies the alias for the user name and password that this URL provider uses to access the data source.
loginConfigName	Specifies the login configuration name.
securityRole	Specifies the security role.
useAdminRoles	Specifies that when this option and administrative security are both enabled, the user administration roles are enforced. For the user administrative roles to be enforced, you must use the scheduler JMX commands or APIs to create and modify tasks. If this option is not enabled, all the users can create and modify tasks. The default value is `false`.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.

Syntax

```
AdminResources.createSchedulerAtScope(scope,
 schedulerName, JNDIName,
 datasourceJNDI, tablePrefix, pollInterval, workMgmtJNDI,
 ScheduleProviderID, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createSchedulerAtScope("myNode",  "myScheduler",
 "myScheduleJndi", "Default", "jdbc/MyDatasource", "sch1", "30",
 "myWorkManager", "SchedulerProvider(cells/AMYLIN4Cell01/nodes/AMY
LIN4Node09/servers/server1|resources-pme.xml#SchedulerProvider_1)")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createSchedulerAtScope("Node=AMYLIN4Node09, server=server1", "myScheduler",
"myScheduleJndi", "jdbc/MyDatasource", "sch1", "30",
"myWorkManager",  "SchedulerProvider(cells/AMYLIN4Cell01/nodes/AMYLIN4Node09/servers/server1|
resources-pme.xml#SchedulerProvider_1)", "category=myCategory,
description='this is my scheduler', datasourceAlias=abc, useAdminRoles=false, loginConfigName=test,
securityRole=user1")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createSchedulerAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myScheduler",
"myScheduleJndi", "jdbc/MyDatasource", "sch1", "30",
"myWorkManager", "SchedulerProvider(cells/AMYLIN4Cell01/clusters/c1|resources-pme.xml#SchedulerProvider_1)",
[['category', 'myCategory'],
['description', 'this is my scheduler'], ['datasourceAlias', 'abc'], ['useAdminRoles', 'false'],
 ['loginConfigName', 'test'], ['securityRole', 'user1']])
```

createWorkManager

This script creates a work manager in your configuration. Work managers contain a pool of threads that are bound into Java Naming and Directory Interface. The script returns the configuration ID of the created work manager.

To run the script, specify the following arguments:

Table 379. createWorkManager arguments. Run the script to create a work manager.

Argument	Description
nodeName	Specifies the name of the node of interest.
serverName	Specifies the name of the server of interest.
workMgrName	Specifies the name by which this work manager is known for administrative purposes.
JNDIName	Specifies the Java Naming and Directory Interface (JNDI) name used to look up the work manager in the namespace.
workMgrCategory	Specifies a string that you can use to classify or group this work manager.
alarmThreads	Specifies the desired maximum number of threads used for alarms. The default value is `2`.
minThreads	Specifies the minimum number of threads available in this work manager.
maxThreads	Specifies the maximum number of threads available in this work manager.
threadPriority	Specifies the priority of the threads available in this work manager. Every thread has a priority. Threads with higher priority are run before threads with lower priority. For more information about how thread priorities are used, see the API documentation for the setPriority method of the java.lang.Thread class in the Java EE specification.
isGrowable	Specifies whether the number of threads in this work manager can be increased. Specify a value of `true` to indicate that the number of threads can increase.
serviceNames	Specifies a list of services to make available to this work manager.

Syntax

```
AdminResources.createWorkManager(nodeName, serverName,
 workMgrName, JNDIName, workMgrCategory,
 alarmThreads, minThreads, maxThreads, threadPriority,
 isGrowable, serviceNames)
```

Example usage

```
AdminResources.createWorkManager("myNode", "myServer", "myWorkManager",
 "Work Manager", "wm/myWorkManager", "Default", 5, 1, 10, 5, "true",
   "AppProfileService UserWorkArea com.ibm.ws.i18n security")
```

createWorkManagerAtScope

This script creates a work manager in your configuration at the scope that you specify. Work managers contain a pool of threads that are bound into Java Naming and Directory Interface. The script returns the configuration ID of the created work manager for the specified scope.

To run the script, specify the scope work manager name, JNDI name, alarm threads, minimum threads, maximum threads, thread priority, and work manager provider ID arguments. You can optionally specify attributes. The arguments and attributes are defined in the following tables:

Table 380. createWorkManageratScope arguments. Run the script to create a work manager.

Argument	Description
scope	Specifies a scope of cell, node, server, or cluster for the JMS provider.
workMgrName	Specifies the name by which this work manager is known for administrative purposes.
JNDIName	Specifies the Java Naming and Directory Interface (JNDI) name used to look up the work manager in the namespace.

Table 380. createWorkManageratScope arguments (continued). Run the script to create a work manager.

Argument	Description
alarmThreads	Specifies the desired maximum number of threads used for alarms. The default value is `2`.
minThreads	Specifies the minimum number of threads available in this work manager.
maxThreads	Specifies the maximum number of threads available in this work manager.
threadPriority	Specifies the priority of the threads available in this work manager. Every thread has a priority. Threads with higher priority are run before threads with lower priority. For more information about how thread priorities are used, see the API documentation for the setPriority method of the java.lang.Thread class in the Java EE specification.
workManagerProviderID	Specifies the configuration ID of the parent work manager provider. The work manager is created at this ID. You can create a work manager under the existing work manager provider or under a new work manager provider. The following wsadmin command is an example of how to retrieve the configuration ID of the existing work manager provider from the server scope: `workManagerProviderID = AdminConfig.getid ("/Cell:myCell/Node:myNode/Server:myServer /WorkManagerProvider:/")`
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Table 381. Optional attributes. Additional attributes available for the script.

Attributes	Description
category	Specifies the category that can be used to classify or group the resource.
description	Specifies a description of the URL provider.
providerType	Specifies the resource provider type that this URL provider uses.
daemonTranClass	Specifies the transaction class that is used for the workload classification of daemon work. The daemon work is for the workload manager service.
defTranClass	Specifies the transaction class name used to classify work run by this work manager instance when the z/OS workload manager service class information is not contained in the work context information.
isDistributable	Specifies whether this work manager is distributable. The default value is `false`.
isGrowable	Specifies whether the number of threads in this work manager can be increased. The default value is `true`.
serviceNames	Specifies a list of services to make available to this work manager.
workReqQFullAction	Specifies the action taken when the thread pool is exhausted, and the work request queue is full. The default value is `0`.
workReqQSize	Specifies the size of the work request queue. The default value is `0`.
workTimeout	Specifies the number of milliseconds to wait before attempting to release a unit of work. The default value is `0`.
referenceable	Specifies the referenceable, which encapsulates the class name of the factory that converts resource environment entry data into a class instance for a physical resource.
otherAttributesList	Optionally specifies additional attributes in a particular format: **List format** `[["attr1", "value1"], ["attr2", "value2"]]` **String format** "attr1=value1, attr2=value2"

Syntax

```
AdminResources.createWorkManagerAtScope(scope,
 workMgrName, JNDIName,
 alarmThreads, minThreads, maxThreads, threadPriority,
 workManagerProviderID, otherAttributesList)
```

Example usage

The following example script contains required attributes only:

```
AdminResources.createWorkManagerAtScope("scope", "myWorkManager",
 "Work Manager", "Default", 5, 1, 10,
   "WorkManagerProvider(cells/AMYLIN4Cell01/nodes/A
MYLIN4Node09/servers/server1|resources-pme.xml#WorkManagerProvider_1)")
```

The following example script includes optional attributes in a string format:

```
AdminResources.createWorkManagerAtScope("Node=AMYLIN4Node09, server=server1", "myWorkManager",
"wm/myWorkManager", "5", "1", "10", "5",
"WorkManagerProvider(cells/AMYLIN4Cell01/nodes/AMYLIN4Node09/servers/server1|resources-pme.xml#WorkManagerProvider_1)")
```

The following example script includes optional attributes in a list format:

```
AdminResources.createWorkManagerAtScope("/Cell:AMYLIN4Cell01/ServerCluster:c1/", "myWorkManager",
"wm/myWorkManager", "5", "1", "10", "5",
WorkManagerProvider(cells/AMYLIN4Cell01/clusters/c1|resources-pme.xml#WorkManagerProvider_1)",
[['category', 'myCategory'], ['description', 'this is my workmanager'],
['daemonTranClass', ''], ['defTranClass', ''], ['isDistributable', 'false'], ['isGrowable', 'true'],
['serviceNames', 'security'], ['workReqQFullAction', 0],
['workReqQSize', 0], ['workTimeout', 10]])
```

help

This script displays the script procedures that the AdminResources script library supports. To display detailed help for a specific script, specify the name of the script of interest, as defined in the following table:

Table 382. help arguments. Run the script to display help.

Argument	Description
script	Specifies the name of the script of interest.

Syntax

```
AdminResources.help(script)
```

Example usage

```
AdminResources.help("createWorkManager")
```

Displaying script library help information using the wsadmin scripting tool

The script library provides Jython script procedures to assist in automating your environment. The script library includes help commands to list each available script library, display information for specific script libraries, and to display information for specific script procedures.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

  # Use one of them as the first member of a cluster
  AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
      "myNode", "Server1")
  ```

```
# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/scriptLibraries directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/scriptLibraries/application/V70 subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

Use the AdminLibHelp script library to display general information about each script library, specific information about a specific script library, and information about specific scripts.

Procedure

- Display general script library information.

  Use the following command invocation to display general script library information with the wsadmin tool:

```
print AdminLibHelp()
```

- Display scripts in a specific script library.

  You can also use AdminLibHelp script to display each script within a specific script library. For example, the following command invocation displays each script in the AdminApplication script library:

```
print AdminLibHelp.help("AdminApplication")
```

- Display detailed script information.

  Use the help script with the script library of interest to display detailed descriptions, arguments, and usage information for a specific script. For example, the following command invocation displays detailed script information for the listApplications script in the AdminApplication script library:

```
print AdminApplication.help('listApplications')
```

Saving changes to the script library

The script library provides Jython script procedures to assist in automating your environment. You can save changes to the master configuration repository, disable the automatic saving of configuration changes, or discard configuration changes.

About this task

The scripting library provides a set of procedures to automate the most common application server administration functions. There are three ways to use the Jython script library.

- Run scripts from the Jython script library in interactive mode with the wsadmin tool. You can launch the wsadmin tool, and run individual scripts that are included in the script library using the following syntax:

  ```
  wsadmin>AdminServerManagement.createApplicationServer("myNode", "myServer", "default")
  ```

- Use a text editor to combine several scripts from the Jython script library, as the following sample displays:

  ```
  #
  # My Custom Jython Script - file.py
  #
  AdminServerManagement.createApplicationServer("myNode", "Server1", "default")
  ```

```
AdminServerManagement.createApplicationServer("myNode", "Server2", "default")

# Use one of them as the first member of a cluster
AdminClusterManagement.createClusterWithFirstMember("myCluster", "APPLICATION_SERVER",
    "myNode", "Server1")

# Add a second member to the cluster
AdminClusterManagement.createClusterMember("myCluster", "myNode", "Server3")

# Install an application
AdminApplication.installAppWithClusterOption("DefaultApplication",
    "..\installableApps\DefaultApplication.ear", "myCluster")

# Start all servers and applications on the node
AdminServerManagement.startAllServers("myNode")
```

Save the custom script and run it from the command line, as the following syntax demonstrates:

```
bin>wsadmin -language jython -f path/to/your/jython/file.py
```

- Use the Jython scripting library code as sample syntax to write custom scripts. Each script example in the script library demonstrates best practices for writing wsadmin scripts. The script library code is located in the *app_server_root*/`scriptLibraries` directory. Within this directory, the scripts are organized into subdirectories according to functionality, and further organized by version. For example, the *app_server_root*/`scriptLibraries/application/V70` subdirectory contains procedures that perform application management tasks that are applicable to Version 7.0 and later of the product.

The script library saves configuration changes to the master configuration repository by default when each script procedure completes. You can disable or enable the automatic saving of configuration changes with the AdminUtilities.configureAutoSave() script procedure. You can alternatively save changes to the configuration before leaving the wsadmin process by using the AdminConfig.sav() command. You can discard configuration changes with the AdminConfig.reset() command.

CAUTION:
If you disable the autosave procedure, call script procedures, then enable the autosave procedure, automatic saving of script procedures in the same wsadmin process does not occur until you call another script procedure.

Procedure

- Disable the automatic saving of configuration changes.

  Use the following script procedure:

```
AdminUtilities.configureAutoSave("false")
```

- Save changes to the master configuration repository.

  You can save changes to the configuration using the save command or enable automatic saving of configuration changes.

  – Save changes to the configuration.

    Use the save command before leaving the wsadmin process.

```
AdminConfig.save()
```

  – Enable the automatic saving of configuration changes.

    Use the following script procedure to save subsequent changes that script procedures make to the configuration:

```
AdminUtilities.configureAutoSave("true")
```

- Discard the configuration changes.

  Use the reset command to discard the changes:

```
AdminConfig.reset()
```

Results

Depending on the steps that you completed, you have disabled or enabled the automatic saving of configuration changes, completed the saving of configuration changes using the save command, or discarded the configuration changes.

What to do next

Continue administering your environment using the script library.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
The default Application Client for IBM WebSphere Application Server profile root is the `/QIBM/UserData/WebSphere/AppClient/V8/client/profiles/`*`profile_name`* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppServer/V8/Base` directory.

java_home

Table 383. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit`
64–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit`

plugins_profile_root
The default Web Server Plug-ins profile root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/`*`profile_name`* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the `/QIBM/ProdData/WebSphere/Plugins/V8/webserver` directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver` directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to `.exe`, `.dll`, `.so` objects) for the installed product. The product library name is `QWAS8`*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is `QWAS8A`. The *app_server_root*`/properties/product.properties` file contains the value for the product library of the installation, `was.install.library`, and is located under the *app_server_root* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/`*profile_name* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS8`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base` directory.

The `profiles` and `profileRegistry` subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Chapter 12. Administering applications using wsadmin scripting

You can use administrative scripts and the wsadmin tool to install, uninstall, and manage applications.

About this task

There are two methods you can use to install, uninstall, and manage applications. You can use the commands for the AdminApp and AdminControl objects to invoke operations on your application configuration.

Alternatively, you can use the AdminApplication and BLAManagement Jython script libraries to perform specific operations to configure your enterprise and business-level applications.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

You might need to complete one or more of the following topics to administer your application configurations with the wsadmin tool.

Procedure

- Install enterprise applications. Use the AdminApp object or the AdminApplication script library to install an application to the application server runtime. You can install an enterprise archive file (EAR), web application archive (WAR) file, servlet archive (SAR), or Java archive (JAR) file.
- Install business-level applications. Use the BLAManagement command group for the AdminTask object or the AdminBLA script library to install business-level applications.
- Manage enterprise applications using pattern matching. Use the AdminApp object or the AdminApplication script library to implement pattern matching when installing, updating, or editing an application. Pattern matching simplifies the task of supplying required values for certain complex options by allowing you to pass in asterisk (*) to all of the required values that cannot be edited.
- Manage Integrated Solutions Console applications. Use the AdminApp object to deploy or remove portlet-based Integration Solutions Console applications.
- Uninstall enterprise applications. Use the AdminApp object or the AdminApplication script library to uninstall applications.
- Uninstall business-level applications. Use the BLAManagement command group for the AdminTask object or the AdminBLA script library to uninstall business-level applications.
- Switch JavaServer Faces implementations. Use the modifyJSFImplementation command to set the Sun Reference Implementation or the Apache MyFaces project as the JSF implementation for web applications.

Installing enterprise applications using wsadmin scripting

Use the AdminApp object or the AdminApplication script library to install an application to the application server run time. You can install an enterprise archive file (EAR), web application archive (WAR) file, servlet archive (SAR), or Java archive (JAR) file.

Before you begin

On a single server installation, verify that the server is running before you install an application. Use the startServer command utility to start the server.

There are two ways to complete this task. Complete the steps in this topic to use the AdminApp object to install enterprise applications. Alternatively, you can use the scripts in the AdminApplication script library to install, uninstall, and administer your application configurations.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

About this task

Use this topic to install an application from an enterprise archive file (EAR), a web application archive (WAR) file, a servlet archive (SAR), or a Java archive (JAR) file. The archive file must end in `.ear`, `.jar`, `.sar` or `.war` for the wsadmin tool to complete the installation. The wsadmin tool uses these extensions to determine the archive type. The wsadmin tool automatically wraps WAR and JAR files as an EAR file.

Note: Use the most recent product version of the wsadmin tool when installing applications to mixed-version environments to ensure that the most recent wsadmin options and commands are available.

Procedure

1. Start the wsadmin scripting tool.
2. Determine which options to use to install the application in your configuration.

   For example, if your configuration consists of a node, a cell, and a server, you can specify that information when you enter the **install** command. Review the list of valid options for the **install** and **installinteractive** commands in the “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands using wsadmin scripting” on page 898 topic to locate the correct syntax for the **-node, -cell,** and **-server** options. For this configuration, use the following command examples:

   Using Jython:

```
AdminApp.install('location_of_ear.ear','[-node nodeName -cell cellName -server serverName]')
```

   Using Jacl:

```
$AdminApp install "location_of_ear.ear" {-node nodeName -cell cellName -server serverName}
```

   You can also obtain a list of supported options for an enterprise archive (EAR) file using the **options** command, for example:

   Using Jython:

```
print AdminApp.options()
```

   Using Jacl:

```
$AdminApp options
```

   You can set or update a configuration value using options in batch mode. To identify which configuration object is to be set or updated, the values of read only fields are used to find the corresponding configuration object. All the values of read only fields have to match with an existing configuration object, otherwise the command fails.

   You can use pattern matching to simplify the task of supplying required values for certain complex options. Pattern matching only applies to fields that are required or read only.
3. Choose to use the **install** or **installInteractive** command to install the application.

   You can install the application in batch mode, using the **install** command, or you can install the application in interactive mode using the **installinteractive** command. Interactive mode prompts you through a series of tasks to provide information. Both the **install** command and the **installinteractive** command support the set of options you chose to use for your installation in the previous step.
4. Install the application. For this example, only the **server** option is used with the **install** command, where the value of the **server** option is `serv2`. Customize your **install** or **installInteractive** command with on the options you chose based on your configuration.
   - Using the **install** command to install the application in batch mode:

– For a single server installation only, the following example uses the EAR file and the command option information to install the application:

- Using Jython string:

```
AdminApp.install('home/myProfile/MyStuff/application1.ear', '[-server serv2]')
```

- Using Jython list:

```
AdminApp.install('/home/myProfile/MyStuff/application1.ear', ['-server', 'serv2'])
```

- Using Jacl:

```
$AdminApp install "/home/myProfile/MyStuff/application1.ear" {-server serv2}
```

Table 384. install server command elements. Run the install command with the -server option.

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object supporting application object management
install	is an AdminApp command
MyStuff/application1.ear	is the name of the application to install
server	is an installation option
serv2	is the value of the server option

- Use the **installInteractive** command to install the application using interactive mode. The following command changes the application information by prompting you through a series of installation tasks:

  – Using Jython:

```
AdminApp.installInteractive('/home/myProfile/MyStuff/application1.ear')
```

  – Using Jacl:

```
$AdminApp installInteractive "/home/myProfile/MyStuff/application1.ear"
```

Table 385. installInteractive command elements. Run the installInteractive command with the name of the application to install.

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object allowing application objects to be managed
installInteractive	is an AdminApp command
MyStuff/application1.ear	is the name of the application to install

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

What to do next

The steps in this task return a success message if the system successfully installs the application. When installing large applications, the command might return a success message before the system extracts each binary file. You cannot start the application until the system extracts all binary files. If you installed a large application, use the **isAppReady** and **getDeployStatus** commands for the AdminApp object to verify that the system extracted the binary files before starting the application.

The **isAppReady** command returns a value of `true` if the system is ready to start the application, or a value of `false` if the system is not ready to start the application. For example, using Jython:

```
print AdminApp.isAppReady('application1')
```

Using Jacl:

```
$AdminApp isAppReady application1
```

If the system is not ready to start the application, the system might be expanding application binaries. Use the **getDeployStatus** command to display additional information about the binary file expansion status, as the following examples display:

Using Jython:

```
print AdminApp.getDeployStatus('application1')
```

Using Jacl:

```
$AdminApp getDeployStatus application1
```

Running the **getDeployStatus** command where *application1* is `DefaultApplication` results in status information about DefaultApplication resembling the following:

```
ADMA5071I: Distribution status check started for application DefaultApplication.
WebSphere:cell=myCell01,node=myNode01,distribution=unknown,expansion=unknown
ADMA5011I: The cleanup of the temp directory for application DefaultApplication is complete.
ADMA5072I: Distribution status check completed for application DefaultApplication.
WebSphere:cell=myCell01,node=myNode01,distribution=unknown,expansion=unknown
```

Setting up business-level applications using wsadmin scripting

You can create an empty business-level application, and then add assets, shared libraries, or business-level applications as composition units to the empty business-level application.

Before you begin

Before you can create a business-level application, determine the assets or other files to add to your application.

Also, verify that the target application server is configured. As part of configuring the server, determine whether your application files can run on your deployment targets.

About this task

You can use the wsadmin tool to create business-level applications in your environment. This topic demonstrates how to use the AdminTask object to import and register assets, create empty business-level applications, and add assets to the business-level application as composition units. Alternatively, you can use the scripts in the AdminBLA script library to set up and administer business-level applications.

Procedure

1. Start the wsadmin scripting tool.
2. Import assets to your configuration.

   Assets represent application binaries that contain business logic that runs on the target runtime environment and serves client requests. An asset can contain an archive of files such as a compressed (zip) or Java archive (JAR) file, or an archive of archive files such as a Java Platform, Enterprise Edition (Java EE) enterprise archive (EAR) file. Examples of assets include EAR files, shared library JAR files, and custom advisors for proxy servers.

   Use the importAsset command to import assets to the application server configuration repository. See the documentation for the BLAManagement command group for the AdminTask object for additional parameter and step options.

   For this example, the commands add three assets to the asset repository. Two of the assets are non-Java EE assets and one is an enterprise asset. The following command imports the `asset1.zip` asset to the asset repository and sets the returned configuration ID to the `asset1` variable:

```
asset1 = AdminTask.importAsset('-source \ears\asset1.zip')
```

   The following command imports the `asset2.zip` asset metadata only, sets the asset name as `testAsset.zip`, sets the deployment directory, specifies that the asset is used for testing, and sets the returned configuration ID to the `testasset` variable:

```
testasset = AdminTask.importAsset('-source \ears\asset2.zip -storageType
 METADATA –AssetOptions [[.* testAsset.zip .* "asset for testing"
 c:/installedAssets/testAsset.zip/BASE/testAsset.zip "" "" "" false]]')
```

The following command imports the `defaultapp.ear` asset, storing all application binaries, and sets the returned configuration ID to the `J2EEAsset` variable:

```
J2EEAsset = AdminTask.importAsset('-source \ears\defaultapplication.ear
 –storageType FULL –AssetOptions [[.* defaultapp.ear .* "desc" "" "" "" "" false]]')
```

The assets of interest are registered as named configuration artifacts in the application server configuration repository, which is referred to as the asset registry. Use the listAssets command to display a list of registered assets and verify that the settings are correct, as the following example demonstrates:

```
AdminTask.listAssets('-includeDescription true -includeDeplUnit
 true')
```

3. Create an empty business-level application.

   Use the createEmptyBLA command to create a new business-level application and set the returned configuration ID to the `myBLA` variable, as the following example demonstrates:

```
myBLA = AdminTask.createEmptyBLA('-name myBLA -description "BLA that contains
 asset1, asset2, and J2EEAsset"')
```

   The system creates the business-level application. Use the listBLAs command to display a list of each business-level application in the cell, as the following example demonstrates:

```
AdminTask.listBLAs()
```

4. Add the assets, as composition units, to the business-level application.

   Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-Application Server run times without backing assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

   The following command adds the `asset1.zip` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` server:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset1 -CUOptions [[.* .*
 compositionUnit1 "composition unit that is backed by asset1" 0]] -MapTargets [[.* server1]]
 –ActivationPlanOptions [[.* specname=actplan0+specname=actplan1]]')
```

   The following command adds the `testAsset.zip` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset2 -CUOptions [[.* .*
 compositionUnit2 "composition unit that is backed by asset2" 0]] -MapTargets [[.*
 server1+testServer]] –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

   The following command adds the `defaultapp.ear` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('[-blaID bla1 -cuSourceID ' + J2EEAsset + '
 -defaultBindingOptions
 defaultbinding.ejbjndi.prefix=ejb#defaultbinding.virtual.host=default_host#defaultbinding.force=yes
-AppDeploymentOptions [-appname defaultapp] -MapModulesToServers [["Default Web Application" .*
 WebSphere:cell=cellName,node=nodeName,server=server1] ["Increment EJB module" .*
 Websphere:cell=cellName,node=nodeName,server=testServer]] -CtxRootForWebMod [["Default Web Application" .*
 myctx/]]]')
```

5. Save your configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

6. Start the business-level application.

   Use the startBLA command to start each composition unit of the business-level application on the deployment targets for which the composition units are configured, as the following example demonstrates:

```
AdminTask.startBLA('-blaID myBLA')
```

Results

The system adds three composition units backed by assets to a new business-level application. Each of the three assets are deployed and started on the `server1` server. The `testAsset.zip` and `defaultapp.ear` assets are also deployed and started on the `testServer` server.

Uninstalling enterprise applications using the wsadmin scripting tool

You can use the AdminApp object or the AdminApplication script library to uninstall applications.

Before you begin

There are two ways to complete this task. This topic uses the AdminApp object to uninstall enterprise applications. Alternatively, you can use the scripts in the AdminApplication script library to install, uninstall, and administer your application configurations.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

Procedure

1. Start the wsadmin scripting tool.
2. Uninstall the application:

   Specify the name of the application you want to uninstall, not the name of the Enterprise Archive (EAR) file.

   - Using Jacl:

```
$AdminApp uninstall application1
```

   - Using Jython:

```
AdminApp.uninstall('application1')
```

Table 386. uninstall command elements. Run the uninstall command to remove an application from a server.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object supporting application objects management
`uninstall`	is an AdminApp command
`application1`	is the name of the application to uninstall

3. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Results

Uninstalling an application removes it from the application server configuration and from each server that the application was installed on. The system deletes the application binaries (EAR file contents) from the installation directory. This occurs when the configuration is saved for single server product versions or when the configuration changes are synchronized from the deployment manager to the individual nodes for network deployment configurations.

Deleting business-level applications using wsadmin scripting

You can use the wsadmin tool to remove business-level applications from your environment. Deleting a business-level application removes the application from the product configuration repository and it deletes the application binaries from the file system of all nodes where the application files are installed.

Before you begin

There are two ways to complete this task. This topic uses the commands in the BLAManagement command group for the AdminTask object to remove business-level applications from your configuration. Alternatively, you can use the scripts in the AdminBLA script library to configure, administer, and remove business-level applications

About this task

Procedure

1. Start the wsadmin scripting tool.
2. Verify that the business-level application is ready to be deleted.

   Before deleting a business-level application, use the deleteCompUnit command to remove each configuration unit that is associated with the business-level application. Also, verify that no other business-level applications reference the business-level application to delete.

   Use the following example to delete the composition units for the business-level application of interest:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID compositionUnit1')
```

   Repeat this step for each composition unit that is associated with the business-level application of interest.
3. Delete the business-level application.

   Use the deleteBLA command to remove a business-level application from your configuration, as the following example demonstrates:

```
AdminTask.deleteBLA('-blaID myBLA')
```

   If the system successfully deletes the business-level application, the command returns the configuration ID of the deleted business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

4. Save your configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Pattern matching using the wsadmin scripting tool

Use the Jython or Jacl scripting language to implement pattern matching when installing, updating, or editing an application. Pattern matching simplifies the task of supplying required values for certain complex options by allowing you to pass in asterisk (*) to all of the required values that cannot be edited.

Before you begin

There are two ways to complete this task. This topic uses the AdminApp object to install enterprise applications. Alternatively, you can use the scripts in the AdminApplication script library to install, uninstall, and administer your application configurations with many options, including pattern matching.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

Procedure

- Install each web application archive (WAR) and Java archive file to the application server.
  1. Start the wsadmin scripting tool.
  2. Install each web application archive (WAR) and Java archive file to the application server, as the following examples demonstrate:
     – Using Jython:

```
AdminApp.install('DefaultApplication.ear', ['-appname', 'TEST',  '-MapModulesToServers', [['.*',
 '.*', 'WebSphere:cell=myCell,node=myNode,server=myServer']]])
```

– Using Jacl:

```
$AdminApp install DefaultApplication.ear {-appname TEST  -MapModulesToServers
  {{.* .* WebSphere:cell=myCell,node=myNode,server=myServer}}}
```

3. Save your configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

- Install each WAR file to the `myServer` server on the `myNode`node and each JAR file to the `yourServer` server on the `yourNode` node.
  1. Start the wsadmin scripting tool.
  2. Install the WAR and JAR files to different application server management scopes, as the following examples demonstrate:

     – Using Jython:

```
AdminApp.install('DefaultApplication.ear', ['-appname', 'TEST', '-MapModulesToServers', [['.*',
 '.*.war,.*', 'WebSphere:cell=myCell,node=myNode,server=myServer'], ['.*', '.*.jar,.*',
 'WebSphere:cell=myCell,node=yourNode,server=yourServer']]])
```

     – Using Jacl:

```
$AdminApp install DefaultApplication.ear {-appname TEST -MapModulesToServers
  {{.* .*.war,.* WebSphere:cell=myCell,node=myNode,server=myServer}
  {.* .*.jar,.* WebSphere:cell=myCell,node=yourNode,server=yourServer}}}
```

  3. Save your configuration changes.

     Use the following command example to save your configuration changes:

     ```
     AdminConfig.save()
     ```

Managing administrative console applications using wsadmin scripting

Use the Jython or Jacl scripting languages to deploy or remove portlet-based administrative console applications.

Before you begin

Verify that the administrative console Enterprise Archive (EAR) file is not archived before installation.

Procedure

- Deploy a portlet-based console application into the EAR file.
  1. Start the wsadmin scripting tool.
  2. Deploy a portlet-based console application into the EAR file.

     To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the `wsadmin.properties` file in the properties directory of the profile of interest.

     – Using Jython:

```
AdminApp.update('isclite', 'modulefile', '[-operation add -contents
/WebSphere/AppServer/systemApps/isclite.ear/upzippedWarName
-contenturi upzippedWARName -usedefaultbindings -contextroot contextroot]')
```

     – Using Jacl:

```
$AdminApp update isclite modulefile {-operation add -contents
/WebSphere/AppServer/systemApps/isclite.ear/upzippedWarName
-contenturi upzippedWARName -usedefaultbindings -contextroot contextroot}
```

  3. Save your configuration changes.

Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

- Remove a portlet-based web application archive (WAR) file.
  1. Start the wsadmin scripting tool.
  2. Remove the portlet-based WAR file, as the following examples demonstrate:
     – Using Jython:

```
AdminApp.update('isclite', 'modulefile', '[-operation delete -contenturi WarName]')
```

     – Using Jacl:

```
$AdminApp update isclite modulefile {-operation delete -contenturi WarName}
```

  3. Save your configuration changes.

     Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Managing JavaServer Faces implementations using wsadmin scripting

JavaServer Faces (JSF) is a user interface framework or application programming interface (API) that eases the development of Java based web applications. The product supports JSF at a runtime level, which reduces the size of Web applications since runtime binaries no longer need to be included in your web application. Use the wsadmin tool to set the JSF implementation as the Sun Reference 1.2 implementation or the Apache MyFaces 2.0 project.

About this task

The JSF runtime:

- Makes it easy to construct a user interface from a set of reusable user interface components.
- Simplifies migration of application data to and from the user interface.
- Helps manage user interface state across server requests.
- Provides a simple model for wiring client-generated events to server-side application code.
- Supports custom user interface components to be easily build and reused.

Procedure

1. Start the wsadmin scripting tool.
2. Determine whether to use JSF with your applications.

   Review specification documentation for JSF 2.0 to determine whether to use JSF with your applications. Then, determine which implementation to use. You can use the Sun Reference Implementation or the open source Apache MyFaces project. MyFaces is the default implementation.
3. Set the JSF implementation.

   Use the modifyJSFImplementation command for the AdminTask object to set the JSF implementation.
   - The following example sets the Sun Reference Implementation for JSF:

```
AdminTask.modifyJSFImplementation('myApplication', '[-implName "SunRI1.2"]')
```

   - The following example sets the MyFaces implementation for JSF:

```
AdminTask.modifyJSFImplementation('myApplication', '[-implName "MyFaces"]')
```

4. Recompile the JavaServer Pages (JSP) if you switched implementations and use precompiled JavaServer Pages (JSP) that contain JSF.

BLAManagement command group for the AdminTask object using wsadmin scripting

You can use the Jython scripting language to configure and administer business-level applications with the wsadmin tool. Use the commands and parameters in the BLAManagement group to create, edit, export, delete, and query business-level applications in your configuration.

In order to configure and administer business-level applications you must use the Configurator administrative role.

An asset represents one or more application binary files that are stored in an asset repository. Typical assets include application business logic such as enterprise archives, library files, and other resource files. Use the following commands to manage your asset configurations:

- deleteAsset
- editAsset
- exportAsset
- importAsset
- listAssets
- updateAsset
- viewAsset

A business-level application is a configuration artifact that consists of zero or more composition units or other business-level applications. Business-level applications are administrative models that define an application, and can contain enterprise archive (EAR) files, shared libraries, PHP applications, and more. Use the following commands to configure and administer business-level applications:

- createEmptyBLA
- deleteBLA
- editBLA
- getBLAStatus
- listBLAs
- listControlOps
- startBLA
- stopBLA
- viewBLA

A composition unit represents an asset in a business-level application. A configuration unit enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents. Use the following commands to manage your composition unit configurations:

- addCompUnit
- deleteCompUnit
- editCompUnit
- listCompUnits
- setCompUnitTargetAutoStart
- viewCompUnit

deleteAsset

The deleteAsset command removes an asset from your business-level application configuration. Before using this command, verify that no composition units are associated with the asset of interest. The command fails if the asset is associated with configuration units.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to delete. The command accepts incomplete IDs for the assetID parameter, as long as the system can match the string to a unique asset. (String, required)

Optional parameters

-force
Specifies whether to force the system to delete the asset, even if other assets depend on this asset. (Boolean, optional)

Return value

The command returns the configuration ID of the deleted asset, as the following example displays:

```
WebSphere:assetname=asset2.zip
```

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteAsset('-assetID asset2.zip -force true')
```

- Using Jython list:

```
AdminTask.deleteAsset(['-assetID', 'asset2.zip', '-force', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteAsset('-interactive')
```

editAsset

The editAsset command modifies additional asset configuration options. You can use this command to modify the description, destination URL, asset relationships, file permissions, and validation settings.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to edit. This parameter accepts an incomplete configuration ID, as long as the system can match the string to a unique asset ID. (String, required)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-AssetOptions
Use the AssetOptions step and the following arguments to set additional properties for the asset.

inputAsset (read-only)
Specifies the source package of the asset.

name (read-only)
Specifies the name of the asset. The default value for this argument is the file name of the source package.

defaultBindingProps (read-only)
Specifies the default binding properties for the asset. This argument only applies to enterprise assets. For assets which are not enterprise assets, specify the asterisk character (*) for pattern matching. For enterprise assets, specify the `.*` value to set the argument as a non-empty value.

description
Specifies a description for the asset.

destinationUrl
Specifies the URL of the asset binaries to deploy.

typeAspect
Specifies the asset type aspect.

relationship
Specifies the asset relationship. Use the plus sign character (+) to add additional assets to the existing relationship. Use the number sign character (#) to delete an existing asset from the relationship. To replace the existing relationships, specify the same syntax as in the importAsset command. If the asset specified in the relationship does not exist for add or update, the command returns an exception.

filePermission
Specifies the file permission configuration.

validate
Specifies whether to validate the asset. The default value is `false`. The product does not save the value specified for validate. Thus, if you select to validate the asset (`true`) now and later edit the asset, when you edit the asset you must enable this setting again for the product to validate the updated files.

-UpdateAppContentVersions
For an EBA asset, use this step and the following arguments to select bundle versions for the asset.

trns: In the WebSphere Application Server Version 7 Feature Pack for OSGi Applications and Java Persistence API 2.0, bundle changes to the asset are applied by restarting the business-level application. In Version 8, these changes are applied by updating the composition unit. The new approach in Version 8 means that many bundle changes can be applied in place, without restarting the running business-level application. To enable this new approach, the **UpdateAppContentVersionsStep** parameter has been replaced with the **UpdateAppContentVersions** parameter, and instead of restarting the business-level application you run the editCompUnit command with the **CompUnitStatusStep** parameter.

bundle_name
Specifies the name of the bundle.

current_version
Specifies either a bundle version number, for example `1.0.0`, or `NOT_DEPLOYED` for shared bundles (that is, *use bundles*) that are declared in the application manifest but not deployed by the runtime environment. This argument describes the current configuration of the bundle, and is not used to change the configuration.

update_preference
Specifies the new bundle version preference. This is either a bundle version number, for example `1.0.0`, or `NOT_DEPLOYED` for shared bundles, or `NO_PREF` if you want the system to choose a bundle version for you. If you do not want to update the version for a given bundle, set this attribute to the same value that you are using for the *current_version* attribute.

Include an entry (that is, the *bundle_name current_version* and *update_preference*) for each bundle that is listed in the application manifest between the application content header and the use bundle header. Include every bundle, whether or not you are updating the bundle version.

Specify the syntax as follows:

```
AdminTask.editAsset('[
  -assetID asset_name
  -UpdateAppContentVersions [
    [bundle_1_name current_version update_preference]
    [bundle_2_name current_version update_preference]
    [bundle_3_name current_version update_preference]
    [bundle_4_name current_version update_preference]
    [bundle_5_name current_version update_preference]
  ]]')
```

Return value

The command returns the configuration ID of the asset of interest.

Batch mode example usage

Use the following examples to edit a non-enterprise asset:

- Using Jython string:

```
AdminTask.editAsset('-assetID asset3.zip –AssetOptions [[.* asset3.zip * "asset for testing"
   c:/installedAssets/asset3.zip/BASE/asset3.zip "" assetname=a.jar "" false]]')
```

- Using Jython list:

```
AdminTask.editAsset(['-assetID', 'asset3.zip', '–AssetOptions', '[[.* asset3.zip * "asset for testing"
   c:/installedAssets/asset3.zip/BASE/asset3.zip "" assetname=a.jar "" false]]'])
```

Use the following examples to edit an enterprise asset:

- Using Jython string:

```
AdminTask.editAsset('-assetID defaultapp.ear –AssetOptions
  [[.* defaultapp.ear .* "asset for testing" "" "" "" "" false]]')
```

- Using Jython list:

```
AdminTask.editAsset(['-assetID', 'defaultapp.ear', '–AssetOptions',
   '[[.* defaultapp.ear .* "asset for testing" "" "" "" "" false]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editAsset('-interactive')
```

exportAsset

The exportAsset command exports an asset configuration to a file.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to export. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, required)

-filename
Specifies the file name to which the system exports the asset configuration. (DownloadFile, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.exportAsset('-assetID asset2.zip –filename c:/temp/a2.zip')
```

- Using Jython list:

```
AdminTask.exportAsset(['-assetID', 'asset2.zip', '–filename', 'c:/temp/a2.zip'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.exportAsset('-interactive')
```

importAsset

The importAsset command imports an asset configuration to the asset repository. After importing assets, you can add the assets to business-level applications as composition units.

Target object

None

Required parameters

-source
Specifies the name of the source file to import. (UploadFile, required)

Optional parameters

-storageType
Specifies the way the system saves the asset in the asset repository. The default asset repository stores full binaries, metadata of binaries, or no binaries. Specify `FULL` to store full binaries. Specify `METADATA` to store the metadata portion of the binaries. Specify `NONE` to store no binaries in the asset repository. The default value is `FULL`. (String, optional)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-AssetOptions
Use the AssetOptions step and the following arguments to set additional properties for the asset.

inputAsset (read-only)
Specifies the source package of the asset.

name Specifies the name of the asset. The extension file name of the asset must match the extension file name of the source package. The default value for this argument is the file name of the source package.

defaultBindingProps (read-only)
Specifies the default binding properties for the asset. This argument only applies to enterprise assets. For assets which are not enterprise assets, specify the asterisk character (*) for pattern matching. For enterprise assets, specify the `.*` value to set the argument as a non-empty value.

description
Specifies a description for the asset.

destinationUrl
Specifies the URL of the asset binaries to deploy.

typeAspect
Specifies the asset type aspect. Specify the typeAspect option in object name format, as the following syntax demonstrates: `spec=xxx`

relationship
Specifies the asset relationship. Use the plus sign character (+) to specify multiple asset relationships. The command returns an exception if you specify assets in the relationship that do not exist.

filePermission
Specifies the file permission configuration.

validate
Specifies whether to validate the asset. The default value is `false`. The product does not save the value specified for validate. Thus, if you select to validate the asset (`true`) now and later edit the asset, when you edit the asset you must enable this setting again for the product to validate the updated files.

Return value

The command returns the configuration ID of the asset that the system creates, as the following example displays:

```
WebSphere:assetname=asset2.zip
```

Batch mode example usage

Use the following examples to import a non-enterprise asset:

- Using Jython string:

```
AdminTask.importAsset('-source c:\ears\asset1.zip -storageType NONE')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\asset1.zip', '-storageType', 'NONE'])
```

Use the following examples to import a non-enterprise asset, set `asset2.zip` as the asset name, save the metadata binaries in the asset repository, and set the destination directory of the binaries to deploy:

- Using Jython string:

```
AdminTask.importAsset('-source c:\ears\asset1.zip -storageType METADATA –AssetOptions
  [[.* asset2.zip .* "asset for testing" c:/installedAssets/asset2.zip/BASE/asset2.zip "" "" "" "" false]]')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\asset1.zip', '-storageType', 'METADATA', '–AssetOptions',
   '[[.* asset2.zip .* "asset for testing" c:/installedAssets/asset2.zip/BASE/asset2.zip "" "" "" "" false]]')
```

Use the following examples to import a non-enterprise asset, and specifies asset relationships with the `a.jar` and `b.jar` assets:

- Using Jython string:

```
AdminTask.importAsset('[-source c:\ears\asset3.zip -storageType FULL –AssetOptions
   [[.* asset3.zip .* "asset for testing" "" spec=zip assetname=a.jar+assetname=b.jar "" false]]]')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\asset3.zip', '-storageType', 'FULL', '–AssetOptions',
  '[[.* asset3.zip .* "asset for testing" "" spec=zip assetname=a.jar+assetname=b.jar "" false]]'])
```

Use the following examples to import an enterprise asset:

- Using Jython string:

```
AdminTask.importAsset('-source c:\ears\defaultapplication.ear –storageType FULL –AssetOptions
   [[.* defaultapp.ear .* "desc" "" "" "" false]]')
```

- Using Jython list:

```
AdminTask.importAsset(['-source', 'c:\ears\defaultapplication.ear', '–storageType',
  'FULL', '–AssetOptions', '[[.* defaultapp.ear .* "desc" "" "" "" false]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importAsset('-interactive')
```

listAssets

The listAssets command displays the configuration ID of each asset within the cell.

Target object

None

Optional parameters

-assetID
Specifies the configuration ID of the asset of interest. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, optional)

-includeDescription
Specifies whether to include the a description of each asset that the command returns. Specify `true` to display the asset descriptions. (String, optional)

-includeDeplUnit
Specifies whether to display the deployable units for each asset that the command returns. Specify `true` to display the deployable units. (String, optional)

Return value

The command returns a list of configuration IDs for the assets of interest. Depending on the parameter values specified, the command might display the description and deployable composition units for each asset, as the following example displays:

```
WebSphere:assetname=asset1.zip
"asset for testing"

WebSphere:assetname=asset2.zip
"second asset for testing"
a.jar

WebSphere:aasetname=asset3.zip
"third asset for testing"
a1.jar+a2.jar

WebSphere:assetname=a.jar0
"a.jar for sharedlib"

WebSphere:assetname=b.jar
"b.jar for sharedlib"

WebSphere:assetname=defaultapp.ear
"default app"
```

Batch mode example usage

Use the following examples to list each asset in the cell:

- Using Jython:

```
AdminTask.listAssets()
```

Use the following examples to list each asset in the cell:

- Using Jython string:

```
AdminTask.listAssets('-assetID asset1.zip')
```

- Using Jython list:

```
AdminTask.listAssets(['-assetID asset1.zip'])
```

Use the following examples to list each asset, asset description, and deployable composition units in the cell:

- Using Jython string:

```
AdminTask.listAssets('-includeDescription true –includeDeplUnit true')
```

- Using Jython list:

```
AdminTask.listAssets(['-includeDescription', 'true', '–includeDeplUnit', 'true')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listAssets('-interactive')
```

updateAsset

The updateAsset command modifies one or more files or module files of an asset. The command updates the asset binary file, but does not update the composition units that the system deploys with the asset as a backing object.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset to update. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, required)

-operation
Specifies the operation to invoke on the asset of interest. (String, required)

The following table displays each operation that you can invoke on an asset:

Table 387. updateAsset supported operations. Specify one of the operations.

Operation	Description
`replace`	The `replace` operation replaces the contents of the asset of interest.
`merge`	The `merge` operation updates multiple files for the asset, but does not update all files.
`add`	The `add` operation adds a new file or module file.
`addupdate`	The `addupdate` operation adds or updates one file or module file. If the file does not exist, the system adds the contents. If the file exists, the system updates the file.
`update`	The `update` operation updates one file or module file.
`delete`	The `delete` operation deletes a file or module file.

-contents
Specifies the file that contains the content to add or update. This parameter is not required for the `delete` operation. (UploadFile, optional)

Optional parameters

-contenturi
Specifies the Uniform Resource Identifier (URI) of the file to add, update, or remove from the asset. This parameter is not required for the `merge` or `replace` operations. (String, optional)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the "" characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-AssetOptions

Use the AssetOptions step and the following arguments to set additional properties for the asset.

name (read-only)

Specifies the name of the asset. The default value for this argument is the file name of the source package.

defaultBindingProps (read-only)

Specifies the default binding properties for the asset. This argument only applies to enterprise assets. For assets which are not enterprise assets, specify the asterisk character (*) for pattern matching. For enterprise assets, specify the `.*` value to set the argument as a non-empty value.

description

Specifies a description for the asset.

destinationUrl

Specifies the URL of the asset binaries to deploy.

typeAspect

Specifies the asset type aspect.

relationship

Specifies the asset relationship. Use the plus sign character (+) to add additional assets to the existing relationship. Use the number sign character (#) to delete an existing asset from the relationship. To replace the existing relationships, specify the same syntax as in the importAsset command. If the asset specified in the relationship does not exist for add or update, the command returns an exception.

filePermission

Specifies the file permission configuration.

validate

Specifies whether to validate the asset. The default value is `false`. The product does not save the value specified for validate. Thus, if you select to validate the asset (`true`) now and later edit the asset, when you edit the asset you must enable this setting again for the product to validate the updated files.

updateAssociatedCUs

Specifies whether to update the composition units that are associated with an enterprise (Java EE) asset. This argument applies to enterprise assets only. The default value is `none`. Specify `all` to update all of the composition units that are associated with the enterprise asset.

For the `replace` operation, specify values for the AssetOptions name, defaultBindingProps, description, destinationUrl, typeAspect, relationship, filePermission, validate, and updateAssociatedCUs arguments. For operations other than `replace`, specify values for the AssetOptions name and updateAssociatedCUs arguments.

Return value

The command returns the configuration ID of the asset of interest.

Batch mode example usage

The following example replaces the contents of a non-enterprise asset:

- Using Jython string:

```
AdminTask.updateAsset('-assetID asset1.zip -operation replace -contents c:/temp/a.zip')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'asset1.zip', '-operation', 'replace', '-contents', 'c:/temp/a.zip'])
```

The following example partially updates the files of a non-enterprise asset:

- Using Jython string:

```
AdminTask.updateAsset('-assetID asset1.zip –operation merge –contents c:/temp/p.zip')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'asset1.zip', '–operation', 'merge', '–contents', 'c:/temp/p.zip'])
```

The following example updates an enterprise asset with an Enterprise JavaBeans (EJB) module file:

- Using Jython string:

```
AdminTask.updateAsset('-assetID defaultapp.ear –operation add –contents
  c:/temp/filename.jar –contenturi filename.jar')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'defaultapp.ear', '–operation', 'add', '–contents',
  'c:/temp/filename.jar', '–contenturi', 'filename.jar'])
```

The following example replaces an enterprise asset and its associated composition units using a replace operation:

- Using Jython string:

```
AdminTask.updateAsset('-assetID defaultapp.ear –operation replace –contents
  c:/temp/newapp.ear –AssetOptions [[defaultapp.ear .* newdesc "" "" "" "" false all]]')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'defaultapp.ear', '–operation', 'replace', '–contents',
  'c:/temp/newapp.ear', '–AssetOptions [[defaultapp.ear .* newdesc "" "" "" "" false all]]'])
```

The following example updates an enterprise asset and its associated composition units using a merge operation:

- Using Jython string:

```
AdminTask.updateAsset('-assetID defaultapp.ear –operation merge –contents
  c:/temp/newapp.ear –AssetOptions [[defaultapp.ear all]]')
```

- Using Jython list:

```
AdminTask.updateAsset(['-assetID', 'defaultapp.ear', '–operation', 'merge', '–contents',
  'c:/temp/newapp.ear', '–AssetOptions [[defaultapp.ear all]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.updateAsset('-interactive')
```

viewAsset

The viewAsset command displays additional asset configuration options and configured values.

Target object

None

Required parameters

-assetID
Specifies the configuration ID of the asset of interest. This parameter accepts an incomplete configuration ID as long as the ID matches a unique asset. (String, required)

Optional parameters

None

Return value

The command returns configuration data for the asset of interest, as the following example displays:

```
Specify Asset options (AssetOptions)

Specify options for Asset.

*Asset Name (name): [defaultapp.ear]
Default Binding Properties (defaultBindingProps):
 [defaultbinding.ejbjndi.prefix#defaultbinding.datasource.jndi#defaultbinding.datasource.username
#defaultbinding.datasource.password#defaultbinding.cf.jndi
#defaultbinding.cf.resauth#defaultbinding.virtual.host#defaultbinding.force]
Asset Description (description): []
Asset Binaries Destination Url (destination): [${USER_INSTALL_ROOT}/installedAssets/defaultapp.ear/BASE/defaultapp.ear]
Asset Type Aspects (typeAspect): [WebSphere:spec=j2ee_ear]
Asset Relationships (relationship): []File Permission (filePermission): [.*\\.dll=755#.*\\.so=755#.*\\.a=755#.*\\.sl=755]
Validate asset (validate): [false]
```

Batch mode example usage

- Using Jython string:

```
AdminTask.viewAsset('-assetID asset3.zip')
```

- Using Jython list:

```
AdminTask.viewAsset(['-assetID', 'asset3.zip'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.viewAsset('-interactive')
```

addCompUnit

The addCompUnit command adds a composition unit to a specific business-level application. A composition unit represents an asset in a business-level application, and enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

-cuSourceID
Specifies the source configuration ID for the composition unit to add. You can specify an asset ID or a business-level application ID. (String, required)

Optional parameters

-deplUnits
Specifies the deployable units to deploy for the asset. You can specify a subset of deployable units, all deployable units, or use the default as a shared library. If you do not specify this parameter, the system deploys each deployable unit. (String, optional)

For Java EE assets, the system ignores this -deplUnits parameter and, regardless of the value specified, can add Java EE assets as part of this command.

-cuConfigStrategyFile
Specifies the fully qualified file path for custom default binding properties. This parameter only applies to enterprise assets. (String, optional)

-defaultBindingOptions

Specifies optional Java Naming and Directory Interface (JNDI) binding properties for an enterprise asset. The binding properties available depend upon the type of enterprise asset. Use the format `property=value` to specify a default binding property. To specify more than one property, separate each `property=value` statement by the delimiter `#`.

You can specify binding properties now, when creating the asset, or later, when adding the asset as a composition unit to a business-level application. If you specify binding properties later, when adding the asset to a business-level application, then you can use a strategy file to specify the binding properties. (String, optional)

Use the following options with the defaultBindingOptions parameter:

Table 388. addCompUnit -defaultBindingOptions supported binding properties. Specify a binding property that is supported for the asset type.

enterprise asset type	Supported binding properties
Enterprise bean (EJB)	`defaultbinding.ejbjndi.prefix` `defaultbinding.force`
Data source	`defaultbinding.datasource.jndi` `defaultbinding.datasource.username` `defaultbinding.datasource.password` `defaultbinding.force`
Connection factory	`defaultbinding.cf.jndi` `defaultbinding.cf.resauth` `defaultbinding.force`
Virtual host	`defaultbinding.virtual.host` `defaultbinding.force`

Optional steps

You can also specify values for optional steps to set additional properties for the new composition unit. These steps do not apply to enterprise assets. For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-CUOptions

Specifies additional properties for the composition unit. Specify the following options with the CUOptions step:

parentBLA (read-only)
Specifies the parent business-level application for the new composition unit.

backingID (read-only)
Specifies the composition unit source ID.

name Specifies the name of the composition unit.

description
Specifies a description of the composition unit.

startingWeight
Specifies the starting weight of the composition unit. Supported values are from 1 to 2147483647, the maximum Integer value.

startedOnDistributed
Specifies whether to start the composition unit after distributing changes to the target nodes. The default value is `false`.

restartBehaviorOnUpdate
Specifies the nodes to restart after editing the composition unit. Specify `ALL` to restart each target node. Specify `DEFAULT` to restart the nodes controlled by the sync plug-ins. Specify `NONE` to prevent the system from restarting nodes.

For example, specify the syntax of this step as `-CUOptions [[.* .* cu4 "cu4 desc" 0 false DEFAULT]]`

-MapTargets
Specifies additional properties for the composition unit target mapping. Specify the following options with the MapTargets step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI).

For an enterprise bundle archive (EBA) asset, this URI is `ebaDeploymentUnit`.

server Specifies the target or targets to deploy the composition units. The default value is the `server1` server. Use the plus sign character (+) to specify multiple targets. Use the plus sign character (+) as a prefix to add an additional target. Specify the complete object name format for each server that is not aWebSphere Application Server server.

For example, specify the syntax of this step as `-MapTargets [[a1.jar cluster1+cluster2] [a2.jar +server2]]`

-ActivationPlanOptions
Specifies additional properties for the composition unit activation plan. Specify the following options with the ActivationPlanOptions step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI).

activationPlan
Specifies a list of runtime components as the activation plan. Specify each activation plan in the format `specName=xxx,specVersion=yyy`, where `specName` represents the name of the specification and is required. Use the plus sign character (+) to specify multiple activation plans.

For example, specify the syntax of this step as `-ActivationPlanOptions [[a1.jar specname=actplan0+specname=actplan1] [a2.jar specname=actplan1+specname=actplan2]]`

For an EBA asset, use the following default values: `-ActivationPlanOptions [[default ""]]`

-CreateAuxCUOptions
Specifies additional properties for an auxiliary composition unit. Use this step if the composition unit source is an asset that corresponds to an asset that does not have a matching composition unit in the business-level application. Specify the following options with the CreateAuxCUOptions step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI).

inputAsset (read-only)
Specifies composition unit source ID.

cuID Specifies the composition unit ID that the system creates for the asset. If you do not want to create a new composition unit, do not specify this argument.

matchTarget
Specifies whether to match the targets of the dependency auxiliary composition unit with the targets of the new composition unit. The default value is `true`.

The product does not save the value specified for matchTarget. Thus if you select to not match the target (`false`) now and later edit the composition unit, when you edit the composition unit you must disable this setting again for the product to not match the targets.

For example, specify the syntax of this step as `–CreateAuxCUOptions [[a1.jar a.jar auxCU true] [a2.jar a.jar defaultCU false]]`

-RelationshipOptions

Specifies additional properties for relationships between assets, composition units, and business-level applications. Use this step if the source ID of the composition unit is an asset that has a matching composition unit in the business-level application. Specify the following options with the RelationshipOptions step:

deplUnit (read-only)

Specifies the deployable unit Uniform Resource Identifier (URI).

relationship

Defines the composition unit relationships. Specify the composition unit object name in the format: `cuName=xxx`. Use the plus sign character (+) to specify multiple composition unit object names in the relationship. If the composition unit specified in the relationship does not exist under the same business-level application, the system returns an error.

matchTarget

Specifies whether to match the targets of the composition unit relationship with the targets of the new composition unit. The default value is `true`.

The product does not save the value specified for matchTarget. Thus if you select to not match the target (`false`) now and later edit the composition unit, when you edit the composition unit you must disable this setting again for the product to not match the targets.

For example, specify the syntax of this step as `–RelationshipOptions [[a1.jar a.jar auxCU true] [a2.jar a.jar defaultCU false]]`

-ContextRootStep

For an EBA asset, context roots determine where the web pages of a particular web application bundle (WAB) are found at run time.

The context root that you specify here is combined with the defined server mapping to compose the full URL that you enter to access the pages of the WAB. For example, if the application server default host is www.example.com:8080 and the context root of the WAB is /sample, the web pages are available at www.example.com:8080/sample.

You should list the context roots for all the WAB modules that are contained in the OSGi application.

Specify the syntax of this step as follows:

```
-ContextRootStep [
  [bundle_symbolic_name_1 bundle_version_1 context_root_1]
  [bundle_symbolic_name_2 bundle_version_2 context_root_2]]
```

For example, for an EBA file containing two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `/hello/web`, and `com.ibm.ws.eba.helloWorldService.withContextRoot` at version 0.9.0, which is to be mapped to `/hello/service`), this aspect of the command is as follows:

```
-ContextRootStep [
  [com.ibm.ws.eba.helloWorldService.web 1.0.0 "/hello/web"]
  [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "/hello/service"]]
```

-VirtualHostMappingStep

For an EBA asset, you use a virtual host to associate a unique port with a module or application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host alias is used in the URL that is used to access artifacts such as servlets and JavaServer Page (JSP) files in a web module. For example, the alias `myhost:8080` is the *host_name:port_number* portion of the URL `http://myhost:8080/sample`.

Each WAB that is contained in a deployed asset must be mapped to a virtual host. WABs can be installed on the same virtual host, or dispersed among several virtual hosts.

If you specify an existing virtual host in the `ibm-web-bnd.xml` or `.xmi` file for a given WAB, the specified virtual host is set by default. Otherwise, the default virtual host setting is `default_host`, which provides several port numbers through its aliases:

80 An internal, insecure port used when no port number is specified
9080 An internal port
9443 An external, secure port

Unless you want to isolate your WAB from other WABs or resources on the same node (physical machine), `default_host` is a suitable virtual host. In addition to `default_host`, WebSphere Application Server provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the WAB relates to system administration.

Specify the syntax of this step as follows:

```
-VirtualHostMappingStep [
    [bundle_symbolic_name_1 bundle_version_1
    web_module_name_1 virtual_host_1]
    [bundle_symbolic_name_2 bundle_version_2
     web_module_name_2 virtual_host_2]]
```

For example, for an EBA file containing two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `default_host`, and `com.ibm.ws.eba.helloWorldService.withContextRoot` at version 0.9.0, which is to be mapped to `test_host`), this aspect of the command is as follows:

```
-VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "HelloWorld second service" test_host]]
```

-MapRolesToUsersStep

For an EBA asset, use this step to map security roles to users or groups.

Specify the syntax of this step as follows:

```
-MapRolesToUsersStep [
    [role_name everyone?
    all_authenticated_in_realm?
    usernames groups]]
```

Key:

- *role_name* is a role name defined in the application.
- *everyone?* is set to Yes or No, to specify whether or not everyone is in the role.
- *all_authenticated_in_realm?* is set to Yes or No, to specify whether or not all authenticated users can access the application realm.
- *usernames* is a list of WebSphere Application Server user names, separated by the "|" character.
- *groups* is a list of WebSphere Application Server groups, separated by the "|" character.

Note: For *usernames*, and *groups*, the empty string "" means "use the default or existing value". The default value is usually that no users or groups are bound to the role. However, when an application contains an `ibm-application-bnd.xmi` file, the default value for *usernames* is obtained from this file. If you are deploying an application that contains an `ibm-application-bnd.xmi` file, and you want to remove the bound users, specify just the "|" character (which is the separator for multiple user names). This explicitly specifies "no users", and therefore guarantees that no users are bound to the role.

For example:

```
-MapRolesToUsersStep [
    [ROLE1 No Yes "" ""]
    [ROLE2 No No WABTestUser1 ""]
    [ROLE3 No No "" WABTestGroup1]
    [ROLE4 Yes No "" ""]]
```

-BlueprintResourceRefBindingStep

For an EBA asset, Blueprint components can access WebSphere Application Server resource references. Each reference is declared in a Blueprint XML file, and can be secured using a Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) authentication alias. Each bundle in an OSGi application can contain any number of resource reference declarations in its various Blueprint XML files.

When you secure resource references, those resource references can be bound only to JCA authentication aliases that exist on every server or cluster that the OSGi application is deployed to. An OSGi application can be deployed to multiple servers and clusters that are in the same security domain. Therefore, each JCA authentication alias must exist in either the security domain of the target servers and clusters, or the global security domain. For more information, see Adding an EBA asset to a composition unit using the addCompUnit command.

Specify the syntax of this step as follows:

```
-BlueprintResourceRefBindingStep [
    [
    bundle_symbolic_name
    bundle_version
    blueprint_resource_reference_id
    interface_name
    jndi_name
    authentication_type
    sharing_setting
    authentication_alias_name
    ]]
```

Note: The value for *jndi_name* must match the jndi name that you declare in the **filter** attribute of the resource reference element in the Blueprint XML file.

For example, for an EBA file that contains a bundle `com.ibm.ws.eba.helloWorldService.properties.bundle.jar` at Version 1.0.0, which is to be bound to authentication alias `alias1`, the command is as follows:

```
-BlueprintResourceRefBindingStep[
    [com.ibm.ws.eba.helloWorldService.properties.bundle 1.0.0 resourceRef
    javax.sql.DataSource jdbc/Account Container Shareable alias1]]
```

-WebModuleMsgDestRefs

For an EBA asset, binding a resource reference maps a resource dependency of the web application to an actual resource available in the server runtime environment. At a minimum, this can be achieved by using a mapping that specifies the JNDI name under which the resource is known in the runtime environment. By default, the JNDI name is the resource ID that you specified in the `web.xml` file during development of the web application bundle (WAB). Use this option to bind resources of type message-destination-ref (message destination reference) or resource-env-ref (resource environment reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform.

Specify the syntax of this step as follows:

```
-WebModuleMsgDestRefs [
    [
    bundle_symbolic_name
    bundle_version
    resource_reference_id
    resource_type
    target_jndi_name
    ]]
```

For example:

```
-WebModuleMsgDestRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myQ javax.jms.Queue jms/workQ]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myT javax.jms.Topic jms/notificationTopic]]
```

-WebModuleResourceRefs

For an EBA asset, binding a resource reference maps a resource dependency of the web application to an actual resource available in the server runtime environment. At a minimum, this can be achieved by using a mapping that specifies the JNDI name under which the resource is known in the runtime environment. By default, the JNDI name is the resource ID that you specified in the `web.xml` file during development of the web application bundle (WAB). Use this option to bind resources of type resource-ref (resource reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform.

Specify the syntax of this step as follows:

```
-WebModuleResourceRefs [
    [
    bundle_symbolic_name
    bundle_version
    resource_reference_id
    resource_type
    target_jndi_name
    login_configuration
    login_properties
    extended_properties
    ]]
```

For example:

```
-WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
```

Note: If you use multiple extended properties, the jython syntax is `"extprop1=extval1,extprop2=extval2"`.

Return value

The command returns the configuration IDs of the composition unit and the new composition unit created for the asset in the asset relationship, as the following example displays:

```
WebSphere:cuname=cu4
WebSphere:cuname=cua
WebSphere:cuname=cud
```

Batch mode example usage

Use the following examples to add a non-enterprise asset:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID assetname=asset1.zip -CUOptions
  [[.* .* cu1 "cu1 desc1" 0 false DEFAULT]] -MapTargets [[.* server1]] –ActivationPlanOptions
  [.* specname=actplan0+specname=actplan1]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'myBLA', '–cuSourceID', 'assetname=asset1.zip', '-CUOptions',
  '[[.* .* cu1 "cu1 desc1" 0 false DEFAULT]]', '-MapTargets', '[[.* server1]]', '–ActivationPlanOptions',
   '[.* specname=actplan0+specname=actplan1]'])
```

Use the following examples to add a business-level application composition unit:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID myBLA -cuSourceID yourBLA -CUOptions [[.* .* cu3 "cu3 desc3" 0 false DEFAULT]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'myBLA', '-cuSourceID', 'yourBLA', '-CUOptions',
  '[[.* .* cu3 "cu3 desc3" 0 false DEFAULT]]'])
```

Use the following example to create an EBA composition unit and add it to a business-level application. For more information, see Adding an EBA asset to a composition unit using the addCompUnit command.

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=helloWorldService
  -cuSourceID WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
  -CUOptions [
    [WebSphere:blaname=helloWorldService.eba
    WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
    com.ibm.ws.eba.helloWorldService_0001.eba "" 1 false DEFAULT]]
  -MapTargets [[ebaDeploymentUnit WebSphere:node=node01,server=server1]]
  -ActivationPlanOptions [[default ""]]
  -ContextRootStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 "/hello/web"]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "/hello/service"]]
  -VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0
    "HelloWorld second service" test_host]]
  -MapRolesToUsersStep [
    [ROLE1 No Yes "" ""]
    [ROLE2 No No WABTestUser1 ""]
    [ROLE3 No No "" WABTestGroup1]
    [ROLE4 Yes No "" ""]]
  -BlueprintResourceRefBindingStep[
    [com.ibm.ws.eba.helloWorldService.properties.bundle 1.0.0 resourceRef
    javax.sql.DataSource jdbc/Account Container Shareable alias1]]
  -WebModuleMsgDestRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myQ javax.jms.Queue jms/workQ]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myT javax.jms.Topic jms/notificationTopic]]
  -WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
]')
```

Use the following examples to add a composition unit for a non-enterprise asset and deploy the composition unit to multiple targets:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID theirBLA –cuSourceID asset2.zip –CUOptions
  [[.* .* cu2 "cu2 desc" 0 false DEFAULT]] -MapTargets [[.* server1+server2]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'theirBLA', '–cuSourceID', 'asset2.zip', '–CUOptions',
  '[[.* .* cu2 "cu2 desc" 0 false DEFAULT]]', '-MapTargets', '[[.* server1+server2]]'])
```

Use the following examples to add a composition unit that is a non-enterprise asset with a deployable unit:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID yourBLA –cuSourceID asset2.zip –deplUnits a.jar –CUOptions
  [[.* .* cu3 "cu3 desc" 0 false DEFAULT]] –MapTargets [[a.jar server1]] –ActivationPlanOptions
  [[a.jar specname=actplan1]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'yourBLA', '–cuSourceID', 'asset2.zip', '–deplUnits', 'a.jar', '–CUOptions',
  '[[.* .* cu3 "cu3 desc" 0 false DEFAULT]]', '–MapTargets', '[[a.jar server1]]', '–ActivationPlanOptions',
   '[[a.jar specname=actplan1]]'])
```

Use the following examples to add a composition unit for a non-enterprise asset as a shared library:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID ourBLA –cuSourceID b.jar –deplUnits default –CUOptions
  [[.* .* cub "cub desc" 0 false DEFAULT]] –MapTargets [[default server1]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'ourBLA', '–cuSourceID', 'b.jar', '–deplUnits', 'default', '–CUOptions',
  '[[.* .* cub "cub desc" 0 false DEFAULT]]', '–MapTargets', '[[default server1]]'])
```

Use the following examples to add a composition unit for a non-enterprise asset with a dependency. For this example, the cub composition unit exists as a shared library of the ourBLA business-level application:

- Using Jython string:

```
AdminTask.addCompUnit('-blaID ourBLA –cuSourceID asset3.zip –deplUnits a1.jar –CUOptions
  [[.* .* cu4 "cu4 desc" 0 false DEFAULT]] –MapTargets [[a1.jar cluster1+cluster2]] –CreateAuxCUOptions
  [[a1.jar a.jar cua true]] –RelationshipOptions [[a1.jar cuname=cub true]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'ourBLA', '–cuSourceID', 'asset3.zip', '–deplUnits', 'a1.jar', '–CUOptions',
  '[[.* .* cu4 "cu4 desc" 0 false DEFAULT]]', '–MapTargets', '[[a1.jar cluster1+cluster2]]', '–CreateAuxCUOptions',
  '[[a1.jar a.jar cua true]]', '–RelationshipOptions', '[[a1.jar cuname=cub true]]'])
```

Use the following examples to add an enterprise asset:

- Using Jython string:

```
AdminTask.addCompUnit('[-blaID yourBLA –cuSourceID defaultapp.ear –defaultBindingOptions
  defaultbinding.ejbjndi.prefix=ejb# defaultbinding.virtual.host=default_host#
  defaultbinding.force=yes –AppDeploymentOptions [-appname defaultapp -installed.ear.destination
  application_root/myCell/defaultapp.ear] –MapModulesToServers
  [[defaultapp.war .* WebSphere:cell=cellName,node=nodeName,server=server1]
  [Increment.jar .* Websphere:cell=cellName,node=nodeName,server=server2]] -CtxRootForWebMod
  [[defaultapp.war .* myctx/]]]')
```

- Using Jython list:

```
AdminTask.addCompUnit(['-blaID', 'yourBLA', '–cuSourceID', 'defaultapp.ear', '–defaultBindingOptions',
   'defaultbinding.ejbjndi.prefix=ejb# defaultbinding.virtual.host=default_host# defaultbinding.force=yes',
   '–AppDeploymentOptions', '[-appname defaultapp -installed.ear.destination application_root/myCell/defaultapp.ear]',
   '–MapModulesToServers', '[[defaultapp.war .* WebSphere:cell=cellName,node=nodeName,server=server1]
   [Increment.jar .* Websphere:cell=cellName,node=nodeName,server=server2]]', '-CtxRootForWebMod',
   '[[defaultapp.war .* myctx/]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.addCompUnit('-interactive')
```

deleteCompUnit

The deleteCompUnit command removes a composition unit. Both parameters for this command accept incomplete configuration IDs, as long as the system can match the string to a unique ID.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

-cuID
Specifies the configuration ID of the composition unit to delete. (String, required)

Optional parameters

-force
Specifies whether to force the system to delete the composition unit, even if other composition units depend on this composition unit. (Boolean, optional)

Return value

The command returns the configuration ID of the composition unit that the system deleted, as the following example displays:

```
WebSphere:cuname=cu1
```

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID cu1 -force true')
```

- Using Jython list:

```
AdminTask.deleteCompUnit(['-blaID', 'myBLA', '–cuID', 'cu1', '-force', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteCompUnit('-interactive')
```

editCompUnit

The editCompUnit command modifies additional composition unit options. You can use this command to modify the starting weight of the composition unit, deployment targets, activation plan options, and relationship settings.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

-cuID
Specifies the configuration ID of the composition unit to edit. (String, required)

Optional steps

You can also specify values for optional steps to edit properties of the composition unit. These steps do not apply to enterprise assets. For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the "" characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-CUOptions
Specifies additional properties for the composition unit. Specify the following options with the CUOptions step:

parentBLA (read-only)
Specifies the parent business-level application for the composition unit.

backingID (read-only)
Specifies the composition unit source ID.

name (read-only)
Specifies the name of the composition unit.

description
Specifies a description of the composition unit.

startingWeight
Specifies the starting weight of the composition unit. Supported values are from 1 to 2147483647, the maximum Integer value.

startedOnDistributed
Specifies whether to start the composition unit after distributing changes to the target nodes. The default value is `false`.

restartBehaviorOnUpdate
Specifies the nodes to restart after editing the composition unit. Specify `ALL` to restart each target node. Specify `DEFAULT` to restart the nodes controlled by the sync plug-ins. Specify `NONE` to prevent the system from restarting nodes.

For example, specify the syntax for this step as `-CUOptions [[.* .* cu4 "cu4 description" 0 false DEFAULT]]`

-MapTargets
Specifies additional properties for the composition unit target mapping. Specify the following options with the MapTargets step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI).

For an enterprise bundle archive (EBA) asset, this URI is `ebaDeploymentUnit`.

server Specifies the target or targets to deploy the composition units. The default value is the `server1` server. Use the plus sign character (+) to specify multiple targets. Use the plus sign character (+) as a prefix to add an additional target. Specify the complete object name format for each server that is not a WebSphere Application Server server.

For example, specify the syntax of this step as `-MapTargets [[a1.jar cluster1+cluster2] [a2.jar server1+server2]]`

-ActivationPlanOptions
Specifies additional properties for the composition unit activation plan. Specify the following options with the ActivationPlanOptions step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI).

activationPlan
Specifies a list of runtime components as the activation plan. Specify each activation plan in the format `specName=xxx,specVersion=yyy`, where `specName` represents the name of the specification and is required. Use the plus sign character (+) to specify multiple activation plans.

For example, specify the syntax of this step as `-ActivationPlanOptions [[a1.jar specname=actplan0+actplan1] [a2.jar specname=actplan1+specname=actplan2]]`

For an EBA asset, do not modify the activation plan. Keep the following default values that were set when the composition unit was added: `-ActivationPlanOptions [[default ""]]`

-RelationshipOptions
Specifies additional properties for relationships between assets, composition units, and business-level applications. Use this step if the source ID of the composition unit is an asset that has a matching composition unit in the business-level application. Specify the following options with the RelationshipOptions step:

deplUnit (read-only)
Specifies the deployable unit Uniform Resource Identifier (URI).

relationship
Defines the composition unit relationships. Specify the composition unit object name in the format: `cuName=xxx`. Use the plus sign character (+) to specify multiple composition unit object names in the relationship. If the composition unit specified in the relationship does not exist under the same business-level application, the system returns an error.

matchTarget

Specifies whether to match the targets of the composition unit relationship with the targets of the new composition unit. The default value is `true`.

The product does not save the value specified for matchTarget. Thus if you select to not match the target (`false`) now and later edit the composition unit, when you edit the composition unit you must disable this setting again for the product to not match the targets.

For example, specify the syntax of this step as `–RelationshipOptions [[a1.jar a.jar auxCU true] [a2.jar a.jar defaultCU false]]`

-ContextRootStep

For an EBA asset, context roots determine where the web pages of a particular web application bundle (WAB) are found at run time.

The context root that you specify here is combined with the defined server mapping to compose the full URL that you enter to access the pages of the WAB. For example, if the application server default host is www.example.com:8080 and the context root of the WAB is /sample, the web pages are available at www.example.com:8080/sample.

You should list the context roots for all the WAB modules that are contained in the OSGi application.

Specify the syntax of this step as follows:

```
-ContextRootStep [
  [bundle_symbolic_name_1 bundle_version_1 context_root_1]
  [bundle_symbolic_name_2 bundle_version_2 context_root_2]]
```

For example, for an EBA file containing two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `/hello/web`, and `com.ibm.ws.eba.helloWorldService.withContextRoot` at version 0.9.0, which is to be mapped to `/hello/service`), this aspect of the command is as follows:

```
-ContextRootStep [
  [com.ibm.ws.eba.helloWorldService.web 1.0.0 "/hello/web"]
  [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "/hello/service"]]
```

-VirtualHostMappingStep

For an EBA asset, you use a virtual host to associate a unique port with a module or application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host alias is used in the URL that is used to access artifacts such as servlets and JavaServer Page (JSP) files in a web module. For example, the alias `myhost:8080` is the *host_name:port_number* portion of the URL `http://myhost:8080/sample`.

Each WAB that is contained in a deployed asset must be mapped to a virtual host. WABs can be installed on the same virtual host, or dispersed among several virtual hosts.

If you specify an existing virtual host in the `ibm-web-bnd.xml` or `.xmi` file for a given WAB, the specified virtual host is set by default. Otherwise, the default virtual host setting is `default_host`, which provides several port numbers through its aliases:

80 An internal, insecure port used when no port number is specified
9080 An internal port
9443 An external, secure port

Unless you want to isolate your WAB from other WABs or resources on the same node (physical machine), `default_host` is a suitable virtual host. In addition to `default_host`, WebSphere Application Server provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the WAB relates to system administration.

Specify the syntax of this step as follows:

```
-VirtualHostMappingStep [
    [bundle_symbolic_name_1 bundle_version_1
    web_module_name_1 virtual_host_1]
    [bundle_symbolic_name_2 bundle_version_2
     web_module_name_2 virtual_host_2]]
```

For example, for an EBA file containing two WABs (com.ibm.ws.eba.helloWorldService.web at version 1.0.0, which is to be mapped to default_host, and com.ibm.ws.eba.helloWorldService.withContextRoot at version 0.9.0, which is to be mapped to test_host), this aspect of the command is as follows:

```
-VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "HelloWorld second service" test_host]]
```

-MapRolesToUsersStep

For an EBA asset, use this step to map security roles to users or groups.

Specify the syntax of this step as follows:

```
-MapRolesToUsersStep [
    [role_name everyone?
    all_authenticated_in_realm?
    usernames groups]]
```

Key:

- *role_name* is a role name defined in the application.
- *everyone?* is set to Yes or No, to specify whether or not everyone is in the role.
- *all_authenticated_in_realm?* is set to Yes or No, to specify whether or not all authenticated users can access the application realm.
- *usernames* is a list of WebSphere Application Server user names, separated by the "|" character.
- *groups* is a list of WebSphere Application Server groups, separated by the "|" character.

Note: For *usernames*, and *groups*, the empty string "" means "use the default or existing value". The default value is usually that no users or groups are bound to the role. However, when an application contains an ibm-application-bnd.xmi file, the default value for *usernames* is obtained from this file. If you are deploying an application that contains an ibm-application-bnd.xmi file, and you want to remove the bound users, specify just the "|" character (which is the separator for multiple user names). This explicitly specifies "no users", and therefore guarantees that no users are bound to the role.

For example:

```
-MapRolesToUsersStep [
    [ROLE1 No Yes "" ""]
    [ROLE2 No No WABTestUser1 ""]
    [ROLE3 No No "" WABTestGroup1]
    [ROLE4 Yes No "" ""]]
```

-BlueprintResourceRefPostDeployStep

For an EBA asset, Blueprint components can access WebSphere Application Server resource references. Each reference is declared in a Blueprint XML file, and can be secured using a Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) authentication alias. Each bundle in an OSGi application can contain any number of resource reference declarations in its various Blueprint XML files.

When you secure resource references, those resource references can be bound only to JCA authentication aliases that exist on every server or cluster that the OSGi application is deployed to. An OSGi application can be deployed to multiple servers and clusters that are in the same security domain. Therefore, each JCA authentication alias must exist in either the security domain of the target servers and clusters, or the global security domain. For more information, see Modifying the configuration of an EBA composition unit using the editCompUnit command.

Specify the syntax of this step as follows:

```
-BlueprintResourceRefPostDeployStep [
    [
    bundle_symbolic_name
```

```
    bundle_version
    blueprint_resource_reference_id
    interface_name
    jndi_name
    authentication_type
    sharing_setting
    authentication_alias_name
    ]]
```

Note: The value for *jndi_name* must match the jndi name that you declare in the **filter** attribute of the resource reference element in the Blueprint XML file.

For example, for an EBA file that contains a bundle `com.ibm.ws.eba.helloWorldService.properties.bundle.jar` at Version 1.0.0, which is to be bound to authentication alias `alias1`, the command is as follows:

```
-BlueprintResourceRefPostDeployStep[
    [com.ibm.ws.eba.helloWorldService.properties.bundle 1.0.0 resourceRef
    javax.sql.DataSource jdbc/Account Container Shareable alias1]]
```

-WebModuleResourceRefs

For an EBA asset, binding a resource reference maps a resource dependency of the web module to an actual resource available in the server runtime environment. At a minimum, this can be achieved by using a mapping that specifies the JNDI name under which the resource is known in the runtime environment. By default, the JNDI name is the resource ID that you specified in the `web.xml` file during development of the web application bundle (WAB).

Specify the syntax of this step as follows:

```
-WebModuleResourceRefs [
    [
    bundle_symbolic_name
    bundle_version
    resource_reference_id
    resource_type
    target_jndi_name
    login_configuration
    login_properties
    extended_properties
    ]]
```

For example:

```
-WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
```

Note: If you use multiple extended properties, the jython syntax is `"extprop1=extval1,extprop2=extval2"`.

Return value

The command returns the configuration ID of the composition unit that the system edits.

Batch mode example usage

Use the following examples to edit a composition unit of an asset and replace the target from existing targets:

- Using Jython string:

```
AdminTask.editCompUnit('-blaID myBLA –cuID cu1 –CUOptions [[.* .* cu1 cudesc 1 false DEFAULT]] -MapTargets
  [[.* server2]] -ActivationPlanOptions [.* #specname=actplan0+specname=actplan2]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'myBLA', '–cuID', 'cu1', '–CUOptions',
  '[[.* .* cu1 cudesc 1 false DEFAULT]]', '-MapTargets', '
  [[.* server2]]', '-ActivationPlanOptions', '[.* #specname=actplan0+specname=actplan2]'])
```

Use the following examples to edit a composition unit of an asset and its relationships:

- Using Jython string:

```
AdminTask.editCompUnit('-blaID ourBLA –cuID cu4 –CUOptions [[.* .* cu4 "new cu desc" 1 false DEFAULT]]
   –MapTargets [[a1.jar server1+server2]] –RelationshipOptions [[a1.jar cuname=cub true]]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'ourBLA', '–cuID', 'cu4', '–CUOptions', '
  [[.* .* cu4 "new cu desc" 1 false DEFAULT]]', '–MapTargets', '[[a1.jar server1+server2]]', '–RelationshipOptions',
  '[[a1.jar cuname=cub true]]'])
```

Use the following examples to edit a composition unit by adding a new relationship to the existing relationship:

- Using Jython string:

```
AdminTask.editCompUnit('[-blaID ourBLA –cuID cu4 –CUOptions [[.* .* cu4 "new cu desc" 1 false DEFAULT]]
   –MapTargets [[a1.jar server1+server2]] –RelationshipOptions [[a1.jar +cuname=cuc true]] -ActivationPlanOptions
  [a1.jar +specname=actplan2#specname=actplan1]]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'ourBLA', '–cuID', 'cu4', '–CUOptions', '
  [[.* .* cu4 "new cu desc" 1 false DEFAULT]]', '–MapTargets', '[[a1.jar server1+server2]]', '–RelationshipOptions',
   '[[a1.jar +cuname=cuc true]]', '-ActivationPlanOptions', '[a1.jar +specname=actplan2#specname=actplan1]'])
```

Use the following examples to edit an enterprise composition unit configuration:

- Using Jython string:

```
AdminTask.editCompUnit('-blaID yourBLA –cuID defaultapp –MapModulesToServers
  [[defaultapp.war .* WebSphere:cluster=cluster1][Increment.jar .* Websphere:cluster=cluster2]]
   –CtxRootForWebMod [[defaultapp.war .* /]] –MapWebModToVH [[defaultapp.war .* vh1]]')
```

- Using Jython list:

```
AdminTask.editCompUnit(['-blaID', 'yourBLA', '–cuID', 'defaultapp', '–MapModulesToServers',
  '[[defaultapp.war .* WebSphere:cluster=cluster1][Increment.jar .* Websphere:cluster=cluster2]]', '–CtxRootForWebMod',
  '[[defaultapp.war .* /]]', '–MapWebModToVH', '[[defaultapp.war .* vh1]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editCompUnit('-interactive')
```

listCompUnits

The listCompUnits command displays each composition unit that is associated with a specific business-level application.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

Optional parameters

-includeDescription
Specifies whether to include a description of each asset that the command returns. (String, optional)

-includeType
Specifies whether to include the type for each asset that the command returns. (String, optional)

Return value

The command returns a list of configuration IDs and the type for each composition unit, as the following example displays:

```
Websphere:cuname=cu1
asset
"description for cu1"
Websphere:cuname=cu4
bla
"description for cu4"
WebSphere:cuname=defaultapp
Java EE
"description for defaultapp"
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listCompUnits('-blaID blaname=theirBLA')
```

- Using Jython list:

```
AdminTask.listCompUnits(['-blaID', 'blaname=theirBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listCompUnits('-interactive')
```

setCompUnitTargetAutoStart

The setCompUnitTargetAutoStart command enables or disables automatic starting of composition units. If you enable this option, the system automatically starts the composition unit when the composition unit target starts.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. The command accepts an incomplete configuration ID if the system matches it to a unique business-level application ID. (String, required)

-cuID
Specifies the composition unit of interest. The command accepts an incomplete configuration ID if the system matches it to a unique composition unit ID. (String, required)

-targetID
Specifies the name of the target of interest. For example, specify the server name to set the target to a specific server. (String, required)

-enable
Specifies whether to automatically start the composition unit of interest when the specified target starts. Specify `true` to start the composition unit automatically. If you do not specify `true`, the system will not start the composition unit when the target starts. The default value is `true`. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.setCompUnitTargetAutoStart('-blaID bla1 –cuID cu1 –targetID server1 –enable true')
```

- Using Jython list:

```
AdminTask.setCompUnitTargetAutoStart(['-blaID', 'bla1', '–cuID', 'cu1', '–targetID',
   'server1', '–enable', 'true'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.setCompUnitTargetAutoStart('-interactive')
```

viewCompUnit

The viewCompUnit command displays configuration information for a composition unit that belongs to a specific business-level application.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. This parameter accepts an incomplete configuration ID if the system matches it to a unique business-level application ID. (String, required)

-cuID
Specifies the configuration ID of the composition unit of interest. This parameter accepts an incomplete configuration ID if the system matches it to a unique composition unit ID. (String, required)

Optional parameters

None

Return value

The command returns configuration information for the composition unit of interest, as the following example displays:

```
Specify Composition Unit options (CUOptions)

Specify name, description options for Composition Unit.

Parent BLA (parentBLA): [WebSphere:blaname=myBLA]
Backing Id (backingId): [WebSphere:assetname=asset1.zip]
Name (name): [cu1]
Description (description): [cuDesc]
Starting Weight (startingWeight): [0]
Started on distributed (startedOnDistributed): [false]
Restart behavior on update (restartBehaviorOnUpdate): [DEFAULT]

Specify servers (MapTargets)

Specify targets such as application servers or clusters of application servers where you want to deploy the cu contained in the application.

Deployable Unit (deplUnit): [default]
*Servers (server): [WebSphere:node=myNode,server=server1]

Specify Composition Unit activation plan options (ActivationPlanOptions)

Specify CU activation plan optionsDeployableUnit Name (deplUnit): [default]
Activation Plan (activationPlan): [WebSphere:specname=actplan0+WebSphere:specname=actplan1]
```

If the composition unit contains an enterprise bundle archive (EBA) asset, the composition unit status is also displayed. This status is one of the following values:

- Using latest OSGi application deployment.
- New OSGi application deployment not yet available because it requires bundles that are still downloading.
- New OSGi application deployment available.

- New OSGi application deployment cannot be applied because bundle downloads have failed.

For more information, see Checking the update status of an OSGi composition unit.

Batch mode example usage

The following example displays configuration information for a non-enterprise asset:

- Using Jython string:

```
AdminTask.viewCompUnit('-blaID myBLA -cuID myCompUnit')
```

- Using Jython list:

```
AdminTask.viewCompUnit(['-blaID', 'myBLA', '-cuID', 'myCompUnit'])
```

The following example displays configuration information for an enterprise asset:

- Using Jython string:

```
AdminTask.viewCompUnit('-blaID myBLA -cuID defaultApplication')
```

- Using Jython list:

```
AdminTask.viewCompUnit(['-blaID', 'myBLA', '-cuID', 'defaultApplication'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.viewCompUnit('-interactive')
```

createEmptyBLA

The createEmptyBLA command to create an empty business-level application. After creating a business-level application, you can add assets or other business-level applications as composition units to the application.

Target object

None

Required parameters

-name
Specifies a unique name for the new business-level application. (String, required)

Optional parameters

-description
Specifies a description of the new business-level application. (String, optional)

Return value

The command returns the configuration ID of the new business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

Batch mode example usage

- Using Jython string:

```
AdminTask.createEmptyBLA('-name myBLA -description "my description for BLA"')
```

- Using Jython list:

```
AdminTask.createEmptyBLA(['-name', 'myBLA', '-description', '"my description for BLA"'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createEmptyBLA('-interactive')
```

deleteBLA

The deleteBLA command removes a business-level application from your configuration. Before deleting a business-level application, use the deleteCompUnit command to remove each configuration unit that is associated with the business-level application. Also, verify that no other business-level applications reference the business-level application to delete.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. The command accepts an incomplete ID for the blaID parameter, as long as the system can match the string to a unique identifier. For example, you can specify the `myBLA` partial ID to identify the `WebSphere:blaname=myBLA` configuration ID. (String, required)

Optional parameters

None

Return value

The command returns the configuration ID of the deleted business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteBLA('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.deleteBLA(['-blaID', 'myBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteBLA('-interactive')
```

editBLA

The editBLA command modifies the description of a business-level application.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

Optional steps

For optional steps, use the `.*` characters to specify a read-only argument in the command syntax. Specify an empty string with the `""` characters to keep the existing value of the argument. If you do not specify a value or an empty string for a writable argument, the command resets the argument to a null value.

-BLAOptions
Use the BLAOptions step to specify a new description for the business-level application of interest.

name (read-only)
Specifies the name of the business-level application.

description
Specifies a description of the business-level application.

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.editBLA('-blaID DefaultApplication –BLAOptions [[.* "my new description"]]')
```

- Using Jython list:

```
AdminTask.editBLA(['-blaID', 'DefaultApplication', '–BLAOptions', '[[.* "my new description"]]'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.editBLA('-interactive')
```

getBLAStatus

The getBLAStatus command displays whether a business-level application or composition unit is running or stopped.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. Use the listBLAs command to display a list of business-level application configuration IDs. (String, required)

Optional parameters

-cuID
Specifies the configuration ID of the composition unit of interest. Use the listCompUnits command to display a list of composition unit configuration IDs. (String, optional)

Return value

The command returns the status of the business-level application or composition unit of interest.

Batch mode example usage

- Using Jython string:

```
AdminTask.getBLAStatus('-blaID WebSphere:blaname=myBLA -cuID Websphere:cuname=cu1')
```

- Using Jython list:

```
AdminTask.getBLAStatus(['-blaID', 'WebSphere:blaname=myBLA', '-cuID', 'Websphere:cuname=cu1'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.getBLAStatus('-interactive')
```

listBLAs

The listBLAs command displays the business-level applications in your configuration.

Target object

None

Optional parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, optional)

-includeDescription
Specifies whether to include a description of each business-level application that the command returns. Specify `true` to display the business-level application descriptions. (String, optional)

Return value

The command returns a list of configuration IDs for each business-level application in your configuration, as the following example displays:

```
WebSphere:blaname=myBLA
WebSphere:blaname=yourBLA
```

Batch mode example usage

The following example lists each business-level application in the configuration:

- Using Jython:

```
AdminTask.listBLAs()
```

Use the following examples to list each business-level application in the configuration:

- Using Jython string:

```
AdminTask.listBLAs('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.listBLAs(['-blaID', 'myBLA'])
```

Use the following examples to list each business-level application and the corresponding descriptions:

- Using Jython string:

```
AdminTask.listBLAs('-includeDescription true')
```

- Using Jython list:

```
AdminTask.listBLAs(['-includeDescription', 'true'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.listBLAs('-interactive')
```

listControlOps

The listControlOps command displays the control operations for a business-level application and the corresponding composition units.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. (String, required)

Optional parameters

-cuID
Specifies the composition unit of interest. (String, optional)

-opName
Specifies the operation name of interest. (String, optional)

-long
Specifies whether to include additional configuration information in the command output. (String, optional)

Return value

The command returns a list of operations, operation descriptions, and parameter descriptions for the query scope, as the following example displays:

```
"Operation: start"
"   Description: Start operation"
"   Operation handler ID: com.mycompany.myasset.ControlOpHandler"
"   Operation handler data URI: None"
"Operation: stop"
"   Description: Stop operation"
"   Operation handler ID: com.mycompany.myasset.ControlOpHandler"
"   Operation handler data URI: None"
"Operation: clearCache"
"   Description: Clears specified cache or all caches"
"   Operation handler ID: com.mycompany.myasset.ControlOpHandler"
"   Operation handler data URI: None"
"   Parameter: cacheName"
"      Description: Name of cache to clear.  If not specified, all caches are cleared."
```

Batch mode example usage

- Using Jython string:

```
AdminTask.listControlOps('-blaID myBLA –cuID myservice.jar –long true')
```

- Using Jython list:

```
AdminTask.listControlOps(['-blaID', 'myBLA', '–cuID', 'myservice.jar', '–long true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listControlOps('-interactive')
```

startBLA

The startBLA command starts the business-level application of interest.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application to start. The command accepts an incomplete configuration ID if the system matches the string to a unique ID in your configuration. (String, required)

Return value

The command returns a status message if the business-level application starts. If the business-level application does not start, the command does not return output. The following example displays the status message output:

```
BLA ID of started BLA if the BLA was not already running.
```

Batch mode example usage

- Using Jython string:

```
AdminTask.startBLA('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.startBLA(['-blaID', 'myBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.startBLA('-interactive')
```

stopBLA

The stopBLA command stops the business-level application of interest.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application to stop. The command accepts an incomplete configuration ID if the system matches the string to a unique ID in your configuration. (String, required)

Return value

The command returns a status message if the business-level application stops. If the business-level application does not stop, the command does not return output. The following example displays the status message output:

```
BLA ID of stopped BLA if the BLA was not already stopped.
```

Batch mode example usage

- Using Jython string:

```
AdminTask.stopBLA('-blaID myBLA')
```

- Using Jython list:

```
AdminTask.stopBLA(['-blaID', 'myBLA'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.stopBLA('-interactive')
```

viewBLA

The viewBLA command displays the name and description of the business-level application of interest.

Target object

None

Required parameters

-blaID
Specifies the configuration ID of the business-level application of interest. The command accepts an incomplete configuration ID if the system matches the string to a unique business-level application. (String, required)

Optional parameters

None

Return value

The command returns configuration information for the business-level application of interest, as the following example displays:

```
Specify BLA options (BLAOptions)

Specify options for BLA

*BLA Name (name): [DefaultApplication]
BLA Description (description): []
```

Batch mode example usage

- Using Jython string:

```
AdminTask.viewBLA('-blaID DefaultApplication')
```

- Using Jython list:

```
AdminTask.viewBLA(['-blaID', 'DefaultApplication'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.viewBLA('-interactive')
```

JSFCommands command group for the AdminTask object

You can use the Jython scripting language to display and modify the JavaServer Faces (JSF) implementation.

Use the following command to administer JSF:

- "listJSFImplementation"
- "modifyJSFImplementation" on page 364

listJSFImplementation

The listJSFImplementation command displays the JSF implementation and version for a specific application.

Target object

Specify the name of the application of interest. (String, required)

Required parameters

None.

Return value

The command displays the JSF implementation and version. For example, if the command returns "SUNRI1.2", then JSF uses version 1.2 of the Sun Reference Implementation.

Batch mode example usage

- Using Jython string:

```
AdminTask.listJSFImplementation('application1')
```

- Using Jython list:

```
AdminTask.listJSFImplementation('application1')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.listJSFImplementation('-interactive')
```

modifyJSFImplementation

The modifyJSFImplementation command modifies the JSF implementation for a specific application.

Target object

Specify the name of the application of interest. (String, required)

Required parameters

-implName
Specifies the name of the implementation to use. Specify SUNRI1.2 to use the Sun Reference 1.2 Implementation, or specify MyFaces to use the Apache MyFaces 2.0 project implementation. By default, applications use MyFaces. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.modifyJSFImplementation('-implName MyFaces')
```

- Using Jython list:

```
AdminTask.modifyJSFImplementation('-implName', 'MyFaces')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.modifyJSFImplementation('-interactive')
```

Application management command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage applications with the wsadmin tool. Use the commands and parameters in the AppManagementCommands group can be used to display and process SQL Java (SQLJ) profiles or IBM Optim PureQuery Runtime bind files.

The AppManagementCommands command group for the AdminTask object includes the following commands:

listSqljProfiles

The listSqljProfiles command parses the enterprise archive (EAR) file of the specified application and returns a list of SQLJ files. SQLJ profiles have a .ser file name extension. If there are any files in the EAR file that are not SQLJ profiles, but have a .ser file name extension, those files can be listed also.

Parameters and return values

-appName
The name of the installed application. Your application must be installed before running customization and binding on it. This parameter is required.

Examples

Batch mode example usage:

- Using JACL:

```
$AdminTask listSqljProfiles {-appName application_name}
```

- Using Jython:

```
print AdminTask.listSqljProfiles('-appName application_name')
```

Interactive mode example usage:

- Using JACL:

```
$AdminTask listSqljProfiles -interactive
```

- Using Jython:

```
print AdminTask.listSqljProfiles('-interactive')
```

The output displays with syntax specific to the local operating system. The list of available profiles can be added to a group file, with a .grp extension, directly.

processSqljProfiles

The processSqljProfiles command creates a DB2 customization of the SQLJ profiles. The command optionally, by default, calls the SQLJ profile binder to bind the DB2 packages.

Note: If you are processing a large enterprise application, or you are processing many SQLJ profiles, the process might take longer than the default timeout for the wsadmin tool. The default connection timeout for the wsadmin tool is set to 3 minutes. If the default timeout is reached and you lose the connection to the server, the wsadmin console issues a timeout statement. You can check the system output log for the final results of the customization and bind process and the amount of time for that the process. Do not start the processSqljProfiles command again until the previous command has completed, or the results might be unpredictable.

To prevent this disconnection, configure the session timeout to a longer time. See the system output log for the total processing time, and use that time period as a basis for the new timeout value. To extend the default timeout value, change the wsadmin properties file that corresponds to the connection type that you are using:

- For the SOAP connection type, change the following entry in the soap.client.props file:

```
com.ibm.SOAP.requestTimeout=180
```

- For JSR160RMI and RMI connection types, change the following entry in the sas.client.props file:

```
com.ibm.CORBA.requestTimeout=180
```

- For the IPC connection type, change the following entry in the ipc.client.props file:

```
com.ibm.IPC.requestTimeout=180
```

There are two ways you can verify if the binding or customization took place:

- If you performed a customization process, you can run a query from the command line to see the application EAR files that were changed:

```
wsadmin>print AdminConfig.hasChanges()
```

  The query returns `0` if there are no changes, and `1` if changes occurred on the server. To view the configuration files that have unsaved changes, run:

```
wsadmin>print AdminConfig.queryChanges()
```

  .
- View the System Out log to determine if the binding or processing was successful.

Target object

The installed application SQLJ profiles. These profiles are either single, serial .ser files or profiles that are grouped in a .grp group file. This target object is required.

Parameters and return values

-appName
The name of the installed application. Your application must be installed before running customization and binding on it. This parameter is required.

-classpath
The path that tells the application server where to find the necessary SQLJ driver JAR files. This parameter is optional.

-dburl
The location of the DB2 server on the network. This parameter is optional.

-user
User name of the account performing the access to the DB2 database. This parameter is optional.

-password
Password for the account accessing the DB2 database. This parameter is optional.

-options
Additional options that are used with the **db2sqljcustomize** command might be inserted under the **-options** parameter except for the parameters listed previously. This parameter is optional. For additional information about the **db2sqljcustomize** command, consult db2sqljcustomize - SQLJ profile customizer.

-profiles
The location of the SQLJ profiles .ser files or .grp file. This parameter is required.

Examples

Batch mode example usage:

Interactive mode example usage:

```
wsadmin>print AdminTask.processSqljProfiles('-interactive') Process serialized SQLJ
 profiles. Process the serialized SQLJ profiles in an installed application.  Customize the profiles with run time information and
 bind static SQL packages in a database.   Refer to the Database SQLJ customize and bind documentation.    Do only bind
 processing. (bindOnly): false  *Application name. (appName): Application  Classpath to SQLJ tools. (classpath):
 C:/IBM/SQLLIB/java/db2jcc.jar  Database connection URL. (dbURL):  Database connection user name. (user):  Database connection
 password. (password):  Options for SQLJ tools. (options):  *SQLJ profile names. (profiles): c:/temp/ApplicationSerNames.grp
 Process serialized SQLJ profiles.    F (Finish) C (Cancel)   Select [F, C]: [F]  WASX7278I: Generated command line:
 AdminTask.processSqljProfiles('[-bindOnly false -appName Application -classpath [C:/IBM/SQLLIB/java/db2jcc.jar] -profiles
 [c:/temp/ApplicationSerNames.grp ]]')
```

listPureQueryBindFiles

The listPureQueryBindFiles command parses the EAR file of the specified application and returns a list of .bindprops and .pdqxml files that are found. PureQuery bind options files have a .bindprops file name extension. Bind files have a .pdqxml file name extension. If the EAR file contains files that are not pureQuery bind files, but have a .bindprops or a .pdqxml file name extension, those files can also be listed.

Parameters and return values

-appName
The name of the installed application. This parameter is required.

Examples

Batch mode example usage:

- Using JACL:

```
$AdminTask listPureQueryBindFiles {-appName application_name}
```

- Using Jython:

```
print AdminTask.listPureQueryBindFiles('-appName application_name')
```

Interactive mode example usage:

- Using JACL:

```
$AdminTask listPureQueryBindFiles -interactive
```

- Using Jython:

```
print AdminTask.listPureQueryBindFiles('-interactive')
```

The output displays with syntax specific to the local operating system.

processPureQueryBindFiles

The processPureQueryBindFiles command invokes the DB2 pureQuery bind utility on a list of pureQuery bind files.

Note: If you are processing a large enterprise application, or you are processing many pureQuery bind files using wsadmin, the process might take longer than the default timeout for the wsadmin tool. The default connection timeout for the wsadmin tool is set to 3 minutes. If the default timeout is reached and the process running on the server has not yet completed, the wsadmin console issues a timeout statement. You can check the system output log on the server for the final results of the bind process and the time when that process completed. Do not starta the processPureQueryBindFiles command again until the previous command has completed, or the results might be unpredictable.

To prevent this timeout, configure the wsadmin request timeout to a longer time. After a successful customization and binding process, use the system output log to estimate the total processing time. Use this time period as a basis for the new timeout value. To extend the default timeout value, change the wsadmin properties file that corresponds to the connection type that you are using:

- For the SOAP connection type, change the following entry in the soap.client.props file:

```
com.ibm.SOAP.requestTimeout=180
```

- For JSR160RMI and RMI connection types, change the following entry in the sas.client.props file:

```
com.ibm.CORBA.requestTimeout=180
```

- For the IPC connection type, change the following entry in the ipc.client.props file:

```
com.ibm.IPC.requestTimeout=180
```

To verify whether the binding took place, view the System Out log to determine if the bind processing was successful.

Parameters and return values

-appName

The name of an installed application that contains the pureQuery bind files to be processed. Your application must be installed before running binding on it.

-classpath

A list of the paths to the Java archive (JAR) files that contain the IBM Optim PureQuery Runtime bind utility and its dependencies: pdq.jar, pdqmgmt.jar, db2jcc4.jar or db2jcc.jar, db2jcc_license_cisuz.jar or db2jcc_license_cu.jar. Use / or \\\\ as a file separator. Use a blank space to separate the paths for the JAR files.

-dburl

The URL for connecting to the database. The format is jdbc:db2://*server_name*:*port*/*database_name*.

-user

User name of the account performing the access to the DB2 database.

-password

Password for the account accessing the DB2 database.

-options

Any additional options that are needed by the IBM Optim PureQuery Runtime bind utility. Provide bind options as **-bindoptions "bind_options_string"**. For additional information about the IBM Optim PureQuery Runtime bind utility, consult the topic about the pureQuery Bind utility.

-files

A list of the names of the pureQuery bind files to be processed. The bind file path names must be relative to the application EAR file that contains them. Use / or \\\\ as a file separator. If you specify multiple profile paths, use a blank space to separate them.

Examples

Batch mode example usage:

Interactive mode example usage:

```
print AdminTask.processPureQueryBindFiles('-interactive') Process pureQuery bind files.
 Process the pureQuery bind files in an installed application.  Bind static SQL packages in a database.  Refer to the pureQuery
 Bind utility documentation.  *Application name. (appName): MyApp Classpath to pureQuery Bind utility. (classpath):
 /pdq_home/pdq.jar /pdq_home/pdqmgmt.jar /db2_home/SQLLIB/java/db2jcc4.jar /db2_home/SQLLIB/java/db2jcc_license_cu.jar *Database
 connection URL. (url): jdbc:db2://hostname:50000/databasename Database connection user name. (user): dbuser1 Database connection
 password. (password): dbpswrd1 Options for the pureQuery Bind utility. (options): -bindoptions "BLOCKING NO" *pureQuery bind file
 names. (files): META-INF/xyz.bindprops META-INF/abc.bindprops  Process pureQuery bind files.  F (Finish) C (Cancel)  Select [F,
 C]: [F] WASX7278I: Generated command line: AdminTask.processPureQueryBindFiles('[-appName MyApp -classpath [/pdq_home/pdq.jar
 /pdq_home/pdqmgmt.jar /db2_home/SQLLIB/java/db2jcc4.jar /db2_home/SQLLIB/java/db2jcc_license_cu.jar ] -url
 jdbc:db2://hostname:50000/databasename -user dbuser1 -password ***** -options [-bindoptions "BLOCKING NO"] -files
 [META-INF/xyz.bindprops META-INF/abc.bindprops ]]')
```

Chapter 13. Managing deployed applications using wsadmin scripting

Use these topics to learn more about managing deployed applications with the wsadmin tool and scripting.

Procedure

- Start enterprise applications and stop enterprise applications. You can use the wsadmin tool and the AdminControl object to start an application that is not running (has a status of Stopped) or stop an application that is running (has a status of Started).
- Start business-level applications and stop business-level applications. You can use the wsadmin tool and the BLAManagement command group to start and stop business-level applications.
- Update applications. Use the wsadmin tool to update installed applications on an application server.
- Manage assets. Use the wsadmin tool and commands in the BLAManagement command group to manage your asset configuration. This topic provides examples for listing assets, viewing asset configuration data, removing assets from the asset repository, updating one or more files for assets, and exporting assets.
- Manage composition units. Use the wsadmin tool and commands in the BLAManagement command group to manage composition units. This topic provides examples for adding, removing, editing, exporting, and viewing composition units.
- List application modules. Use the wsadmin tool and the AdminApp object listModules command to list the modules in an installed application.
- Query the application state. Use the wsadmin tool and scripting to determine if an application is running.
- Export applications. You can use the wsadmin tool and the AdminApp object to export your applications.

Starting applications using wsadmin scripting

Use scripting and the wsadmin tool to start an application that is not running.

Before you begin

There are two ways to complete this task. This topic uses the AdminControl object to start an application. Alternatively, you can use the scripts in the AdminApplication script library to start, stop, and manage applications.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the application manager MBean for the server where the application resides and assign it the appManager variable. The following example returns the name of the application manager MBean.
   - Using Jacl:

     ```
     set appManager [$AdminControl queryNames cell=mycell,node=mynode,type
     =ApplicationManager,
     process=server1,*]
     ```

   - Using Jython:

     ```
     appManager = AdminControl.queryNames('cell=mycell,node=mynode,type
     =ApplicationManager,
     process=server1,*')
     print appManager
     ```

Table 389. queryNames command elements. Run the queryNames command to get the name of the application manager MBean.

`set`	is a Jacl command
`appManager`	is a variable name

Table 389. queryNames command elements (continued). Run the queryNames command to get the name of the application manager MBean.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`queryNames`	is an AdminControl command
`cell=mycell,node=mynode,type=ApplicationManager ,process=server1`	is the hierarchical containment path of the configuration object
`print`	is a Jython command

Example output:

```
WebSphere:cell=mycell,name=ApplicationManager,mbeanIdentifier=ApplicationManager,
type=ApplicationManager,node=mynode,process=server1
```

3. Start the application. The following example invokes the startApplication operation on the MBean, providing the application name that you want to start.
   - Using Jacl:

     ```
     $AdminControl invoke $appManager startApplication myApplication
     ```

   - Using Jython:

     ```
     AdminControl.invoke(appManager, 'startApplication', 'myApplication')
     ```

Table 390. invoke command elements. Run the invoke command to start the application.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`invoke`	is an AdminControl command
`appManager`	evaluates to the ID of the server that is specified in step number 1
`startApplication`	is an attribute of the modify command
myApplication	is the value of the startApplication attribute

Starting business-level applications using wsadmin scripting

You can use the wsadmin tool and the BLAManagement command group to start business-level applications.

Before you begin

There are two ways to complete this task. Use the BLAManagement command group for the AdminTask object or the scripts in the AdminBLA script library to start your business-level applications.

Procedure

- Use the AdminTask object commands to start business-level applications.
  1. Start the wsadmin scripting tool.
  2. List the business-level applications in your environment.

     Use the listBLAs command to display a list of business-level applications in your environment, as the following example demonstrates:

```
AdminTask.listBLAs()
```

     You can optionally specify the partial name of the business-level application of interest to display the configuration ID of the business-level application. The command accepts a partial business-level application name if the system matches the specified name to a unique configuration ID. Use the following example to set the configuration ID of the `myBLA` business-level application to the `blaID` variable:

```
myBLA=AdminTask.listBLAs('-blaID BLA1')
```

3. Determine the status of the business-level application.

   Use the getBLAStatus command to display the status of the business-level application of interest, as the following example demonstrates:

```
AdminTask.getBLAStatus('-blaID myBLA')
```

   The command returns the status of the business-level application as `STOPPED` or `STARTED`.

4. Start the business-level application.

   Use the startBLA command to start the business-level application, as the following example demonstrates:

```
AdminTask.startBLA('-blaID myBLA')
```

   The command returns the following message if the system successfully starts the business-level application:

```
BLA ID of started BLA if the BLA was not already running.
```

- Use the Jython script library to start business-level applications.
  1. Start the wsadmin scripting tool.
  2. List the business-level applications in your environment.

     Use the listBLAs script to display a list of business-level applications in your environment, using the following syntax:

```
AdminBLA.listBLAs(blaName, displayDescription)
```

     You can specify one, both, or neither the blaName and displayDescription arguments. Use the blaName argument to specify the name of a specific business-level application, and the displayDescription argument to specify whether to display the description of each returned business-level application. Specify an empty string in place of arguments that you do not want to specify, as the following example demonstrates:

```
AdminBLA.listBLAs("", "true")
```

  3. Start the business-level application.

     Use the startBLA script to start the business-level application, using the following syntax:

```
AdminBLA.startBLA(blaName)
```

     Use the blaName argument to specify the name of the business-level application to start, as the following example demonstrates:

```
AdminBLA.startBLA("myBLA")
```

Stopping applications using wsadmin scripting

You can use the wsadmin tool to stop applications.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminControl object to stop the application. Alternatively, you can use the scripts in the AdminApplication script library to start, stop, and administer your application configurations.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the application manager MBean for the server where the application resides, and assign it to the appManager variable.
   - Using Jacl:

     ```
     set appManager [$AdminControl queryNames cell=mycell,node=mynode,type=
     ApplicationManager,process=server1,*]
     ```

   - Using Jython:

     ```
     appManager = AdminControl.queryNames('cell=mycell,node=mynode,type=
     ApplicationManager,process=server1,*')
     print appManager
     ```

Table 391. queryNames command elements. Run the queryNames command to get the name of the application manager MBean.

`set`	is a Jacl command
`appManager`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`queryNames`	is an AdminControl command
`cell=`*`mycell`*`,node=`*`mynode`*`,type=`*`ApplicationManager`*`,process=`*`server1`*	is the hierarchical containment path of the configuration object
`print`	is a Jython command

This command returns the application manager MBean.

Example output:

```
WebSphere:cell=mycell,name=ApplicationManager,mbeanIdentifier=ApplicationManager,
type=ApplicationManager,node=mynode,process=server1
```

3. Query the running applications belonging to this server and assign the result to the apps variable.
   - Using Jacl:

     ```
     set apps [$AdminControl queryNames cell=mycell,node=mynode,type=Application,
     process=server1,*]
     ```

   - Using Jython:

     ```
     # get line separator
     import  java.lang.System  as sys
     lineSeparator = sys.getProperty('line.separator')

     apps = AdminControl.queryNames('cell=mycell,node=mynode,type=Application,
     process=server1,*').split(lineSeparator)
     print apps
     ```

Table 392. queryNames command elements. Run the queryNames command to query running applications.

`set`	is a Jacl command
`apps`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`queryNames`	is an AdminControl command
`cell=`*`mycell`*`,node=`*`mynode`*`,type=`*`ApplicationManager`*`,process=`*`server1`*	is the hierarchical containment path of the configuration object
`print`	is a Jython command

This command returns a list of application MBeans.

Example output:

```
WebSphere:cell=mycell,name=adminconsole,mbeanIdentifier=deployment.xml
#ApplicationDeployment_1,type=Application,node=mynode,Server=server1,
process=server1,J2EEName=adminconsole
WebSphere:cell=mycell,name=filetransfer,mbeanIdentifier=deployment.xml
#ApplicationDeployment_1,type=Application,node=mynode,Server=server1,
process=server1,J2EEName=filetransfer
```

4. Stop all the running applications.
   - Using Jacl:

```
foreach app $apps {
     set appName [$AdminControl getAttribute $app name]
     $AdminControl invoke $appManager stopApplication $appName}
```

- Using Jython:

```
for app in apps:
    appName = AdminControl.getAttribute(app, 'name')
    AdminControl.invoke(appManager, 'stopApplication', appName)
```

This command stops all the running applications by invoking the stopApplication operation on the MBean, passing in the application name to stop.

Results

Once you complete the steps for this task, all running applications on the server are stopped.

Stopping business-level applications using wsadmin scripting

You can use the wsadmin tool and the BLAManagement command group to stop business-level applications.

Before you begin

There are two ways to complete this task. Use the BLAManagement command group for the AdminTask object or the scripts in the AdminBLA script library to stop your business-level applications.

Procedure

- Use the AdminTask object to stop business-level applications.
  1. Start the wsadmin scripting tool.
  2. List the business-level applications in your environment.

     Use the listBLAs command to display a list of business-level applications in your environment, as the following example demonstrates:

```
AdminTask.listBLAs()
```

     You can optionally specify the partial name of the business-level application of interest to display the configuration ID of the business-level application. The command accepts a partial business-level application name if the system matches the specified name to a unique configuration ID. Use the following example to set the configuration ID of the `myBLA` business-level application to the `blaID` variable:

```
myBLA=AdminTask.listBLAs('-blaID BLA1')
```

  3. Determine the status of the business-level application.

     Use the getBLAStatus command to display the status of the business-level application of interest, as the following example demonstrates:

```
AdminTask.getBLAStatus('-blaID myBLA')
```

     The command returns the status of the business-level application as `STOPPED` or `RUNNING`.
  4. Stop the running business-level application.

     Use the stopBLA command to stop the business-level application, as the following example demonstrates:

```
AdminTask.stopBLA('-blaID myBLA')
```

     The command returns the following message if the system successfully stops the business-level application:

```
BLA ID of stopped BLA if the BLA was not already stopped.
```

- Use the Jython script library to stop business-level applications.
  1. Start the wsadmin scripting tool.
  2. List the business-level applications in your environment.

Use the listBLAs script to display a list of business-level applications in your environment, using the following syntax:

```
AdminBLA.listBLAs(blaName, displayDescription)
```

You can specify one, both, or neither the blaName and displayDescription arguments. Use the blaName argument to specify the name of a specific business-level application, and the displayDescription argument to specify whether to display the description of each returned business-level application. Specify an empty string in place of arguments that you do not want to specify, as the following example demonstrates:

```
AdminBLA.listBLAs("", "true")
```

3. Stop the business-level application.

   Use the stopBLA script to stop the business-level application, using the following syntax:

```
AdminBLA.stopBLA(blaName)
```

Use the blaName argument to specify the name of the business-level application to stop, as the following example demonstrates:

```
AdminBLA.stopBLA("myBLA")
```

Updating installed applications using the wsadmin scripting tool

Use the wsadmin tool and scripting to update installed applications on an application server.

About this task

Both the **update** command and the **updateinteractive** command support a set of options. You can also obtain a list of supported options for an Enterprise Archive (EAR) file using the **options** command, for example:

Using Jacl:

```
$AdminApp options
```

Using Jython:

```
print AdminApp.options()
```

You can set or update a configuration value using options in batch mode. To identify which configuration object is to be set or updated, the values of read only fields are used to find the corresponding configuration object. All the values of read only fields have to match with an existing configuration object, otherwise the command fails.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Perform the following steps to update an application:

Procedure

1. Start the wsadmin scripting tool.
2. Update the installed application using one of the following options.
   - The following command updates a single file in a deployed application:
     - Using Jacl:

```
$AdminApp update app1 file {-operation update -contents
 /home/myProfile/apps/app1/my.xml -contenturi app1.jar/my.xml}
```

     - Using Jython string:

```
AdminApp.update('app1', 'file', '[-operation update -contents /home/myProfile/
 apps/app1/my.xml -contenturi app1.jar/my.xml]')
```

– Using Jython list:

```
AdminApp.update('app1', 'file', ['-operation', 'update', '-contents',
 '/home/myProfile/apps/app1/my.xml', '-contenturi', 'app1.jar/my.xml'])
```

Table 393. update file command elements. Run the update command to change an installed application file.

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
file	is the content type value
operation	is an option of the **update** command
update	is the value of the operation option
contents	is an option of the **update** command
/apps/app1/my.xml	is the value of the contents option
contenturi	is an option of the **update** command
app1.jar/my.xml	is the value of the contenturi option

- The following command adds a module to the deployed application, if the module does not exist. Otherwise, the existing module is updated.

  – Using Jacl:

```
$AdminApp update app1 modulefile {-operation addupdate -contents
 /home/myProfile/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb  -BindJndiForEJBNonMessageBinding {{"Increment EJB
 module"  Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
```

  – Using Jython string:

```
AdminApp.update('app1', 'modulefile', '[-operation addupdate -contents
 /home/myProfile/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb -BindJndiForEJBNonMessageBinding [["Increment EJB
 module" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
```

  – Using Jython list:

```
bindJndiForEJBValue = [["Increment EJB module", "Increment", "
 Increment.jar,META-INF/ejb-jar.xml", "Inc"]]  AdminApp.update('app1', 'modulefile', ['-operation', 'addupdate', '-contents',
 '/home/myProfile/apps/app1/Increment.jar', '-contenturi','Increment.jar' '-nodeployejb', `-BindJndiForEJBNonMessageBinding',
 bindJndiForEJBValue])
```

Table 394. update modulefile command elements. Run the update command to change an installed module file.

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
modulefile	is the content type value
operation	is an option of the **update** command
addupdate	is the value of the operation option
contents	is an option of the **update** command
/apps/app1/Increment.jar	is the value of the contents option
contenturi	is an option of the **update** command
Increment.jar	is the value of the contenturi option
nodeployejb	is an option of the **update** command
BindJndiForEJBNonMessageBinding	is an option of the **update** command

Table 394. update modulefile command elements (continued). Run the update command to change an installed module file.

"Increment EJB module" Increment Increment.jar,META-INF/ejb-jar.xml Inc	is the value of the BindJndiForEJBNonMessageBinding option. The value of this option is defined in your application configuration. To determine the value of this option, use the following Jython or Jacl command: Using Jython: `AdminApp.view('myAppName')` Using Jacl: `$AdminApp view myAppName`
bindJndiForEJBValue	is a Jython variable that contains the value of the BindJndiForEJBNonMessageBinding option

- The following command uses a partial application to update a deployed application:
  - Using Jacl:

```
$AdminApp update app1 partialapp {-contents
 /home/myProfile/apps/app1/app1Partial.zip}
```

  - Using Jython string:

```
AdminApp.update('app1', 'partialapp', '[-contents
 /home/myProfile/apps/app1/app1Partial.zip]')
```

  - Using Jython list:

```
AdminApp.update('app1', 'partialapp', ['-contents',
 '/home/myProfile/apps/app1/app1Partial.zip'])
```

Table 395. update partialapp command elements. Run the update command to change part of an installed application.

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
partialapp	is the content type value
contents	is an option of the **update** command
/apps/app1/app1Partial.zip	is the value of the contents option

- Update the entire deployed application.

  To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the `wsadmin.properties` file in the properties directory of the profile of interest.

  - Using Jacl:

```
$AdminApp update app1 app {-operation update -contents
 /home/myProfile/apps/app1/newApp1.jar -usedefaultbindings -nodeployejb -BindJndiForEJBNonMessageBinding {{"Increment EJB module"
 Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
```

  - Using Jython string:

```
AdminApp.update('app1', 'app', '[-operation update -contents
 /home/myProfile/apps/app1/ newApp1.ear -usedefaultbindings -nodeployejb -BindJndiForEJBNonMessageBinding [["Increment EJB module"
 Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
```

  - Using Jython list:

```
bindJndiForEJBValue = [["Increment EJB module", "Increment", "
 Increment.jar,META-INF/ejb-jar.xml", "Inc"]]  AdminApp.update('app1', 'app', ['-operation', 'update', '-contents',
 '/apps/app1/NewApp1.ear', '-usedefaultbindings', '-nodeployejb', `-BindJndiForEJBNonMessageBinding', bindJndiForEJBValue])
```

Table 396. update app command elements. Run the update command to change an installed application.

$	is a Jacl operator for substituting a variable name with its value
AdminApp	is an object that supports application objects management
update	is an AdminApp command
app1	is the name of the application to update
app	is the content type value
operation	is an option of the **update** command
update	is the value of the operation option
contents	is an option of the **update** command
/apps/app1/newApp1.ear	is the value of the contents option
usedefaultbindings	is an option of the **update** command
nodeployejb	is an option of the **update** command
BindJndiForEJBNonMessageBinding	is an option of the **update** command
"Increment EJB module" Increment Increment.jar,META-INF/ejb-jar.xml Inc	is the value of the BindJndiForEJBNonMessageBinding option. The value of this option is defined in your application configuration. To determine the value of this option, use the following Jython or Jacl command: Using Jython: `AdminApp.view('myAppName')` Using Jacl: `$AdminApp view myAppName`
bindJndiForEJBValue	is a Jython variable containing the value of the BindJndiForEJBNonMessageBinding option

3. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

What to do next

The steps in this task return a success message if the system successfully updates the application. When updating large applications, the command might return a success message before the system extracts each binary file. You cannot start the application until the system extracts all binary files. If you installed a large application, use the **isAppReady** and **getDeployStatus** commands for the AdminApp object to verify that the system extracted the binary files before starting the application.

The **isAppReady** command returns a value of `true` if the system is ready to start the application, or a value of `false` if the system is not ready to start the application.

```
AdminApp.isAppReady('myapp1')
```

If the system is not ready to start the application, the system might be expanding application binaries. Use the **getDeployStatus** command to display additional information about the binary file expansion status, as the following example displays:

```
AdminApp.getDeployStatus('app1')
```

Running the **getDeployStatus** command where *app1* is `DefaultApplication` results in status information about DefaultApplication resembling the following:

```
ADMA5071I: Distribution status check started for application DefaultApplication.
WebSphere:cell=myCell01,node=myNode01,distribution=unknown,expansion=unknown
ADMA5011I: The cleanup of the temp directory for application DefaultApplication is complete.
ADMA5072I: Distribution status check completed for application DefaultApplication.
WebSphere:cell=myCell01,node=myNode01,distribution=unknown,expansion=unknown
```

Managing assets using wsadmin scripting

Use the commands in the BLAManagement command group to manage your asset configuration. Use the examples in this topic to list assets, view asset configuration data, remove assets from the asset repository, update one or more files for assets, and export assets.

Before you begin

There are two ways to complete this task. Complete the tasks in this topic to manage assets with the BLAManagement command group for the AdminTask object. Alternatively, you can use the scripts in the AdminBLA script library to administer your asset configurations.

Procedure

- List assets.
  1. Start the wsadmin scripting tool.
  2. List the assets registered to the asset repository.

     Use the listAssets command to display the configuration ID, description, and deployment target for each asset within the cell, as the following command demonstrates:

```
AdminTask.listAssets()
```

- View asset settings.
  1. Start the wsadmin scripting tool.
  2. Display the asset settings.

     Use the viewAsset command to display the configuration information for the asset of interest, as the following example demonstrates:

```
AdminTask.viewAsset('-assetID myAsset.zip')
```

     The command returns the configured asset options, as the following sample output displays:

```
Specify Asset options (AssetOptions)  Specify options for Asset.  *Asset Name (name):
 [defaultapp.ear] Default Binding Properties (defaultBindingProps):
 [defaultbinding.ejbjndi.prefix#defaultbinding.datasource.jndi#
  defaultbinding.datasource.username# defaultbinding.datasource.password# defaultbinding.cf.jndi#
  defaultbinding.cf.resauth#defaultbinding.virtual.host# defaultbinding.force]
  Asset Description (description): [] Asset Binaries
  Destination Url (destination): [${USER_INSTALL_ROOT}/installedAssets/defaultapp.ear/BASE/defaultapp.ear]
  Asset Type Aspects(typeAspect): [WebSphere:spec=j2ee_ear] Asset Relationships (relationship):
  []File Permission (filePermission):
  [.*\\.dll=755#.*\\.so=755#.*\\.a=755#.*\\.sl=755] Validate asset (validate): [false]
```

- Remove one or more assets from the product management domain.
  1. Start the wsadmin scripting tool.
  2. Determine if the asset can be deleted.

     You cannot delete an asset from the asset registry if it is associated with composition unit in a business-level application. Use the listCompUnits command to display the configuration ID, type, and description for each composition unit in a business-level application, as the following example demonstrates:

```
AdminTask.listCompUnits('-blaID myBLA -includeDescription true')
```

     The command returns the following sample output:

```
Websphere:cuname=cu1 asset "Composition unit for asset.zip"  Websphere:cuname=cu4 bla  "cu4
 description" WebSphere:cuname=defaultapp __j2ee "defaultapp description"
```

The type for the cu1 composition unit is asset, which denotes that the composition unit is associated with an asset. Use the deleteCompUnit command to remove the composition unit before deleting the asset from the asset repository, as the following example demonstrates:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID cu1')
```

3. Delete the asset.

   Use the deleteAsset command to remove the asset of interest from the asset repository, as the following example demonstrates:

```
AdminTask.deleteAsset('-assetID asset2.zip')
```

   The command returns the configuration ID of the deleted asset, as the following example displays:

```
WebSphere:assetname=asset2.zip
```

- Update the contents of an asset.
  1. Start the wsadmin scripting tool.
  2. Determine how to update the asset.

     You can invoke several different operations on assets that are registered in the asset repository, as the following table displays:

Table 397. updateAsset supported operations. Run an updateAsset command with an operation.

Operation	Description
replace	The replace operation replaces the contents of the asset of interest.
merge	The merge operation updates multiple files for the asset, but does not update all files.
add	The add operation adds a new file or module file.
addupdate	The addupdate operation adds or updates one file or module file. If the file does not exist, the system adds the contents. If the file exists, the system updates the file.
update	The update operation updates one file or module file.
delete	The delete operation deletes a file or module file.

  3. Update the asset of interest.

     The updateAsset command modifies one or more files or module files of an asset, as the following example demonstrates:

```
AdminTask.updateAsset('-assetID asset2.zip –operation merge –contents
 \temp\updatedFiles_asset1.zip')
```

     The command updates the asset binary file, but does not update the composition unit that the system deploys with the asset as a backing object.
  4. Save your configuration changes.

     Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

- Export an asset to a target location.
  1. Start the wsadmin scripting tool.
  2. Export the asset of interest.

     Use the exportAsset command to save an asset configuration to a file. The command accepts an incomplete asset configuration ID if the system matches it to a unique ID in your configuration. The following example exports an asset:

```
AdminTask.exportAsset('-assetID asset2.zip –filename \temp\a2.zip')
```

Managing composition units using wsadmin scripting

Use the commands in the BLAManagement command group to manage composition units. Use the examples in this topic to add, remove, edit, export, and view composition units.

Before you begin

There are two ways to complete the examples in this task. Use the BLAManagement command group for the AdminTask object to manage composition units. Alternatively, you can use the scripts in the AdminBLA

script library to administer your composition unit configurations.

About this task

Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-WebSphere Application Server runtime environments without associated assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

Procedure

- Add composition units.
  1. Start the wsadmin scripting tool.
  2. Add composition units.

     Use the addCompUnit command to add composition units to business-level applications.

     Note: If the asset is an enterprise bundle archive (EBA) asset, there are additional parameters to set. For more information, see Adding an EBA asset to a composition unit using the addCompUnit command.

     Use the following command example to add the `asset1` asset as a composition unit in the `myBLA` business-level application, and map the deployment to the `server1` server:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset1 -CUOptions [[.* .*
 compositionUnit1 "composition unit that is backed by asset1" 0]] -MapTargets
[[.* server1]]
 –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

     Use the following command to add the `asset2` asset as a composition unit in the `myBLA` business-level application, and map the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset2 -CUOptions [[.* .*
 compositionUnit2 "composition unit that is backed by asset2" 0]] -MapTargets [[.*
 server1+testServer]] –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

     Use the following command to add the `J2EEAsset` asset as a composition unit in the `myBLA` business-level application, and map the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('[-blaID myBLA –cuSourceID J2EEAsset
 –defaultBindingOptions defaultbinding.ejbjndi.prefix=ejb# defaultbinding.virtual.host=default_host#
 defaultbinding.force=yes –AppDeploymentOptions [-appname defaultapp -installed.ear.destination
 application_root/myCell/defaultapp.ear] –MapModulesToServers [[defaultapp.war
 .* WebSphere:cell=cellName,node=nodeName,server=server1][Increment.jar .*
 Websphere:cell=cellName,node=nodeName,server=testServer]] -CtxRootForWebMod [[defaultapp.war .*
 myctx/]]]')
```

     The command returns the configuration IDs of the composition unit and the new composition unit created for the asset in the asset relationship, as the following example displays:

```
WebSphere:cuname=compositionUnit1 WebSphere:cuname=compositionUnit2
 WebSphere:cuname=J2EEAsset
```

  3. Save your configuration changes.

     Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

- Display composition units and configuration settings.

  Use the listCompUnits and viewCompUnits commands to display the configuration IDs of each composition unit that matches a specific search scope.

  You can use the listCompUnits command to display each composition unit in your configuration or within a specific business-level application. The following example displays each composition unit in the `myBLA` business-level application:

```
AdminTask.listCompUnits('-blaID blaname=myBLA')
```

  The command returns the configuration IDs and type of backing asset for each composition unit that matches the search scope, as the following sample displays:

```
Websphere:cuname=cu1 asset  Websphere:cuname=cu4 bla  WebSphere:cuname=defaultapp
 __j2ee
```

You can use the viewCompUnits command to display additional configuration information about a specific composition unit of a business-level application. For example, the following example displays additional information about the cu1 composition unit for the myBLA business-level application:

```
AdminTask.viewCompUnit('-blaID myBLA -cuID cu1')
```

The command returns detailed configuration information for the composition unit, as the following sample displays:

```
Specify Composition Unit options (CUOptions)  Specify name, description options for
 Composition Unit.  Parent BLA (parentBLA): [WebSphere:blaname=myBLA] Backing Id (backingId):
[WebSphere:assetname=asset1.zip]
 Name (name): [cu1] Description (description): [my description of cu1 composition unit] Starting Weight
 (startingWeight): [0]
 Specify servers (MapTargets)  Specify targets such as application servers or clusters of application servers
where you want to deploy the composition unit contained in the application.  Deployable Unit (deplUnit):
 [default] *Servers (server):
 [WebSphere:node=myNode,server=server1]  Specify Composition Unit activation plan options (ActivationPlanOptions)
Specify composition unit activation plan optionsDeployableUnit Name (deplUnit):
 [default] Activation Plan (activationPlan):
 [WebSphere:specname=actplan0+WebSphere:specname=actplan1]
```

If the composition unit contains an enterprise bundle archive (EBA) asset, the composition unit status is also displayed. This status is one of the following values:

- Using latest OSGi application deployment.
- New OSGi application deployment not yet available because it requires bundles that are still downloading.
- New OSGi application deployment available.
- New OSGi application deployment cannot be applied because bundle downloads have failed.

For more information, see Checking the update status of an OSGi composition unit.

- Edit composition units.
  1. Start the wsadmin scripting tool.
  2. Modify the composition unit.

     Use the editCompUnit command to modify composition unit options. You can use this command to modify the starting weight of the composition unit, deployment targets, activation plan options, and relationship settings. See the documentation for the BLAManagement command group for the AdminTask object to view descriptions of each option that you can modify.

     Note: If the composition unit contains an enterprise bundle archive (EBA) asset, there are additional parameters that you can modify. For more information, see Modifying the configuration of an OSGi composition unit using the editCompUnit command.

     The following example edits a composition unit, which is associated with an asset, and replaces the deployment target:

```
AdminTask.editCompUnit('-blaID myBLA –cuID cu1 –CUOptions [[.* .* cu1
 cudesc 1]] -MapTargets [[.* server2]] -ActivationPlanOptions [.*
 #specname=actplan0+specname=actplan2]')
```

     The command returns the configuration ID of the composition unit that the system edits, as the following sample displays:

```
WebSphere:cuname=cu1
```

  3. Save your configuration changes.

     Use the following command example to save your configuration changes:

     ```
     AdminConfig.save()
     ```

- Remove composition units.
  1. Start the wsadmin scripting tool.
  2. Remove composition units.

     Use the deleteCompUnit command to remove a composition unit. Both parameters for the following command accept incomplete configuration IDs, as long as the system can match the string to a unique ID:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID cu1')
```

The command returns the configuration ID of the composition unit that the system deletes, as the following sample demonstrates:

```
WebSphere:cuname=cu1
```

3. Save your configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Listing the modules in an installed application using wsadmin scripting

Use the AdminApp object **listModules** command to list the modules in an installed application.

About this task

You can run the listModules command to see what modules are in an installed application.

Procedure

1. Start the wsadmin scripting tool.
2. Display the application modules.

   Using Jacl:

```
$AdminApp listModules DefaultApplication -server
```

   Using Jython:

```
print AdminApp.listModules('DefaultApplication', '-server')
```

Table 398. listmodules command elements. Run the listmodules command to list application modules.

`$`	is a Jacl operator for substituting a variable name with its value
`print`	is a Jython command
`AdminApp`	is an object that supports application object management
`listmodules`	is an AdminApp command
`DefaultApplication`	is the name of the application
`-server`	is an optional option specified

   Example output:

```
DefaultApplication#IncCMP11.jar+META-INF/ejb-jar.xml#WebSphere:cell=mycell,node=mynode,server=myserver
DefaultApplication#DefaultWebApplication.war+WEB-INF/web.xml#WebSphere:cell=mycell,node=mynode,server=myserver
```

Example

The following example lists all of the modules on all of the enterprise applications that are installed on the `server1` server in a node named `node1`.

An asterisk (*****) means that the module is installed on server1 and node1 and another server or node. A plus sign (**+**) means that the module is installed on server1 and node1 only.

```
 1  #-------------------------------------------------------------------------------
       2  # setting up variables to keep server name and node name
       3  #-------------------------------------------------------------------------
       4  set  serverName  server1
       5  set  nodeName  node1
       6  #-------------------------------------------------------------------------------
       7  # setting up 2 global lists to keep the modules
       8  #-------------------------------------------------------------------------------
       9  set  ejbList {}
      10 set webList {}
11
12 #-------------------------------------------------------------------------------
13 # gets all deployment objects and assigned it to deployments variable
14 #-------------------------------------------------------------------------------
```

```
15 set deployments [$AdminConfig getid /Deployment:/]
16
17 #-------------------------------------------------------------------------------------------
18 # lines 22 thru 148 Iterates through all the deployment objects to get the modules
19 # and perform filtering to list application that has at least one module installed
20 # in server1 in node myNode
21 #-------------------------------------------------------------------------------------------
22 foreach deployment $deployments {
23
24     # ------------------------------------------------------------------------------------
25     # reset the lists that hold modules for each application
26     #-------------------------------------------------------------------------------------
27     set webList {}
28     set ejbList {}
29
30     #-----------------------------------------
31     # get the application name
32     #-----------------------------------------
33     set appName [lindex [split $deployment (] 0]
34
35     #-----------------------------------------
36     # get the deployedObjects
37     #-----------------------------------------
38     set depObject [$AdminConfig showAttribute $deployment deployedObject]
39
40     #-----------------------------------------
41     # get all modules in the application
42     #-----------------------------------------
43     set modules [lindex [$AdminConfig showAttribute $depObject modules] 0]
44
45     #-------------------------------------------------------------------------------------------
46     # initialize lists to save all the modules in the appropriate list to where they belong
47     #-------------------------------------------------------------------------------------------
48     set modServerMatch {}
49     set modServerMoreMatch {}
50     set modServerNotMatch {}
51
52         #-------------------------------------------------------------------------------------
53         # lines 55 to 112 iterate through all modules to get the targetMappings
54         #-------------------------------------------------------------------------------------
55         foreach module $modules {
56             #-------------------------------------------------------------------------------
57             # setting up some flag to do some filtering and get modules for server1 on node1
58             #-------------------------------------------------------------------------------
59             set sameNodeSameServer "false"
60            set diffNodeSameServer "false"
61             set sameNodeDiffServer "false"
62             set diffNodeDiffServer "false"
63
64             #-----------------------------------------
65             # get the targetMappings
66             #-----------------------------------------
67             set targetMaps [lindex [$AdminConfig showAttribute $module targetMappings] 0]
68
69             #-------------------------------------------------------------------------------
70            # lines 72 to 111 iterate through all targetMappings to get the target
71             #-------------------------------------------------------------------------------
72             foreach targetMap $targetMaps {
73                 #-----------------------------
74                # get the target
75                 #-----------------------------
76                 set target [$AdminConfig showAttribute $targetMap target]
77
78                 #-------------------------------------------------
79                 # do filtering to skip ClusteredTargets
80                 #-------------------------------------------------
81                 set targetName [lindex [split $target #] 1]
82                if {[regexp "ClusteredTarget" $targetName] != 1} {
83                     set sName [$AdminConfig showAttribute $target name]
84                     set nName [$AdminConfig showAttribute $target nodeName]
85
86                     #-------------------------------------------
```

```
87                      # do the server name match
88                      #-------------------------------------------
89                      if {$sName == $serverName} {
90                        if {$nName == $nodeName} {
91                  set sameNodeSameServer "true"
92                        } else {
93                  set diffNodeSameServer "true"
94                        }
95                     } else {
96                           #---------------------------------------
97                           # do the node name match
98                           #---------------------------------------
99                           if {$nName == $nodeName} {
100                  set sameNodeDiffServer "true"
101                       } else {
102                           set diffNodeDiffServer "true"
103                       }
104                  }
105
106                  if {$sameNodeSameServer == "true"} {
107                      if {$sameNodeDiffServer == "true" || $diffNodeDiffServer == "true" ||
             $diffNodeSameServer == "true"} {
108                          break
109                      }
110                 }
111             }
112     }
113
114     #-------------------------------------------
115     # put it in the appropriate list
116     #-------------------------------------------
117     if {$sameNodeSameServer == "true"} {
118         if {$diffNodeDiffServer == "true" || $diffNodeSameServer == "true" ||
      $sameNodeDiffServer == "true"} {
119             set modServerMoreMatch [linsert $modServerMoreMatch end
       [$AdminConfig showAttribute $module uri]]
120         } else {
121             set modServerMatch [linsert $modServerMatch end [$AdminConfig showAttribute $module uri]]
122         }
123     } else {
124         set modServerNotMatch [linsert $modServerNotMatch end [$AdminConfig showAttribute $module uri]]
125     }
126 }
127
128
129 #-----------------------------------------------------------------
130 # print the output with some notation as a mark
131 #-----------------------------------------------------------------
132 if {$modServerMatch != {} || $modServerMoreMatch != {}} {
133     puts stdout "\tApplication name: $appName"
134 }
135
136 #---------------------------------------------------------
137 # do grouping to appropriate module and print
138 #---------------------------------------------------------
139 if {$modServerMatch != {}} {
140     filterAndPrint $modServerMatch "+"
141 }
142 if {$modServerMoreMatch != {}} {
143     filterAndPrint $modServerMoreMatch "*"
144 }
145 if {($modServerMatch != {} || $modServerMoreMatch != {}) "" $modServerNotMatch != {}} {
146     filterAndPrint $modServerNotMatch ""
147 }
148}
149
150
151 proc filterAndPrint {lists flag} {
152    global webList
153    global ejbList
154    set webExists "false"
155    set ejbExists "false"
```

```
156
157     #-------------------------------------------------------------------------------------
158     # If list already exists, flag it so as not to print the title more then once
159     # and reset the list
160     #-------------------------------------------------------------------------------------
161     if {$webList != {}} {
162         set webExists "true"
163         set webList {}
164     }
165     if {$ejbList != {}} {
166         set ejbExists "true"
167         set ejbList {}
168     }
169
170     #---------------------------------------------------------------
171     # do some filtering for web modules and ejb modules
172     #---------------------------------------------------------------
173     foreach list $lists {
174          set temp [lindex [split $list .] 1]
175          if {$temp == "war"} {
176               set webList [linsert $webList end $list]
177          } elseif {$temp == "jar"} {
178               set ejbList [linsert $ejbList end $list]
179          }
180     }
181
182     #--------------------------------------
183     # sort the list before printing
184     #--------------------------------------
185     set webList [lsort -dictionary $webList]
186     set ejbList [lsort -dictionary $ejbList]
187
188     #---------------------------------------------------------------
189     # print out all the web modules installed in server1
190     #---------------------------------------------------------------
191     if {$webList != {}} {
192         if {$webExists == "false"} {
193              puts stdout "\t\tWeb Modules:"
194         }
195         foreach web $webList {
196             puts stdout "\t\t\t$web  $flag"
197         }
198     }
199
200     #-------------------------------------------------------------
201     # print out all the ejb modules installed in server1
202     #-------------------------------------------------------------
203     if {$ejbList != {}} {
204          if {$ejbExists == "false"} {
205               puts stdout "\t\tEJB Modules:"
206          }
207          foreach ejb $ejbList {
208               puts stdout "\t\t\t$ejb  $flag"
209          }
210     }
211}

Example output for server1 on node node1:
       Application name: TEST1
                EJB Modules:
                        deplmtest.jar  +
                Web Modules:
                        mtcomps.war  *
        Application name: TEST2
                Web Modules:
                        mtcomps.war  +
                EJB Modules:
                        deplmtest.jar  +
        Application name: TEST3
                Web Modules:
                        mtcomps.war  *
                EJB Modules:
```

```
                deplmtest.jar  *
 Application name: TEST4
         EJB Modules:
                deplmtest.jar  *
         Web Modules:
                mtcomps.war
```

Example: Listing the modules in an application server

This example lists all of the modules on all of the enterprise applications that are installed on the `server1` server in a node named `node1`.

An asterisk (*****) means that the module is installed on server1 and node1 and another server or node. A plus sign (**+**) means that the module is installed on server1 and node1 only.

```
 1  #-------------------------------------------------------------------------------------------------
      2  # setting up variables to keep server name and node name
      3  #-------------------------------------------------------------------------------
      4  set  serverName  server1
      5  set  nodeName  node1
      6  #-------------------------------------------------------------------------------------------------
      7  # setting up 2 global lists to keep the modules
      8  #-------------------------------------------------------------------------------------------------
      9  set  ejbList {}
     10 set webList {}
11
12 #-------------------------------------------------------------------------------------------------
13 # gets all deployment objects and assigned it to deployments variable
14 #-------------------------------------------------------------------------------------------------
15 set deployments [$AdminConfig getid /Deployment:/]
16
17 #-------------------------------------------------------------------------------------------------
18 # lines 22 thru 148 Iterates through all the deployment objects to get the modules
19 # and perform filtering to list application that has at least one module installed
20 # in server1 in node myNode
21 #-------------------------------------------------------------------------------------------------
22 foreach deployment $deployments {
23
24     # -------------------------------------------------------------------------------------------------
25     # reset the lists that hold modules for each application
26     #-------------------------------------------------------------------------------------------------
27     set webList {}
28     set ejbList {}
29
30     #-----------------------------------------
31     # get the application name
32     #-----------------------------------------
33     set appName [lindex [split $deployment (] 0]
34
35     #-----------------------------------------
36     # get the deployedObjects
37     #-----------------------------------------
38     set depObject [$AdminConfig showAttribute $deployment deployedObject]
39
40     #-----------------------------------------
41     # get all modules in the application
42     #-----------------------------------------
43     set modules [lindex [$AdminConfig showAttribute $depObject modules] 0]
44
45     #-------------------------------------------------------------------------------------------------
46     # initialize lists to save all the modules in the appropriate list to where they belong
47     #-------------------------------------------------------------------------------------------------
48     set modServerMatch {}
49     set modServerMoreMatch {}
50     set modServerNotMatch {}
51
52         #-------------------------------------------------------------------------------------------------
53         # lines 55 to 112 iterate through all modules to get the targetMappings
54         #-------------------------------------------------------------------------------------------------
55         foreach module $modules {
56             #-------------------------------------------------------------------------------------------
```

```
57            # setting up some flag to do some filtering and get modules for server1 on node1
58            #-------------------------------------------------------------------------------
59            set sameNodeSameServer "false"
60           set diffNodeSameServer "false"
61            set sameNodeDiffServer "false"
62            set diffNodeDiffServer "false"
63
64            #-----------------------------------------
65            # get the targetMappings
66            #-----------------------------------------
67            set targetMaps [lindex [$AdminConfig showAttribute $module targetMappings] 0]
68
69            #-------------------------------------------------------------------------------
70           # lines 72 to 111 iterate through all targetMappings to get the target
71            #-------------------------------------------------------------------------------
72            foreach targetMap $targetMaps {
73                #-----------------------------
74               # get the target
75                #-----------------------------
76                set target [$AdminConfig showAttribute $targetMap target]
77
78                #-------------------------------------------------
79                # do filtering to skip ClusteredTargets
80                #-------------------------------------------------
81                set targetName [lindex [split $target #] 1]
82               if {[regexp "ClusteredTarget" $targetName] != 1} {
83                    set sName [$AdminConfig showAttribute $target name]
84                    set nName [$AdminConfig showAttribute $target nodeName]
85
86                    #-----------------------------------------
87                    # do the server name match
88                    #-----------------------------------------
89                    if {$sName == $serverName} {
90                      if {$nName == $nodeName} {
91                set sameNodeSameServer "true"
92                        } else {
93                set diffNodeSameServer "true"
94                        }
95                   } else {
96                         #---------------------------------------
97                         # do the node name match
98                         #---------------------------------------
99                         if {$nName == $nodeName} {
100                set sameNodeDiffServer "true"
101                       } else {
102                           set diffNodeDiffServer "true"
103                       }
104                }
105
106                if {$sameNodeSameServer == "true"} {
107                    if {$sameNodeDiffServer == "true" || $diffNodeDiffServer == "true" ||
           $diffNodeSameServer == "true"} {
108                           break
109                    }
110                }
111           }
112     }
113
114     #-----------------------------------------
115     # put it in the appropriate list
116     #-----------------------------------------
117     if {$sameNodeSameServer == "true"} {
118         if {$diffNodeDiffServer == "true" || $diffNodeSameServer == "true" || $sameNodeDiffServer == "true"} {
119              set modServerMoreMatch [linsert $modServerMoreMatch end [$AdminConfig showAttribute $module uri]]
120          } else {
121              set modServerMatch [linsert $modServerMatch end [$AdminConfig showAttribute $module uri]]
122         }
123    } else {
124         set modServerNotMatch [linsert $modServerNotMatch end [$AdminConfig showAttribute $module uri]]
125    }
126 }
127
```

```
128
129 #-------------------------------------------------------------
130 # print the output with some notation as a mark
131 #-------------------------------------------------------------
132 if {$modServerMatch != {} || $modServerMoreMatch != {}} {
133     puts stdout "\tApplication name: $appName"
134 }
135
136 #---------------------------------------------------------
137 # do grouping to appropriate module and print
138 #---------------------------------------------------------
139 if {$modServerMatch != {}} {
140     filterAndPrint $modServerMatch "+"
141 }
142 if {$modServerMoreMatch != {}} {
143     filterAndPrint $modServerMoreMatch "*"
144 }
145 if {($modServerMatch != {} || $modServerMoreMatch != {}) "" $modServerNotMatch != {}} {
146     filterAndPrint $modServerNotMatch ""
147 }
148}
149
150
151 proc filterAndPrint {lists flag} {
152    global webList
153    global ejbList
154    set webExists "false"
155    set ejbExists "false"
156
157    #-----------------------------------------------------------------------------------------
158    # If list already exists, flag it so as not to print the title more then once
159    # and reset the list
160    #-----------------------------------------------------------------------------------------
161    if {$webList != {}} {
162        set webExists "true"
163        set webList {}
164    }
165    if {$ejbList != {}} {
166        set ejbExists "true"
167        set ejbList {}
168    }
169
170    #-------------------------------------------------------------------
171    # do some filtering for web modules and ejb modules
172    #-------------------------------------------------------------------
173    foreach list $lists {
174         set temp [lindex [split $list .] 1]
175         if {$temp == "war"} {
176              set webList [linsert $webList end $list]
177         } elseif {$temp == "jar"} {
178              set ejbList [linsert $ejbList end $list]
179         }
180    }
181
182    #---------------------------------------
183    # sort the list before printing
184    #---------------------------------------
185    set webList [lsort -dictionary $webList]
186    set ejbList [lsort -dictionary $ejbList]
187
188    #-------------------------------------------------------------------
189    # print out all the web modules installed in server1
190    #-------------------------------------------------------------------
191    if {$webList != {}} {
192        if {$webExists == "false"} {
193             puts stdout "\t\tWeb Modules:"
194        }
195        foreach web $webList {
196            puts stdout "\t\t\t$web  $flag"
197        }
198    }
199
```

```
200     #-------------------------------------------------------------
201     # print out all the ejb modules installed in server1
202     #-------------------------------------------------------------
203     if {$ejbList != {}} {
204          if {$ejbExists == "false"} {
205               puts stdout "\t\tEJB Modules:"
206          }
207          foreach ejb $ejbList {
208               puts stdout "\t\t\t$ejb  $flag"
209          }
210     }
211}

Example output for server1 on node node1:
       Application name: TEST1
                EJB Modules:
                        deplmtest.jar  +
                Web Modules:
                        mtcomps.war  *
        Application name: TEST2
                Web Modules:
                        mtcomps.war  +
                EJB Modules:
                        deplmtest.jar  +
        Application name: TEST3
                Web Modules:
                        mtcomps.war  *
                EJB Modules:
                        deplmtest.jar  *
        Application name: TEST4
                EJB Modules:
                        deplmtest.jar  *
                Web Modules:
                        mtcomps.war
```

Querying the application state using wsadmin scripting

Use the wsadmin tool and scripting to determine if an application is running.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

Procedure

1. Start the wsadmin scripting tool.
2. Determine the application state.

   The following example queries the presence of the Application MBean to find out whether the application is running.

   - Using Jacl:

```
$AdminControl completeObjectName type=Application,name=myApplication,*
```

   - Using Jython:

```
print AdminControl.completeObjectName('type=Application,name=myApplication,*')
```

Table 399. completeObjectName command elements. Run the completeObjectName command to see if an application is running.

$	is a Jacl operator for substituting a variable name with its value

Table 399. completeObjectName command elements (continued). Run the completeObjectName command to see if an application is running.

`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere Application Server process
`completeObjectName`	is an AdminControl command
`type=Application,name=`*`myApplication`*	is the hierarchical containment path of the configuration object
`print`	is a Jython command

Results

If *myApplication* is running, then an MBean is created. Otherwise, the command returns nothing. If *myApplication* is running, the output would resemble the following:

```
WebSphere:cell=mycell,name=myApplication,mbeanIdentifier=cells/mycell/applications/myApplication.ear/
deployments/myApplication/deployment.xml#ApplicationDeployment_1,type=Application,node=mynode,Server=
dmgr,process=dmgr,J2EEName=myApplication
```

Disabling application loading in deployed targets using wsadmin scripting

You can use the AdminConfig object and scripting to disable application loading in deployed targets.

About this task

The following example uses the AdminConfig object to disable application loading in deployed targets:

Procedure

1. Start the wsadmin scripting tool.
2. Obtain the Deployment object for the application and assign it to the deployments variable, for example:
   - Using Jacl:

     ```
     set deployments [$AdminConfig getid /Deployment:myApp/]
     ```
   - Using Jython:

     ```
     deployments = AdminConfig.getid("/Deployment:myApp/")
     ```

Table 400. getid Deployment command elements. Run the getid command to get a deployment object.

`set`	is a Jacl command
`deployments`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is an attribute
`myApp`	is the value of the application name **Note:** If you are using the Jython scripting language, make sure that you read the notes concerning a space character within application names.

   Example output:

   ```
   myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
   ```
3. Obtain the target mappings in the application and assign them to the targetMappings variable, for example:

- Using Jacl:

```
set deploymentObj1 [$AdminConfig showAttribute $deployments deployedObject]
set targetMap1 [lindex [$AdminConfig showAttribute $deploymentObj1 targetMappings] 0]
```

  Example output:

```
(cells/mycell/applications/ivtApp.ear/deployments/ivtApp|deployment.xml#DeploymentTargetMapping_1)
```

- Using Jython:

```
deploymentObj1 = AdminConfig.showAttribute(deployments, 'deployedObject')
targetMap1 = AdminConfig.showAttribute(deploymentObj1, 'targetMappings')
targetMap1 = targetMap1[1:len(targetMap1)-1].split(" ")
print targetMap1
```

  Note: When you attempt to obtain the target mappings in the application through scripting and then assigning those values to the *targetMappings* variable, be aware when the application has a space in the name or blank. In these cases, you must compensate for the occurrence of a blank or space character as the Jython example demonstrates. Errors can occur if you do not make this adjustment. Consider the following scenarios:

  – If only one single *DeploymentTargetMapping* value exists within the `deployment.xml` file, you can either split the *targetMappings* value with a space or a `line.separator` entry. The `line.separator` entry syntax works when the application name contains a space, such as "IVT Application". For example:

```
targetMap1 = targetMap1[1:len(targetMap1)-1].split(" ")
```

    or

```
targetMap1 =
targetMap1[1:len(targetMap1)-1].split(java.lang.System.getProperty("line.separator"))
```

  – If multiple *DeploymentTargetMapping* values exist within the `deployment.xml` file, split the *targetMappings* values with a space. However, you must also use "\"" and " ", as appropriate, when the application name or deployment target string contains a space character. For example:

```
targetMap1 = targetMap1[1:len(targetMap1)-1].split(" ")
```

  Example output:

```
['(cells/mycell/applications/ivtApp.ear/deployments/ivtApp|deployment.xml#DeploymentTargetMapping_1)']
```

Table 401. showAttribute command elements. Run the showAttribute command to assign target mappings.

`set`	is a Jacl command
`deploymentObj1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates the ID of the Deployment object that is specified in step number 1
`deployedObject`	is an attribute
`targetMap1`	is a variable name
`targetMappings`	is an attribute
`lindex`	is a Jacl command
`print`	is a Jython command

4. Disable the loading of the application on each deployed target, for example:
   - Using Jacl:

```
foreach tm $targetMap1 {
      $AdminConfig modify $tm {{enable false}}
}
```

- Using Jython:

```
for targetMapping in targetMap1:
   AdminConfig.modify(targetMapping, [["enable", "false"]])
```

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Exporting applications using wsadmin scripting

You can export your applications before you update installed applications or before you migrate to a different version of the WebSphere Application Server product.

Before you begin

The application whose contents you want to export is installed on a server.

About this task

Exporting applications enables you to back them up and preserve their binding information.

Procedure

1. Start the wsadmin scripting tool.
2. Export applications.
   - Export an enterprise application to a location of your choice, for example:
     - Using Jacl:
     - Using Jython:

Table 402. export command elements. Run the export command to export an application to a file.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects management
`export`	is an AdminApp command
app1	is the name of the application that will be exported
`/mystuff/exported.ear`	is the name of the file where the exported application will be stored

   - Export Data Definition Language (DDL) files in the enterprise bean module of an application to a destination directory, for example:
     - Using Jacl:
     - Using Jython:

Table 403. exportDDL command elements. Run the exportDDL command to export DDL files.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects management
`exportDDL`	is an AdminApp command
app1	is the name of the application whose DDL files will be exported
/mystuff	is the name of the directory where the DDL files export from the application

Chapter 14. Configuring applications using scripting

Using the wsadmin tool, you can run scripts to configure applications.

Configuring applications for session management using scripting

This task provides an example that uses the AdminConfig object to configure a session manager for the application.

Before you begin

An application must be installed on a running server.

About this task

You can use the AdminConfig object to set configurations in an application. Some configuration settings are not available through the AdminConfig object.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the deployment configuration object for the application and assign it to the deployment variable.

   Note: This step is not needed for an OSGi application. See Adding an EBA asset to a composition unit using wsadmin commands and Modifying the configuration of an EBA composition unit using wsadmin commands.

   For example:

   - Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:myApp/]
```

   - Using Jython:

```
deployments = AdminConfig.getid('/Deployment:myApp/')
print deployments
```

   where:

Table 404. getid command elements. Run the getid command to identify a deployment object.

`set`	is a Jacl command
`deployments`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is an attribute
myApp	is the value of the attribute

   Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

3. Retrieve the application deployment object and assign it to the appDeploy variable. For example:

   - Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployments deployedObject]
```

   - Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployments, 'deployedObject')
print appDeploy
```

Note: For an OSGi application, use the following jython code for this step:

```
appDeploy = AdminTask.getOSGiApplicationDeployedObject('-cuName cu_name')
```

where:

Table 405. set command elements. Run the set command to assign the deployment object a value.

`set`	is a Jacl command
`appDeploy`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates the ID of the deployment object that is specified in step number 1
`deployedObject`	is an attribute
cu_name	is the name of the composition unit

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#ApplicationDeployment_1)
```

4. To obtain a list of attributes that you can set for a session manager, use the **attributes** command. For example:
   - Using Jacl:

```
$AdminConfig attributes SessionManager
```

   - Using Jython:

```
print AdminConfig.attributes('SessionManager')
```

where:

Table 406. attributes command elements. Run the attributes command to list attributes of a session manager.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`attributes`	is an AdminConfig command
`SessionManager`	is an attribute

Example output:

```
"accessSessionOnTimeout Boolean"
"allowSerializedSessionAccess Boolean"
"context ServiceContext@"
"defaultCookieSettings Cookie"
"enable Boolean"
"enableCookies Boolean"
"enableProtocolSwitchRewriting Boolean"
"enableSSLTracking Boolean"
"enableSecurityIntegration Boolean"
"enableUrlRewriting Boolean"
"maxWaitTime Integer"
"properties Property(TypedProperty)*"
"sessionDRSPersistence DRSSettings"
"sessionDatabasePersistence SessionDatabasePersistence"
"sessionPersistenceMode ENUM(DATABASE, DATA_REPLICATION, NONE)"
"tuningParams TuningParams"
```

When you configure an application for session management, it is recommended that you specify each attribute.

gotcha: If you are setting up the session management attributes for a cluster, you must also update the targetMappings element of the AdminConfig object for the cluster before the settings you

specify for the sessionManagment element become effective. If you do not update the targetMappings element, the settings are not effective even though they appear in the deployment.xml file.

5. Set up the attributes for the session manager.

   The following example sets four top-level attributes in the session manager. You can modify the example to set other attributes of the session manager, including the nested attributes in DRSSettings, SessionDataPersistence, and TuningParms object types.

   gotcha: The session manager requires that you set both the defaultCookieSettings and tuningParams attributes before you initialize an application. If you do not set these attributes, the session manager cannot initialize the application, and the application does not start.

   To list the attributes for those object types, use the **attributes** command of the AdminConfig object.

   - Using Jacl:

```
set attr1 [list enableSecurityIntegration true]
set attr2 [list maxWaitTime 30]
set attr3 [list sessionPersistenceMode NONE]
set kuki [list maximumAge -1]
set cookie [list $kuki]
Set cookieSettings [list defaultCookieSettings $cookie]
set attrs [list $attr1 $attr2 $attr3 $cookieSettings]
set sessionMgr [list sessionManagement $attrs]
```

     Example output using Jacl:

```
sessionManagement {{enableSecurityIntegration true} {maxWaitTime 30} {sessionPersistenceMode NONE}
{defaultCookieSettings {{maximumAge -1}}}}
```

   - Using Jython:

```
attr1 = ['enableSecurityIntegration', 'true']
attr2 = ['maxWaitTime', 30]
attr3 = ['sessionPersistenceMode', 'NONE']
kuki = ['maximumAge', -1]
cookie = [kuki]
cookieSettings = ['defaultCookieSettings', cookie]
attrs = [attr1, attr2, attr3, cookieSettings]
sessionMgr = [['sessionManagement', attrs]]
```

     Example output using Jython:

```
[[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30], [sessionPersistenceMode, NONE],
[defaultCookieSettings [[maximumAge, -1]]]]
```

   where:

Table 407. set command elements. Run the set command to set attributes for a session manager.

set	is a Jacl command
attr1, attr2, attr3, attrs, sessionMgr	are variable names
$	is a Jacl operator for substituting a variable name with its value
enableSecurityIntegration	is an attribute
true	is a value of the enableSecurityIntegration attribute
maxWaitTime	is an attribute
30	is a value of the maxWaitTime attribute
sessionPersistenceMode	is an attribute
NONE	is a value of the sessionPersistenceMode attribute

6. Perform one of the following:
   - Create the session manager for the application. For example:
     - Using Jacl:

```
$AdminConfig create ApplicationConfig $appDeploy [list $sessionMgr]
```

     - Using Jython:

```
print AdminConfig.create('ApplicationConfig', appDeploy, sessionMgr)
```

where:

Table 408. create command elements. Run the create command to create a session manager.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
create	is an AdminConfig command
ApplicationConfig	is an attribute
appDeploy	evaluates the ID of the deployed application that is specified in step number 2
list	is a Jacl command
sessionMgr	evaluates the ID of the session manager that is specified in step number 4

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#ApplicationConfig_1)
```

- If a session manager already exists, use the **modify** command of the AdminConfig object to update the configuration of the session manager. For example:
  - Using Jacl:

```
set configs [lindex [$AdminConfig showAttribute $appDeploy configs] 0]
set appConfig [lindex $configs 0]
set SM [$AdminConfig showAttribute $appConfig sessionManagement]
$AdminConfig modify $SM $attrs
```

  - Using Jython:

```
configs = AdminConfig.showAttribute (appDeploy, 'configs')
appConfig = configs[1:len(configs)-1]
SM = AdminConfig.showAttribute (appConfig, 'sessionManagement')
AdminConfig.modify (SM, attrs)
```

7. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Configuring applications for session management in web modules using scripting

Use scripting and the wsadmin tool to configure applications for session management in web modules.

About this task

You can use the AdminApp object to set configurations in an application. Some configuration settings are not available through the AdminApp object. The following task uses the AdminConfig object to configure a session manager for a web module in the application.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the deployment configuration object for the application and assign it to the deployment variable. For example:
   - Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:myApp/]
```

   - Using Jython:

```
deployments = AdminConfig.getid('/Deployment:myApp/')
print deployments
```

where:

Table 409. Deployment configuration values. The following table describes the elements of the getid command.

`set`	is a Jacl command
`deployments`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is an attribute
myApp	is the value of the attribute

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

3. Get all the modules in the application and assign them to the modules variable. For example:
   - Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployments deployedObject]
set mod1 [$AdminConfig showAttribute $appDeploy modules]
set mod1 [lindex $mod1 0]
```

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp:deployment.xml#WebModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp:deployment.xml#EJBModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp:deployment.xml#WebModuleDeployment_2)
```

   - Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployments, 'deployedObject')
mod1 = AdminConfig.showAttribute(appDeploy, 'modules')
print mod1
```

Example output:

```
[(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#WebModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#EJBModuleDeployment_1)
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#EJBModuleDeployment_2)]
```

where:

Table 410. Application module values. The following table describes the elements of the showAttribute AdminConfig command.

`set`	is a Jacl command
`appDeploy`	is a variable name
`mod1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`showAttribute`	is an AdminConfig command
`deployments`	evaluates the ID of the deployment object that is specified in step number 1
`deployedObject`	is an attribute

4. To obtain a list of attributes that you can set for a session manager, use the **attributes** command. For example:
   - Using Jacl:

```
$AdminConfig attributes SessionManager
```

- Using Jython:

```
print AdminConfig.attributes('SessionManager')
```

where:

Table 411. attributes AdminConfig command. The following table describes the elements for the attributes AdminConfig command.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
attributes	is an AdminConfig command
SessionManager	is an attribute

Example output:

```
"accessSessionOnTimeout Boolean"
"allowSerializedSessionAccess Boolean"
"context ServiceContext@"
"defaultCookieSettings Cookie"
"enable Boolean"
"enableCookies Boolean"
"enableProtocolSwitchRewriting Boolean"
"enableSSLTracking Boolean"
"enableSecurityIntegration Boolean"
"enableUrlRewriting Boolean"
"maxWaitTime Integer"
"properties Property(TypedProperty)*"
"sessionDRSPersistence DRSSettings"
"sessionDatabasePersistence SessionDatabasePersistence"
"sessionPersistenceMode ENUM(DATABASE, DATA_REPLICATION, NONE)"
"tuningParams TuningParams"
```

5. Set up the attributes for session manager. The following example sets four top-level attributes in the session manager.

   You can modify the example to set other attributes in the session manager, including the nested attributes in Cookie, DRSSettings, SessionDataPersistence, and TuningParms object types. To list the attributes for those object types, use the **attributes** command of AdminConfig object.

   gotcha: The session manager requires that you set both the defaultCookieSettings and tuningParams attributes before you initialize an application. If you do not set these attributes, the session manager cannot initialize the application, and the application does not start.

   - Using Jacl:

```
set attr0 [list enable true]
set attr1 [list enableSecurityIntegration true]
set attr2 [list maxWaitTime 30]
set attr3 [list sessionPersistenceMode NONE]
set attr4 [list enableCookies true]
set attr5 [list invalidationTimeout 45]
set tuningParmsDetailList [list $attr5]
set tuningParamsList [list tuningParams $tuningParmsDetailList]
set pwdList [list password 95ee608]
set userList [list userId Administrator]
set dsNameList [list datasourceJNDIName jdbc/session]
set dbPersistenceList [list $dsNameList $userList $pwdList]
set sessionDBPersistenceList [list $dbPersistenceList]
set sessionDBPersistenceList [list sessionDatabasePersistence $dbPersistenceList]
set kuki [list maximumAge 1000]
set cookie [list $kuki]
set cookieSettings [list defaultCookieSettings $cookie]
set sessionManagerDetailList [list $attr0 $attr1 $attr2 $attr3 $attr4 $cookieSettings
$tuningParamsList $sessionDBPersistenceList]
set sessionMgr [list sessionManagement $sessionManagerDetailList]
set id [$AdminConfig create ApplicationConfig $appDeploy [list $sessionMgr] configs]
set targetMappings [lindex [$AdminConfig showAttribute $appDeploy targetMappings] 0]
set attrs [list config $id]
$AdminConfig modify $targetMappings [list $attrs]
```

Note: The last five lines in the sample above assume that you deployed the web module to only one target server. You can target a module to multiple servers or clusters, by using a loop, if you want to apply the update to each target. Replace the last six lines of the sample above with the following code to apply updates to multiple targets:

```
set id [$AdminConfig create ApplicationConfig $appDeploy [list $sessionMgr] configs]

set targetMappings [lindex [$AdminConfig showAttribute $appDeploy targetMappings] 0]

foreach target $targetMappings {
    if {[regexp DeploymentTargetMapping $target] == 1} {
        set attrs [list config $id]
        $AdminConfig modify $target [list $attrs]
    }
}
```

Example output using Jacl:

```
sessionManagement {{enableSecurityIntegration true} {maxWaitTime 30}
{sessionPersistenceMode NONE} {enabled true}}
```

- Using Jython:

```
attr0 = ['enable', 'true']
attr1 = ['enableSecurityIntegration', 'true']
attr2 = ['maxWaitTime', 30]
attr3 = ['sessionPersistenceMode', 'NONE']
attr4 = ['enableCookies', 'true']
attr5 = ['invalidationTimeout', 45]
tuningParmsDetailList = [attr5]
tuningParamsList = ['tuningParams', tuningParmsDetailList]
pwdList = ['password', '95ee608']
userList = ['userId', 'Administrator']
dsNameList = ['datasourceJNDIName', 'jdbc/session']
dbPersistenceList = [dsNameList, userList, pwdList]
sessionDBPersistenceList = [dbPersistenceList]
sessionDBPersistenceList = ['sessionDatabasePersistence', dbPersistenceList]
kuki = ['maximumAge', 1000]
cookie = [kuki]
cookieSettings = ['defaultCookieSettings', cookie]
sessionManagerDetailList = [attr0, attr1, attr2, attr3, attr4, cookieSettings,
tuningParamsList, sessionDBPersistenceList]
sessionMgr = ['sessionManagement', sessionManagerDetailList]
id = AdminConfig.create('ApplicationConfig', appDeploy,[sessionMgr], 'configs')
targetMappings = AdminConfig.showAttribute(appDeploy, 'targetMappings')
targetMappings = targetMappings[1:len(targetMappings)-1]
print targetMappings
attrs = ['config', id]
AdminConfig.modify(targetMappings,[attrs])
```

Note: The last six lines in the sample above assume that you deployed the web module to only one target server. You can target a module to multiple servers or clusters, by using a loop, if you want to apply the update to each target. Replace the last six lines of the sample above with the following code to apply updates to multiple targets:

```
id = AdminConfig.create('ApplicationConfig', appDeploy,[sessionMgr], 'configs')

targetMappings = AdminConfig.showAttribute(appDeploy, 'targetMappings')
targetMappings = targetMappings[1:len(targetMappings)-1].split(" ")
for target in targetMappings:
 if target.find('DeploymentTargetMapping') != -1:
  attrs = ['config', id]
  AdminConfig.modify(target,[attrs])
 #endif
#endfor
```

Example output using Jython:

```
[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30], [sessionPersistenceMode, NONE]]
```

6. Set up the attributes for the web module. For example:
   - Using Jacl:

```
set nameAttr [list name myWebModuleConfig]
set descAttr [list description "Web Module config post create"]
set webAttrs [list $nameAttr $descAttr $sessionMgr]
```

Example output:

```
{name myWebModuleConfig} {description {Web Module config post create}}
{sessionManagement {{enableSecurityIntegration true} {maxWaitTime 30}
{sessionPersistenceMode NONE} {enabled true}}}
```

- Using Jython:

```
nameAttr = ['name', 'myWebModuleConfig']
descAttr = ['description', "Web Module config post create"]
webAttrs = [nameAttr, descAttr, sessionMgr]
```

Example output:

```
[[name, myWebModuleConfig], [description, "Web Module config post create"],
[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30],
[sessionPersistenceMode, NONE], [enabled, true]]]]
```

where:

Table 412. Setting the attributes for the web module. The following table describes the elements for setting the attributes of the web module.

set	is a Jacl command
nameAttr, descAttr, webAttrs	are variable names
$	is a Jacl operator for substituting a variable name with its value
name	is an attribute
myWebModuleConfig	is a value of the name attribute
description	is an attribute
Web Module config post create	is a value of the description attribute

7. Create the session manager for each web module in the application. You can modify the following example to set other attributes of the session manager in a web module configuration. You must also define a target mapping for this step.

   - Using Jacl:

```
foreach module $mod1 {
 if {[regexp WebModuleDeployment $module] == 1} {
  set moduleConfig [$AdminConfig create WebModuleConfig $module $webAttrs]
  set targetMappings [lindex [$AdminConfig showAttribute $module targetMappings] 0]
  set attrs [list config $moduleConfig]
  $AdminConfig modify $targetMappings [list $attrs]
 }
}
```

   Note: You can add an optional, additional loop to assign new web module configuration to each target, if the web module is deployed to more than one target server:

```
foreach module $mod1 {
    if {[regexp WebModuleDeployment $module] == 1} {
        set moduleConfig [$AdminConfig create WebModuleConfig $module $webAttrs]
        set targetMappings [lindex [$AdminConfig showAttribute $module targetMappings] 0]
        foreach target $targetMappings {
        if {[regexp DeploymentTargetMapping $target] == 1} {
            set attrs [list config $moduleConfig]
            $AdminConfig modify $target [list $attrs]
        }
    }
}
```

   - Using Jython:

```
arrayModules = mod1[1:len(mod1)-1].split(" ")
for module in arrayModules:
 if module.find('WebModuleDeployment') != -1:
```

```
AdminConfig.create('WebModuleConfig', module, webAttrs)
targetMappings = targetMappings[1:len(targetMappings)-1]
attrs = ['config', moduleConfig]
AdminConfig.modify (targetMappings, [attrs])
```

Note: You can add an optional, additional loop to assign new web module configuration to each target, if the web module is deployed to more than one target server:

```
arrayModules = mod1[1:len(mod1)-1].split(" ")
for module in arrayModules:
  if module.find('WebModuleDeployment') != -1:
    moduleConfig = AdminConfig.create('WebModuleConfig', module, webAttrs)
    attrs = ['config', moduleConfig]
    targetMappings = AdminConfig.showAttribute(appDeploy, 'targetMappings')
    targetMappings = targetMappings[1:len(targetMappings)-1].split(" ")
    for target in targetMappings:
        if target.find('DeploymentTargetMapping') != -1:
            attrs = ['config', moduleConfig]
            AdminConfig.modify(target,[attrs])
```

Example output:

```
myWebModuleConfig(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#WebModuleConfiguration_1)
```

If you do not specify the tuningParamsList attribute when you create the session manager, you will receive an error when you start the deployed application.

8. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Configuring a shared library using scripting

You can use scripting to configure a shared library for application servers. Shared libraries are files used by multiple applications. Create a shared library to reduce the number of duplicate library files on your system.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the server and assign it to the server variable. For example:
   - Using Jacl:

```
set serv [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
serv = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print serv
```

Table 413. getid command elements. The following table describes each element of the getid command.

set	is a Jacl command
serv	is a variable name
$	is a Jacl operator for substituting a variable name with its value

Table 413. getid command elements (continued). The following table describes each element of the getid command.

`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Cell`	is an attribute
`mycell`	is the value of the attribute
`Node`	is an attribute
`mynode`	is the value of the attribute
`Server`	is an attribute
`server1`	is the value of the attribute

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. Create the shared library in the server. For example:
   - Using Jacl:

```
$AdminConfig create Library $serv {{name mySharedLibrary} {classPath
/home/myProfile/mySharedLibraryClasspath}}
```

   - Using Jython:

```
print AdminConfig.create('Library', serv, [['name', 'mySharedLibrary'], ['classPath',
'home/myProfile/mySharedLibraryClasspath']])
```

Table 414. create Library command elements. The following table describes each element of the create Library command.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Library`	is an attribute
`serv`	evaluates the ID of the server that is specified in step number 1
`name`	is an attribute
`mySharedLibrary`	is a value of the name attribute
`classPath`	is an attribute
/mySharedLibraryClasspath	is the value of the classpath attribute
`print`	is a Jython command

Example output:

```
MysharedLibrary(cells/mycell/nodes/mynode/servers/server1|libraries.xml#Library_1)
```

4. Identify the application server from the server and assign it to the appServer variable. For example:
   - Using Jacl:

```
set appServer [$AdminConfig list ApplicationServer $serv]
```

   - Using Jython:

```
appServer = AdminConfig.list('ApplicationServer', serv)
print appServer
```

Table 415. list command elements. The following table describes each element of the create list command.

`set`	is a Jacl command
`appServer`	is a variable name

Table 415. list command elements (continued). The following table describes each element of the create list command.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
list	is an AdminConfig command
ApplicationServer	is an attribute
serv	evaluates the ID of the server that is specified in step number 1
print	is a Jython command

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#ApplicationServer_1
```

5. Identify the class loader in the application server and assign it to the classLoader variable. For example:
   - To use the existing class loader that is associated with the server, the following commands use the first class loader:
     - Using Jacl:

```
set classLoad [$AdminConfig showAttribute $appServer classloaders]
set classLoader1 [lindex $classLoad 0]
```

     - Using Jython:

```
classLoad = AdminConfig.showAttribute(appServer, 'classloaders')
cleanClassLoaders = classLoad[1:len(classLoad)-1]
classLoader1 = cleanClassLoaders.split(' ')[0]
```

Table 416. showAttribute command elements. The following table describes each element of the showAttribute command.

set	is a Jacl command
classLoad, classLoader1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
appServer	evaluates the ID of the application server that is specified in step number 3
classloaders	is an attribute
print	is a Jython command

   - To create a new class loader, issue the following command:
     - Using Jacl:

```
set classLoader1 [$AdminConfig create Classloader $appServer {{mode PARENT_FIRST}}]
```

     - Using Jython:

```
classLoader1 = AdminConfig.create('Classloader', appServer, [['mode',  'PARENT_FIRST']])
```

Table 417. create Classloader command elements. The following table describes each element of the create Classloader command.

set	is a Jacl command
classLoader1	is a variable name
$	is a Jacl operator for substituting a variable name with its value

Table 417. create Classloader command elements (continued). The following table describes each element of the create Classloader command.

`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Classloader`	is an attribute
`appServer`	evaluates the ID of the application server that is specified in step number 3
`mode`	is an attribute
`PARENT_FIRST`	is the value of the attribute
`print`	is a Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#Classloader_1)
```

6. Associate the shared library that you created with the application server through the class loader. For example:
   - Using Jacl:

```
$AdminConfig create LibraryRef $classLoader1 {{libraryName MyshareLibrary}}
```

   - Using Jython:

```
print AdminConfig.create('LibraryRef', classLoader1, [['libraryName', 'MyshareLibrary']])
```

Table 418. create LibraryRef command elements. The following table describes each element of the create LibraryRef command.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`create`	is an AdminConfig command
`LibraryRef`	is an attribute
`classLoader1`	evaluates the ID of the class loader that is specified in step number 4
`libraryName`	is an attribute
`MyshareLibrary`	is the value of the attribute
`print`	is a Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#LibraryRef_1)
```

7. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Configuring a shared library for an application using wsadmin scripting

This task uses the AdminConfig object to configure a shared library for an application. Shared libraries are files used by multiple applications. Create a shared library to reduce the number of duplicate library files on your system.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the shared library and assign it to the library variable. You can either use an existing shared library or create a new one, for example:
   - To create a new shared library, perform the following steps:
     a. Identify the node and assign it to a variable, for example:
        – Using Jacl:

```
set n1 [$AdminConfig getid /Cell:mycell/Node:mynode/]
```

        – Using Jython:

```
n1 = AdminConfig.getid('/Cell:mycell/Node:mynode/')
print n1
```

Table 419. getid command elements. Run the getid command to identify a shared library.

`set`	is a Jacl command
`n1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Cell`	is the object type
`mycell`	is the name of the object that will be modified
`Node`	is the object type
`mynode`	is the name of the object that will be modified

Example output:

```
mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
```

     b. Create the shared library in the node. The following example creates a new shared library in the node scope. You can modify it to use the cell or server scope.
        – Using Jacl:
        – Using Jython:

Table 420. create Library command elements. Run the create command to create a shared library.

`set`	is a Jacl command
`library`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`create`	is an AdminConfig command
`Library`	is an AdminConfig object
`n1`	evaluates to the ID of host node specified in step number 1
`name`	is an attribute
`mySharedLibrary`	is the value of the name attribute

Table 420. create Library command elements (continued). Run the create command to create a shared library.

classPath	is an attribute
/mySharedLibraryClasspath	is the value of the classPath attribute

Example output:

```
MySharedLibrary(cells/mycell/nodes/mynode|libraries.xml#Library_1)
```

- To use an existing shared library, issue the following command:
  - Using Jacl:

```
set library [$AdminConfig getid /Library:mySharedLibrary/]
```

  - Using Jython:

```
library = AdminConfig.getid('/Library:mySharedLibrary/')
print library
```

Table 421. getid Library command elements. Run the getid command to identify a shared library.

set	is a Jacl command
library	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
getid	is an AdminConfig command
Library	is an attribute
mySharedLibrary	is the value of the Library attribute

Example output:

```
MySharedLibrary(cells/mycell/nodes/mynode|libraries.xml#Library_1)
```

3. Identify the deployment configuration object for the application and assign it to the deployment variable. For example:
   - Using Jacl:

```
set deployment [$AdminConfig getid /Deployment:myApp/]
```

   - Using Jython:

```
deployment = AdminConfig.getid('/Deployment:myApp/')
print deployment
```

Table 422. getid Deployment command elements. Run the getid command to identify a deployment object.

set	is a Jacl command
deployment	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
getid	is an AdminConfig command
Deployment	is an attribute
myApp	is the value of the Deployment attribute
print	is a Jython command

Example output:

```
myApp(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Deployment_1)
```

4. Retrieve the application deployment and assign it to the appDeploy variable. For example:
   - Using Jacl:

```
set appDeploy [$AdminConfig showAttribute $deployment deployedObject]
```

   - Using Jython:

```
appDeploy = AdminConfig.showAttribute(deployment, 'deployedObject')
print appDeploy
```

Table 423. showAttribute deployment command elements. Run the showAttribute command to assign a deployed object.

set	is a Jacl command
appDeploy	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
deployment	evaluates the ID of the deployment configuration object specified in step number 2
deployedObject	is an attribute of modify objects
print	is a Jython command

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/
myApp|deployment.xml#ApplicationDeployment_1)
```

5. Identify the class loader in the application deployment and assign it to the classLoader variable. For example:
   - Using Jacl:

```
set classLoad1 [$AdminConfig showAttribute $appDeploy classloader]
```

   - Using Jython:

```
classLoad1 = AdminConfig.showAttribute(appDeploy, 'classloader')
print classLoad1
```

Table 424. showAttribute appDeploy command elements. Run the showAttribute command to assign a class loader.

set	is a Jacl command
classLoad1	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
appDeploy	evaluates the ID of the application deployment specified in step number 3
classLoader	is an attribute of modify objects
print	is a Jython command

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#Classloader_1)
```

6. Associate the shared library in the application through the class loader. For example:
   - Using Jacl:

```
$AdminConfig create LibraryRef $classLoad1 {{libraryName
MyshareLibrary}}
```

   - Using Jython:

```
print AdminConfig.create('LibraryRef', classLoad1, [['libraryName',
'MyshareLibrary']])
```

Table 425. create LibraryRef command elements. Run the create command to create a library reference.

$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object representing the WebSphere Application Server configuration
create	is an AdminConfig command
LibraryRef	is an AdminConfig object
classLoad1	evaluates to the ID of class loader specified in step number 4

Table 425. create LibraryRef command elements (continued). Run the create command to create a library reference.

`libraryName`	is an attribute
`MyshareLibrary`	is the value of the libraryName attribute

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#LibraryRef_1)
```

7. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Setting background applications using wsadmin scripting

You can enable or disable a background application using scripting and the wsadmin tool.

About this task

Background applications specify whether the application must initialize fully before the server starts. The default setting is `false` and this indicates that server startup will not complete until the application starts. If you set the value to `true`, the application starts on a background thread and server startup continues without waiting for the application to start. The application may not ready for use when the application server starts.

Procedure

1. Start the wsadmin scripting tool.
2. Locate the application deployment object for the application. For example:
   - Using Jacl:

     ```
     set applicationDeployment [$AdminConfig getid /Deployment:adminconsole/ApplicationDeployment:/]
     ```

   - Using Jython:

     ```
     applicationDeployment = AdminConfig.getid('/Deployment:adminconsole/ApplicationDeployment:/')
     ```

Table 426. getid command elements. Run the getid command to get an application object.

`set`	is a Jacl command
`applicationDeployment`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Deployment`	is a type
`ApplicationDeployment`	is a type
`adminconsole`	is the name of the application

3. Enable the background application. For example:
   - Using Jacl:

     ```
     $AdminConfig modify $applicationDeployment "{backgroundApplication true}"
     ```

   - Using Jython:

     ```
     AdminConfig.modify(applicationDeployment, ['backgroundApplication', 'true'])
     ```

Table 427. modify command elements. Run the modify command to set the backgroundApplication value.

`$`	is a Jacl operator for substituting a variable name with its value

Table 427. modify command elements (continued). Run the modify command to set the backgroundApplication value.

`AdminConfig`	is an object that represents the WebSphere Application Server configuration
`modify`	is an AdminConfig command
`applicationDeployment`	is a variable name that was set in step 1
`backgroundApplication`	is an attribute
`true`	is the value of the backgroundApplication attribute

4. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Modifying WAR class loader policies for applications using wsadmin scripting

You can use scripting and the wsadmin tool to modify WAR class loader policies for applications.

Before you begin

About this task

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

To modify WAR class loader policies for an application, perform the following steps:

Procedure

1. Start the wsadmin scripting tool.
2. Retrieve the configuration ID of the object that you want to modify and set it to the dep variable. For example:
   - Using Jacl:

     ```
     set dep [$AdminConfig getid /Deployment:MyApp/]
     ```
   - Using Jython:

     ```
     dep = AdminConfig.getid("/Deployment:MyApp/")
     ```
3. Identify the deployed object and set it to the deployedObject variable. For example:
   - Using Jacl:

     ```
     set deployedObject [$AdminConfig showAttribute $dep deployedObject]
     ```
   - Using Jython:

     ```
     deployedObject = AdminConfig.showAttribute(dep, "deployedObject")
     ```
4. Show the current attribute values of the configuration object with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $deployedObject warClassLoaderPolicy
     ```

     Example output:

     ```
     {warClassLoaderPolicy MULTIPLE}
     ```
   - Using Jython:

     ```
     AdminConfig.show(deployedObject, 'warClassLoaderPolicy')
     ```

Example output:

```
'[warClassLoaderPolicy MULTIPLE]'
```

5. Modify the attributes of the configuration object with the **modify** command, for example:
   - Using Jacl:

     ```
     $AdminConfig modify $deployedObject {{warClassLoaderPolicy SINGLE}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(deployedObject, [['warClassLoaderPolicy', 'SINGLE']])
     ```

6. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

7. Verify the changes that you made to the attribute value with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $deployedObject warClassLoaderPolicy
     ```

     Example output:

     ```
     {warClassLoaderPolicy SINGLE}
     ```

   - Using Jython:

     ```
     AdminConfig.show(deployedObject, 'warClassLoaderPolicy')
     ```

     Example output:

     ```
     '[warClassLoaderPolicy SINGLE]'
     ```

Modifying WAR class loader mode using wsadmin scripting

You can use scripting and the wsadmin tool to modify WAR class loader mode for applications.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 topic for more information.

About this task

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

To modify WAR class loader mode for an application, complete the following steps:

Procedure

1. Set a reference to the `deployment.xml` document. For example:
   - Using Jacl:

     ```
     set deployments [$AdminConfig getid /Deployment:my_application/]
     ```

     Example output:

     ```
     application_name(cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#Deployment_1276887608391)
     ```

   - Using Jython:

     ```
     deployments = AdminConfig.getid('/Deployment:my_application/')
     ```

Table 428. Deployment configuration values. The following table describes the elements of the getid command.

set	is a Jacl command

Table 428. Deployment configuration values (continued). The following table describes the elements of the getid command.

deployments	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
getid	is an AdminConfig command
Deployment	is an attribute
my_application	is an application in the *profile_root*/config/cells/*cell_name*/applications/ directory

2. Set a reference to the deployedObject attribute within the deployment.xml document and set it to the *deployedObject* variable. For example:
   - Using Jacl:

     ```
     set deploymentObject [$AdminConfig showAttribute $deployments deployedObject]
     ```

     Example output:

     ```
     (cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#ApplicationDeployment_1276887608391)
     ```
   - Using Jython:

     ```
     deploymentObject = AdminConfig.showAttribute(deployments, 'deployedObject')
     ```

Table 429. Deployment configuration values. The following table describes the elements in this command.

set	is a Jacl command
deploymentObject	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
deployments	is a variable to which the deployment.xml document is assigned
deployedObject	is an attribute within the deployment.xml document

3. List the modules for the deployedObject attribute and set the list to the *myModules* variable. For example:
   - Using Jacl:

     ```
     set myModules [lindex [$AdminConfig showAttribute $deploymentObject modules] 0]
     ```

     Example output:

     ```
     (cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#WebModuleDeployment_1276887608391)
     (cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#EJBModuleDeployment_1276887608391)
     ```
   - Using Jython:

     ```
     myModules = AdminConfig.showAttribute(deploymentObject, 'modules')
     myModules = myModules[1:len(myModules)-1].split(" ")
     print myModules
     ```

     Example output:

     ```
     ['(cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#WebModuleDeployment_1276887608391)',
     '(cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#EJBModuleDeployment_1276887608391)']
     ```

Table 430. Deployment configuration values. The following table describes the elements in this command.

set	is a Jacl command
myModules	is a variable name
$	is a Jacl operator for substituting a variable name with its value
AdminConfig	is an object that represents the WebSphere Application Server configuration
showAttribute	is an AdminConfig command
deployedObject	is an attribute within the deployment.xml document
modules	is an attribute within the deployment.xml document

4. Find the web module and set the mode for the class loader. For example:
   - Using Jacl:

     ```
     foreach module $myModules {
          if {[regexp WebModuleDeployment $module] == 1} {
             $AdminConfig modify $module {{classloaderMode mode}}}}
     ```

   - Using Jython

     ```
     for module in myModules:
          if (module.find('WebModuleDeployment')!= -1):
             AdminConfig.modify(module, [['classloaderMode', 'mode']])
     ```

Table 431. Deployment configuration values. The following table describes the elements in this command.

`foreach`	is a Jacl command
`for`	is a Jython command
`module`	is an object that is being modified
`$`	is a Jacl operator for substituting a variable name with its value
`myModules`	is a variable name
`regexp`	is a function to use regular expression is used for searching within the previous commands
`module.find`	is a function to use regular expression is used for searching within the previous commands
`AdminConfig`	is an object that represents the WebSphere Application Server configuration
modify	is an AdminConfig command
`classloaderMode`	is an attribute within the `deployment.xml` document
`mode`	is the class loader mode value that you want to set for the WAR module. The mode value is either PARENT_FIRST or PARENT_LAST. For more information, see the documentation about class loaders.

5. Save the configuration, for example:
   - Using Jacl:

     ```
     $AdminConfig save
     ```

   - Using Jython:

     ```
     AdminConfig.save()
     ```

6. Verify the changes that you made to the attribute value with the showall command. For example:
   - Using Jacl:

     ```
     $AdminConfig showall $module
     ```

   - Using Jython:

     ```
     AdminConfig.showall(module)
     ```

     Example output:

     ```
     {applicationDeployment (cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#ApplicationDeployment_1276887608391)}

     {classloader (cells/cell_name/applications/application_name.ear/deployments/
     application_name|deployment.xml#Classloader_1276887608392)}
     {classloaderMode mode}
     {configs {}}
     {deploymentId 1}
     {startingWeight 10000}
     {targetMappings {(cells/cell_name/applications/application_nameear/deployments/
     application_name|deployment.xml#DeploymentTargetMapping_1276887608392)}}
     {uri WAR_file_name.war}
     ```

Modifying class loader modes for applications using wsadmin scripting

You can modify class loader modes for an application with scripting and the wsadmin tool.

Before you begin

There are two ways to complete this task. The example in this topic uses the AdminConfig object to create and configure a shared library. Alternatively, you can use the createSharedLibrary script in the AdminResources script library to configure shared libraries.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

About this task

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Procedure

1. Start the wsadmin scripting tool.
2. Retrieve the configuration ID of the object that you want to modify and set it to the dep variable. For example:
   - Using Jacl:

     ```
     set dep [$AdminConfig getid /Deployment:ivtApp/]
     ```
   - Using Jython:

     ```
     dep = AdminConfig.getid('/Deployment:ivtApp/')
     ```
3. Identify the deployed object and set it to the depObject variable. For example:
   - Using Jacl:

     ```
     set depObject [$AdminConfig showAttribute $dep deployedObject]
     ```
   - Using Jython:

     ```
     depObject = AdminConfig.showAttribute(dep, 'deployedObject')
     ```
4. Identify the class loader and set it to the classldr variable. For example:
   - Using Jacl:

     ```
     set classldr [$AdminConfig showAttribute $depObject classloader]
     ```
   - Using Jython:

     ```
     classldr = AdminConfig.showAttribute(depObject, 'classloader')
     ```
5. Show the current attribute values of the configuration object with the **showall** command, for example:
   - Using Jacl:

     ```
     $AdminConfig showall $classldr
     ```

     Example output:

     ```
     {libraries {}} {mode PARENT_FIRST}
     ```
   - Using Jython:

     ```
     print AdminConfig.showall(classldr)
     ```

     Example output:

     ```
     [libraries []] [mode PARENT_FIRST]
     ```
6. Modify the attributes of the configuration object with the **modify** command, for example:
   - Using Jacl:

     ```
     $AdminConfig modify $classldr {{mode PARENT_LAST}}
     ```
   - Using Jython:

```
AdminConfig.modify(classldr, [['mode', 'PARENT_LAST']])
```

7. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

8. Verify the changes that you made to the attribute value with the **showall** command, for example:
   - Using Jacl:

     ```
     $AdminConfig showall $classldr
     ```

     Example output:

     ```
     {libraries {}} {mode PARENT_LAST}
     ```

   - Using Jython:

     ```
     AdminConfig.showall(classldr)
     ```

     Example output:

     ```
     [libraries []] [mode PARENT_LAST]
     ```

Modifying the starting weight of applications using wsadmin scripting

You can use the wsadmin tool and scripting to modify the starting weight of an application.

Before you begin

About this task

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

To modify the starting weight of an application, perform the following steps:

Procedure

1. Start the wsadmin scripting tool.
2. Retrieve the configuration ID of the object that you want to modify and set it to the dep variable. For example:
   - Using Jacl:

     ```
     set dep [$AdminConfig getid /Deployment:MyApp/]
     ```

   - Using Jython:

     ```
     dep = AdminConfig.getid("/Deployment:MyApp/")
     ```

3. Identify the deployed object and set it to the depObject variable. For example:
   - Using Jacl:

     ```
     set depObject [$AdminConfig showAttribute $dep deployedObject]
     ```

   - Using Jython:

     ```
     depObject = AdminConfig.showAttribute(dep, "deployedObject")
     ```

4. Show the current attribute values of the configuration object with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $depObject startingWeight
     ```

     Example output:

     ```
     {startingWeight 1}
     ```

   - Using Jython:

     ```
     AdminConfig.show(depObject, 'startingWeight')
     ```

Example output:

```
[startingWeight 1]
```

5. Modify the attributes of the configuration object with the **modify** command, for example:
   - Using Jacl:

     ```
     $AdminConfig modify $depObject {{startingWeight 2}}
     ```
   - Using Jython:

     ```
     AdminConfig.modify(depObject, [['startingWeight', '2']])
     ```
6. Verify the changes that you made to the attribute value with the **show** command, for example:
   - Using Jacl:

     ```
     $AdminConfig show $depObject startingWeight
     ```

     Example output:

     ```
     {startingWeight 2}
     ```
   - Using Jython:

     ```
     AdminConfig.show(depObject, 'startingWeight')
     ```

     Example output:

     ```
     [startingWeight 2]
     ```
7. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Configuring namespace bindings using the wsadmin scripting tool

Use this topic to configure name space bindings with the Jython or Jacl scripting languages and the wsadmin tool.

Before you begin

About this task

Use this task and the following examples to configure string, Enterprise JavaBeans (EJB), CORBA, or indirect name space bindings on a cell.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the cell and assign it to the cell variable.

   Using Jacl:

   ```
   set cell [$AdminConfig getid /Cell:mycell/]
   ```

   Example output:

   ```
   mycell(cells/mycell|cell.xml#Cell_1)
   ```

   Using Jython:

   ```
   cell = AdminConfig.getid('/Cell:mycell/')
   print cell
   ```

   You can change this example to configure on a node or server here.
3. Add a new name space binding on the cell. There are four binding types to choose from when configuring a new name space binding. They are string, EJB, CORBA, and indirect.
   - To configure a string type name space binding:

     Using Jacl:

     ```
     $AdminConfig create StringNameSpaceBinding $cell {{name binding1} {nameInNameSpace
     myBindings/myString} {stringToBind "This is the String value that gets bound"}}
     ```

Example output:

```
binding1(cells/mycell|namebindings.xml#StringNameSpaceBinding_1)
```

Using Jython:

```
print AdminConfig.create('StringNameSpaceBinding', cell, [['name', 'binding1'],
['nameInNameSpace', 'myBindings/myString'], ['stringToBind', "This is the String value that gets bound"]])
```

- To configure an EJB type name space binding:

  Using Jacl:

```
$AdminConfig create EjbNameSpaceBinding $cell {{name binding2} {nameInNameSpace myBindings/myEJB}
{applicationNodeName mynode} {bindingLocation SINGLESERVER} {applicationServerName server1}
{ejbJndiName ejb/myEJB}}
```

  Using Jython:

```
print AdminConfig.create('EjbNameSpaceBinding', cell, [['name', 'binding2'],  ['nameInNameSpace',
'myBindings/myEJB'], ['applicationNodeName', 'mynode'],  ['bindingLocation', 'SINGLESERVER'],
['applicationServerName', 'server1'], ['ejbJndiName', 'ejb/myEJB']])
```

  This example is for an EJB located in a server. For an EJB in a cluster, change the configuration example to:

  Using Jacl:

```
$AdminConfig create EjbNameSpaceBinding $cell {{name binding2} {nameInNameSpace myBindings/myEJB}
{bindingLocation SERVERCLUSTER} {applicationServerName cluster1} {ejbJndiName ejb/myEJB}}
```

  Using Jython:

```
print AdminConfig.create('EjbNameSpaceBinding', cell, [['name','binding2'],
['nameInNameSpace','myBindings/myEJB'], ['bindingLocation','SERVERCLUSTER'],
['applicationServerName','cluster1'], ['ejbJndiName','ejb/myEJB']])
```

  Example output:

```
binding2(cells/mycell|namebindings.xml#EjbNameSpaceBinding_1)
```

- To configure a CORBA type name space binding:

  Using Jacl:

```
$AdminConfig create CORBAObjectNameSpaceBinding $cell {{name binding3} {nameInNameSpace
myBindings/myCORBA}{corbanameUrl corbaname:iiop:somehost.somecompany.com:2809#stuff/MyCORBAOjbect}
{federatedContext false}}
```

  Example output:

```
binding3(cells/mycell|namebindings.xml#CORBAObjectNameSpaceBinding_1)
```

  Using Jython:

```
print AdminConfig.create('CORBAObjectNameSpaceBinding', cell, [['name', 'binding3'],['nameInNameSpace',
'myBindings/myCORBA'],['corbanameUrl','corbaname:iiop:somehost.somecompany.com:2809#stuff/MyCORBAOjbect'],
['federatedContext', 'false']])
```

- To configure an indirect type name space binding:

  Using Jacl:

```
$AdminConfig create IndirectLookupNameSpaceBinding $cell
{{name binding4} {nameInNameSpace myBindings/myIndirect} {providerURL
corbaloc::myCompany.com:9809/NameServiceServerRoot} {jndiName jndi/name/for/EJB}}
```

  Example output:

```
binding4(cells/mycell|namebindings.xml#IndirectLookupNameSpaceBinding_1)
```

  Using Jython:

```
print AdminConfig.create('IndirectLookupNameSpaceBinding', cell, [['name', 'binding4'],
['nameInNameSpace','myBindings/myIndirect'], ['providerURL', 'corbaloc::myCompany.com:9809/NameServiceServerRoot'],
['jndiName', 'jndi/name/for/EJB']])
```

4. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

WSScheduleCommands command group of the AdminTask object

You can use the Jython or Jacl scripting languages to manage deployed applications with the wsadmin tool. The commands and parameters in the WSScheduleCommands group can be used to create and manage scheduler settings in your configuration. Schedulers enable J2EE application tasks to run at a requested time.

The WSScheduleCommands command group for the AdminTask object includes the following commands:

- "deleteWSSchedule"
- "getWSSchedule"
- "listWSSchedules"
- "modifyWSSchedule" on page 418

deleteWSSchedule

The **deleteWSSchedule** command deletes the settings of a scheduler from the configuration.

Parameters and return values

-name
The name that uniquely identifies the scheduler. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

getWSSchedule

The **getWSSchedule** command returns the settings of the specified scheduler.

Parameters and return values

-name
The name that uniquely identifies the scheduler. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

listWSSchedules

The **listWSSchedules** command lists the scheduler.

Parameters and return values

-displayObjectNames
Set the value of this parameter to `true` to list the key set configuration objects within the scope. Set the value of this parameter to `false` to list the strings that contain the key set group name and management scope. (Boolean, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

modifyWSSchedule

The **modifyWSSchedule** command changes the settings of an existing scheduler.

Parameters and return values

-name
The name that uniquely identifies the scheduler. (String, required)

-frequency
The period of time in days to wait before checking for expired certificates. (Integer, optional)

-dayOfWeek
The day of the week to check for expired certificates. (Integer, optional)

-hour
The hour of the day to check for expired certificates. (Integer, optional)

-minute
The minute to check for expired certificates. Use this parameter with the hour parameter. (Integer, optional)

-nextStartDate
The next time, in seconds, to check for expired certificate. (Long, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

WSNotifierCommands command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure deployed applications with the wsadmin tool. The commands and parameters in the WSNotifierCommands group can be used to create and manage notifications settings. WS-Notification enables web services to use the "publish and subscribe" messaging pattern, creating a one-to-many message distribution pattern.

The WSNotifierCommands command group for the AdminTask object includes the following commands:

- "deleteWSNotifier"
- "getWSNotifier" on page 419
- "listWSNotifier" on page 419
- "modifyWSNotifier" on page 419

deleteWSNotifier

The **deleteWSNotifier** command deletes the settings of a notification configuration.

Parameters and return values

-name
The name that uniquely identifies the notification configuration. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

getWSNotifier

The **getWSNotifier** command displays the settings of a particular notification configuration.

Parameters and return values

-name
The name that uniquely identifies the notification configuration. (String, required)

Examples

Batch mode example usage:

Interactive mode example usage:

listWSNotifier

The **listWSNotifier** command lists the notifier from the configuration.

Parameters and return values

-displayObjectNames
If you set the value of this parameter to true, this command returns all notification configuration objects within the scope. If you set the value of this parameter to false, this command returns a list of strings that contain the key set group name and the management scope. (Boolean, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

modifyWSNotifier

The **modifyWSNotifier** command changes the settings of an existing notification configuration.

Parameters and return values

-name
The name that uniquely identifies the notification configuration. (String, required)

-logToSystemOut
Set the value of this parameter to true if you want the certificate expiration information to log to system out. If not, set the value of this parameter to false. (Boolean, optional)

-sendEmail
Set the value of this parameter to true if you want to email the certificate expiration information. If not, set the value of this parameter to false. (Boolean, optional)

-emailList
The list of email addresses where you want to send certificate expiration information. Separate the values in the list with colons (:). (String, optional)

Examples

Batch mode example usage:

Interactive mode example usage:

CoreGroupManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage deployed applications. The commands and parameters in the CoreGroupManagement group can be used to create and manage core groups. A core group is a high availability domain that consists of a set of processes in the same cell that can directly establish high availability relationships. A cell must contain at least one core group.

The CoreGroupManagement command group for the AdminTask object includes the following commands:

createCoreGroup

The **createCoreGroup** command creates a new core group. The core group that you create contains no members.

Target object

None

Parameters and return values

-coreGroupName
The name of the core group that you are creating. (String required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createCoreGroup {-coreGroupName MyCoreGroup} AdminConfig.save()
```

Optionally, you can use the following sample script to add a description for the new core group:

```
set core [$AdminConfig getid /Cell:myCell/CoreGroup:MyCoreGroup/] $AdminConfig
 modify $core {{description "My Description"}} $AdminConfig save
```

- Using Jython string:

```
AdminTask.createCoreGroup('[-coreGroupName MyCoreGroup]')
```

Optionally, you can use the following sample script to add a description for the new core group:

```
core = AdminConfig.getid('/Cell:myCell/CoreGroup:MyCoreGroup/')
 AdminConfig.modify(core, [['description', "This is my new description"]]) AdminConfig.save()
```

- Using Jython list:

```
AdminTask.createCoreGroup(['-coreGroupName', 'MyCoreGroup'])
```

Optionally, you can use the following sample script to add a description for the new core group:

```
core = AdminConfig.getid('/Cell:myCell/CoreGroup:MyCoreGroup/')
 AdminConfig.modify(core, [['description', "This is my new description"]]) AdminConfig.save()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.createCoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createCoreGroup (['-interactive'])
```

deleteCoreGroup

The **delete Core Group** command deletes an existing core group. The core group that you specify must not contain any members. You cannot delete the default core group.

Target object

None

Parameters and return values

-coreGroupName
The name of the existing core group that will be deleted. (String required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroup {-coreGroupName MyCoreGroup}
```

- Using Jython string:

```
AdminTask.deleteCoreGroup ('[-coreGroupName MyCoreGroup]')
```

- Using Jython list:

```
AdminTask.deleteCoreGroup (['-coreGroupName', 'MyCoreGroup'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.deleteCoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteCoreGroup ('[-interactive]')
```

doesCoreGroupExist

The **doesCore Group Exist** command returns a boolean value that indicates if the core group that you specify exists.

Target object

None

Parameters and return values

-coreGroupName
The name of the core group. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask doesCoreGroupExist {-coreGroupName MyCoreGroup}
```

- Using Jython string:

```
AdminTask.doesCoreGroupExist ('[-coreGroupName MyCoreGroup]')
```

- Using Jython list:

```
AdminTask.doesCoreGroupExist (['-coreGroupName', 'MyCoreGroup'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask doesCoreGroupExist {-interactive}
```

- Using Jython string:

```
AdminTask.doesCoreGroupExist ('[-interactive]')
```

- Using Jython list:

```
AdminTask.doesCoreGroupExist (['-interactive'])
```

getAllCoreGroupNames

The **getAll CoreGroup Names** command returns a string that contains the names of all of the existing core groups

Target object

None

Parameters and return values

- Parameters: None
- Returns: String array (String[ ])

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getAllCoreGroupNames
```

- Using Jython string:

```
AdminTask.getAllCoreGroupNames()
```

- Using Jython list:

```
AdminTask.getAllCoreGroupNames()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getAllCoreGroupNames {-interactive}
```

- Using Jython string:

```
AdminTask.getAllCoreGroupNames ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getAllCoreGroupNames (['-interactive'])
```

getCoreGroupNameForServer

The **getCore GroupName ForServer** command returns the name of the core group in which the server that you specify is currently a member.

Target object

None

Parameters and return values

-nodeName
The name of the node that contains the server. (String, required)

-serverName
The name of the server. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getCoreGroupNameForServer {-nodeName myNode -serverName myServer}
```

- Using Jython string:

```
AdminTask.getCoreGroupNameForServer ('[-nodeName myNode -server Name
 myServer]')
```

- Using Jython list:

```
AdminTask.getCoreGroupNameForServer (['-nodeName', 'myNode', '-serverName',
 'myServer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getCoreGroupName ForServer {-interactive}
```

- Using Jython string:

```
AdminTask.getCoreGroupName ForServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getCoreGroupName ForServer (['-interactive'])
```

getDefaultCoreGroupName

The **getDefault CoreGroup Name** command returns the name of the default core group.

Target object

None

Parameters and return values

- Parameters: None
- Returns: String

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getDefaultCoreGroupName
```

- Using Jython string:

```
AdminTask.getDefaultCoreGroupName()
```

- Using Jython list:

```
AdminTask.getDefaultCoreGroupName()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getDefaultCoreGroupName {-interactive}
```

- Using Jython string:

```
AdminTask.getDefaultCoreGroupName ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getDefaultCoreGroupName (['-interactive'])
```

moveClusterToCoreGroup

The **moveCluster ToCore Group** command moves all of the servers in a cluster that you specify from a core group to another core group. All of the servers in a cluster must be members of the same core group.

Target object

None

Parameters and return values

-source
The name of the core group that contains the cluster that you want to move. The core group must exist prior to running this command. The cluster that you specify must be a member of this core group. (String, required)

-target
The name of the core group where you want to move the cluster. (String, required)

-clusterName
The name of the cluster that you want to move. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask moveClusterToCoreGroup {-source OldCoreGroup -target NewCoreGroup
 -clusterName ClusterOne}
```

- Using Jython string:

```
AdminTask.moveClusterToCoreGroup ('[-source OldCoreGroup -target NewCoreGroup
 -clusterName ClusterOne]')
```

- Using Jython list:

```
AdminTask.moveClusterToCoreGroup (['-source', 'OldCoreGroup', '-target',
 'NewCoreGroup', '-clusterName', 'ClusterOne'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask moveClusterToCoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.moveClusterToCoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.moveClusterToCoreGroup (['-interactive'])
```

moveServerToCoreGroup

The **moveServer ToCore Group** command moves a server to a core group that you specify. When the server is added to the core group that you specify, it is removed from the core group where it originally resided.

Target object

None

Parameters and return values

-source
The name of the core group that contains the server that you want to move. The core group must already exist with the server that you specify being a member of the core group. (String, required)

-target
The name of the core group where you want to move the server. The core group that you specify must exist prior to running the command. (String, required)

-nodeName
The name of the node that contains the server that you want to move. (String, required)

-serverName
The name of the server that you want to move. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask moveServerToCoreGroup {-source OldCoreGroup -target NewCoreGroup
 -nodeName myNode -serverName myServer}
```

- Using Jython string:

```
AdminTask.moveServerToCoreGroup ('[-source OldCoreGroup -target NewCoreGroup
 -nodeName myNode -server Name myServer]')
```

- Using Jython list:

```
AdminTask.moveServerToCore Group(['-source', 'OldCore Group', '-target',
 'NewCoreGroup', '-node Name', 'myNode', '-serverName', 'myServer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask moveServerTo CoreGroup {-interactive}
```

- Using Jython string:

```
AdminTask.moveServerTo CoreGroup ('[-interactive]')
```

- Using Jython list:

```
AdminTask.moveServerTo CoreGroup (['-interactive'])
```

CoreGroupBridgeManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage deployed applications using scripting. The commands and parameters in the CoreGroupBridgeManagement group can be used to create and manage core group access points, TCP inbound channel port, and bridge interfaces. A bridge interface specifies a particular node and server that runs the core group bridge service.

The CoreGroupBridgeManagement command group for the AdminTask object includes the following commands:

createCoreGroupAccessPoint

The createCoreGroupAccessPoint command creates a default core group access point for the core group that you specify and adds it to the default access point group. If the default access point group does not exist, the command creates a default access point group.

Target object

Core group bridge settings object for the cell. (ObjectName, required).

Required parameters

-coreGroupName
The name of the core group for which the core group access point will be created. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createCoreGroupAccessPoint {-interactive}
```

- Using Jython:

```
AdminTask.createCoreGroupAccessPoint('-interactive')
```

deleteCoreGroupAccessPoints

The deleteCoreGroupAccessPoints command deletes all the core group access points that are associated with a group that you specify.

Target object

Core group bridge settings object for the cell. (ObjectName, required)

Required parameters

-coreGroupName
The name of the core group whose core group access points will be deleted. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroupAccessPoints (cells/rohitbuildCell01|coregroupbridge.xml#
 CoreGroupBridgeSettings_1) "-coreGroupName DefaultCoreGroup"
```

- Using Jython string:

```
AdminTask.deleteCoreGroupAccessPoints('cells/rohitbuildCell01|coregroupbridge.xml#CoreGroupBridgeSettings_1',
 '[-coreGroupName DefaultCoreGroup]')
```

- Using Jython list:

```
AdminTask.deleteCoreGroupAccessPoints('(cells/ rohitbuildCell01|coregroupbridge.xml#
 CoreGroupBridgeSettings_1)', ['-coreGroupName', 'DefaultCoreGroup'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteCoreGroupAccessPoints {-interactive}
```

- Using Jython:

```
AdminTask.deleteCoreGroupAccessPoints('-interactive')
```

exportTunnelTemplate

The exportTunnelTemplate command exports a tunnel template and its associated children to a simple properties file.

Target object

None

Required parameters

-tunnelTemplateName
Specifies the name of the tunnel template to export. (String, required)

-outputFileName
Specifies the name of the properties file to create. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.exportTunnelTemplate('[-tunnelTemplateName tunnelTemplate1 -outputFileName tunnelTemplate1.props]')
```

- Using Jython list:

```
AdminTask.exportTunnelTemplate(['-tunnelTemplateName', 'tunnelTemplate1', '-outputFileName',
 'tunnelTemplate1.props'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.exportTunnelTemplate('-interactive')
```

getNamedTCPEndPoint

The getNamedTCPEndPoint command returns the port associated with the bridge interface that you specify. The port that is returned is the one that is specified on the TCP inbound channel of the transport channel chain for bridge interface that you specify.

Target object

The bridge interface object for which the port will be listed. (ObjectName, required)

Required parameters

None

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getNamedTCPEndPoint (cells/rohitbuildCell01|coregroupbridge.xml#BridgeInterface_2)
```

- Using Jython string:

```
AdminTask.getNamedTCPEndPoint('cells/rohitbuildCell01|coregroupbridge.xml#BridgeInterface_2')
```

- Using Jython list:

```
AdminTask.getNamedTCPEndPoint('cells/rohitbuildCell01|coregroupbridge.xml#BridgeInterface_2')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getNamedTCPEndPoint {-interactive}
```

- Using Jython string:

```
AdminTask.getNamedTCPEndPoint('-interactive')
```

importTunnelTemplate

The importTunnelTemplate command imports a tunnel template and its children to the cell configuration.

Target object

None

Required parameters

-inputFileName
Specifies the name of the tunnel template file to import. (String, required)

-bridegeInterfaceNodeName
Specifies the name of the secure proxy node to use for the core group bridge interface. (String, required)

-bridegeInterfaceServerName
Specifies the name of the secure proxy server to use for the core bridge interface. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.importTunnelTemplate('[-inputFileName tunnelTemplate1.props
 -bridegeInterfaceNodeName secureProxyNode -bridegeInterfaceServerName mySecureProxyServer]')
```

- Using Jython list:

```
AdminTask.importTunnelTemplate(['-inputFileName', 'tunnelTemplate1.props',
 '-bridegeInterfaceNodeName', 'secureProxyNode', '-bridegeInterfaceServerName', 'mySecureProxyServer'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.importTunnelTemplate('-interactive')
```

listCoreGroups

The listCoreGroups command returns a collection of core groups that are related to the core group that you specify.

Target object

The name of the core group for which the related core groups will be listed. (String, required)

Required parameters

-cgBridgeSettings
The group bridge settings object for the cell. (ObjectName, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listCoreGroups DefaultCoreGroup "-cgBridgeSettings
 (cells/rohitbuildCell01|coregroupbridge.xml# CoreGroupBridgeSettings_1)"
```

- Using Jython string:

```
AdminTask.listCoreGroups('DefaultCoreGroup', '[-cgBridgeSetting (cells/
 rohitbuildCell01|coregroupbridge.xml#CoreGroupBridgeSettings_1)]')
```

- Using Jython list:

```
AdminTask.listCoreGroups('DefaultCoreGroup', ['-cgBridgeSetting', '(cells/
 rohitbuildCell01|coregroupbridge.xml#CoreGroupBridgeSettings_1)'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listCoreGroups {-interactive}
```

- Using Jython:

```
AdminTask.listCoreGroups('-interactive')
```

listEligibleBridgeInterfaces

The listEligibleBridgeInterfaces command returns a collection of node, server, and transport channel chain combinations that are eligible to become bridge interfaces for the specified core group access point.

Target object

The core group access point object for which bridge interfaces will be listed. (ObjectName, required)

Required parameters

None

Optional parameters

None

Example output

A set of bridge interfaces. (Set of String) Each bridge interface is represented by a combination of a node, a server and a DCS channel chain: <node *name*>, <server *name*>, <DCS Channel Chain *objectName*. For example, an element of the set returned by this command may look like the following: `rohitbuild dmgr DCS-Secure(cells/rohitbuildCell/nodes/rohitbuild/servers/dmgr|server.xml#Chain_4)`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listEligibleBridgeInterfaces
 CGAP_DCG_2(cells/rohitbuildCell01|coregroupbridge.xml#CoreGroupAccessPoint_1089636614062)
```

- Using Jython string:

```
AdminTask.listEligibleBridgeInterfaces('CGAP_DCG_2(cells/rohitbuildCell01|coregroupbridge.xml#
 CoreGroupAccessPoint_1089636614062)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listEligibleBridgeInterfaces {-interactive}
```

- Using Jython:

```
AdminTask.listEligibleBridgeInterfaces('-interactive')
```

Chapter 15. Configuring servers with scripting

You can use the wsadmin tool to configure application servers in your environment. An application server configuration provides settings that control how an application server provides services for running applications and their components.

About this task

After installing the product, you might need to configure additional options for your application server. With the wsadmin tool, you can use the commands for the AdminTask and AdminConfig objects to retrieve configuration IDs and invoke operations on the objects to configure the application server. Alternatively, you can use the script library to perform specific operations to configure your application servers. The scripting library provides a set of procedures to automate the most common application server administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

You might need to complete one or more of the following tasks to configure your application server:

Procedure

- Create servers. Use the commands in the ServerManagement command group for the AdminTask object or the AdminServerManagement script library to create a new application server, web server, proxy server, or generic server.
- Configure a unique HTTP session clone ID for each application server. If you require session affinity, use the commands in this topic to configure an HTTP session clone ID for each application server.
- Configure database session persistence. You can use the AdminConfig object to configure database persistence.
- Configure the Java virtual machine to run in debug mode. Use the commands in the ServerManagement command group for the AdminTask object or the configureJavaVirtualMachine script in the AdminServerManagement script library to modify your Java virtual machine (JVM) configuration.
- Configure EJB containers. You can use the AdminConfig object or the configureEJBContainer script in the AdminServerManagement script library to configure Enterprise JavaBeans (EJB) containers in your configuration.
- Configure the Performance Monitoring Infrastructure. You can use the wsadmin tool to configure the Performance Monitoring Infrastructure (PMI) in your environment. PMI enables the server to collect performance data from various product components. PMI provides information about average system resource usage statistics, with no correlation between the data across different components.
- Configure Object Request Broker (ORB) services. You can use the AdminConfig object or the configureORBService script in the AdminServerManagement script library to configure an ORB service in your environment. An ORB manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.
- Configure processes. You can use the AdminConfig object or the configureProcessDefintion script in the AdminServerManagement script library to configure processes in your application server configuration. Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.
- Configure the runtime transaction service. Use the AdminControl object or the configureTransactionService script in the AdminServerManagement script library to configure transaction properties for servers. The transaction service is a server runtime component that coordinates updates to multiple resource managers to ensure atomic updates of data. Transactions are started and ended by applications or the container in which the applications are deployed.

- Set port numbers to the serverindex.xml file. You can use the AdminConfig object, AdminTask object, or the scripts in the AdminServerManagement script library to modify the port numbers specified in the `serverindex.xml` file. The end points of the `serverindex.xml` file are part of different objects in the configuration.
- Disable components. You can use the AdminConfig object or the configureStateManageable script in the AdminServerManagement script library to disable components by invoking operations. This topic describes how to disable the nameServer component of the product. You can modify the examples in this topic to disable other components.
- Disable the trace service. Refer to the topic on disabling trace service for more information.
- Modify variables. Refer to the topic on modifying variables for more information.
- Increase the Java virtual machine heap size. Refer to the topic for more information.

Creating a server using scripting

Use the commands in the ServerManagement command group for the AdminTask object or the AdminServerManagement script library to create a new application server, web server, proxy server, or generic server.

Before you begin

There are three ways to complete this task. This topic uses the AdminConfig object and the commands for the AdminTask object to create a new server configuration. Alternatively, you can use the scripts in the AdminServerManagement script library to create an application server, web server, proxy server, or generic server.

Procedure

1. Start the wsadmin scripting tool.
2. Obtain the configuration ID of the node object.

   The following examples obtain the configuration ID of the node object and assign it to the `node` variable. In these examples, *node_name* is the name of the node to which you are adding the new server, *server_name* is the name of the server you are creating, *template_name* is the name of the template you want used to create the server. The AdminTask.createApplicationServer() command requires you to specify a node name. The node name must be enclosed within single quotation marks. The AdminConfig.create() command requires you to specify the configuration ID of the node object. If you do not know the configuration ID, issue the AdminConfig.getid('/Node:*node_name*') command to obtain that information.

   - Using Jacl:

     ```
     set node [$AdminConfig getid /Node:node_name/]
     ```

   - Using Jython:

     ```
     node = AdminConfig.getid('/Node:node_name/')
     ```

   If you want to display the configuration ID of the node object, issue the following command:

   ```
   print node
   ```

3. Determine whether to use the AdminConfig or AdminTask object to create the server.
4. Create the server.
   - The following example uses the commands for the AdminTask object to create a server:

     Using the AdminTask object:

     - Using Jacl:

       ```
       $AdminTask createApplicationServer node_name
       {-name server_name –templateName template_name}
       ```

     - Using Jython:

```
AdminTask.createApplicationServer('node_name',
['-name', 'server_name', '-templateName', 'template_name'])
```

- The following example uses the AdminConfig object to create a server. In these examples `node` is the node variable to which the configuration ID of the node object is assigned.

  Using the AdminConfig object:

  – Using Jacl:

  ```
  $AdminConfig create Server $node {name server_name}
  ```

  – Using Jython:

  ```
  AdminConfig.create('Server', node, ['name', 'server_name']
  ```

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Configuring a unique HTTP session clone ID for each application server using scripting

You can use scripting and the wsadmin tool to configuring a unique HTTP session clone ID for each application server.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 topic for more information.

About this task

Perform the following steps to configure a unique HTTP session clone ID for each application server. Within these steps, the following variables apply to the Jython and Jacl commands:

- *node_name* is the affected node within your configuration.
- *server_name* is the affected server within your configuration.
- *cell_name* is the affected cell within your configuration.
- *unique_value* is 8 to 9 unique alphanumeric characters. For example, `test1234`.

Procedure

1. Retrieve the node name and server name values. Assign those two values to the server variable.
   - Using Jacl:

     ```
     set server [$AdminConfig getid /Node:node_name/Server:server_name/]
     ```

     Example output:

     ```
     server_name(cells/cell_name/nodes/node_name/servers/
     server_name|server.xml#Server_1265038035855)
     ```

   - Using Jython:

     ```
     server = AdminConfig.getid('/Node:node_name/Server:server_name/')
     ```

     Example output: None
2. Retrieve the name of the web container, which is associated with the node and server values that are identified in the previous step, and assign the value to the wc variable.
   - Using Jacl:

     ```
     set wc [$AdminConfig list WebContainer $server]
     ```

     Example output:

     ```
     (cells/cell_name/nodes/node_name/servers/server_name
     |server.xml#WebContainer_1265038035855)
     ```

- Using Jython:

  ```
  wc = AdminConfig.list('WebContainer', server)
  ```

  Example output: None

3. Create the HTTPSessionCloneId custom property using the node, server, and web container values that were assigned in the previous steps.
   - Using Jacl:

     ```
     $AdminConfig create Property $wc {{name "HttpSessionCloneId"} {description ""}
     {value "value"} {required "false"}}
     ```

     Example output:

     ```
     HttpSessionCloneId(cells/cell_name/nodes/node_name/servers/
     server_name|server.xml#Property_1265840905884)
     ```

   - Using Jython:

     ```
     AdminConfig.create('Property', wc, '[[validationExpression ""][name "HttpSessionCloneId"]
     [description ""][value "value"][required "false"]]')
     ```

     Example output:

     ```
     'HttpSessionCloneId(cells/cell_name/nodes/node_name/servers/
     server_name|server.xml#Property_1265841318634)'
     ```

4. Save the configuration changes. For more information, see the documentation on saving configuration changes with the wsadmin tool.

Configuring database session persistence using scripting

You can use scripting and the wsadmin tool to configure database persistence.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 topic for more information.

About this task

Perform the following steps to configure database persistence. Within these steps, the following variables apply to the Jython and Jacl commands:

- *node_name* is the affected node within your configuration.
- *server_name* is the affected server within your configuration.
- *cell_name* is the affected cell within your configuration.
- *db2_administrator* is the ID for the database administrator.
- *db2_password* is the password for the ID that is associated with the database administrator.

Procedure

1. Retrieve the configuration ID of the server to enable database persistence and store its values in the server variable.
   - Using Jacl:

     ```
     set server [$AdminConfig getid /Node:node_name/Server:server_name/]
     ```

     Example output:

     ```
     server_name(cells/cell_name/nodes/node_name/servers/server_name
     |server.xml#Server_1265038035855)
     ```

   - Using Jython:

     ```
     server = AdminConfig.getid('/Node:node_name/Server:server_name/')
     ```

     Example output: None

2. Retrieve the name of the session manager, which is associated with the server values in the previous step, and assign the session manager to the sm variable.
   - Using Jacl:

     ```
     set sm [$AdminConfig list SessionManager $server]
     ```

     Example output:

     ```
     (cells/cell_name/nodes/node_name/servers/server_name
     |server.xml#SessionManager_1256932276179)
     ```

   - Using Jython:

     ```
     sm = AdminConfig.list('SessionManager', server)
     ```

     Example output: None
3. Add the database session persistence mode value to the sm variable, which already contains the session manager value from the previous steps.
   - Using Jacl:

     ```
     $AdminConfig modify $sm {{sessionPersistenceMode "DATABASE"}}
     ```

     Example output: None
   - Using Jython:

     ```
     AdminConfig.modify(sm,'[[sessionPersistenceMode "DATABASE"]]')
     ```

     Example output: None
4. Retrieve the database session persistence value for the session manager and the database session persistence mode that are set to the sm variable. Set this value to the sesdb variable.
   - Using Jacl:

     ```
     set sesdb [$AdminConfig list SessionDatabasePersistence $sm]
     ```

     Example output:

     ```
     (cells/cell_name/nodes/node_name/servers/server_name
     |server.xml#SessionDatabasePersistence_1256932276179)
     ```

   - Using Jython:

     ```
     sesdb = AdminConfig.list('SessionDatabasePersistence',sm)
     ```

     Example output: None
5. Modify the sesdb variable to include the user ID and password to access the database and the table space name; and the Java naming and directory interface (JNDI) name.
   - Using Jacl:

     ```
     $AdminConfig modify $sesdb { {userId "db2_administrator"} {password "db2_password"}
      {tableSpaceName ""} {datasourceJNDIName "jdbc/SessionDataSource"} }
     ```

     Example output: None
   - Using Jython:

     ```
     AdminConfig.modify(sesdb,'[[userId "db2_administrator"] [password "db2_password"]
     [tableSpaceName ""] [datasourceJNDIName "jdbc/SessionDataSource"]]')
     ```

     Example output: None
6. Save the configuration changes. For more information, see the documentation on saving configuration changes with the wsadmin tool.

Configuring the Java virtual machine using scripting

Use the wsadmin tool to configure settings for a Java virtual machine (JVM). As part of configuring an application server, you might define settings that enhance the way your operating system uses of the Java virtual machine.

About this task

There are three ways to perform this task. Use the steps in this topic to use the setJVMDebugMode command for the AdminTask object or the AdminConfig object to modify your JVM configuration. Alternatively, you can use the configureJavaVirtualMachine Jython script in the AdminServerManagement script library to enable, disable, or configure the debug mode for the JVM. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure JVM settings using the configureJavaVirtualMachine script:

```
AdminServerManagement.configureJavaVirtualMachine(nodeName, serverName, debugMode, debugArgs, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

The Java virtual machine (JVM) is an interpretive computing engine responsible for running the byte codes in a compiled Java program. The JVM translates the Java byte codes into the native instructions of the host machine. The application server, being a Java process, requires a JVM in order to run, and to support the Java applications running on it. JVM settings are part of an application server configuration.

Procedure

1. Start the wsadmin scripting tool.
2. There are two ways to complete this step. You can use the setJVMDebugMode command for the AdminTask object or the AdminConfig object to modify your JVM configuration. Choose one of the following configuration methods:
   - Using the AdminTask object:
     - Using Jacl:

```
$AdminTask setJVMDebugMode {-serverName server1 -nodeName node1 -debugMode true}
```

     - Using Jython:

```
AdminTask.setJVMDebugMode (['-serverName', 'server1', '-nodeName', 'node1', '-debugMode', 'true'])
```

   - Using the AdminConfig object:
     a. Identify the server and assign it to the `server1` variable, as the following example demonstrates:
        - Using Jacl:

```
set server1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

        - Using Jython:

```
server1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print server1
```

          Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

     b. Identify the JVM that belongs to the server of interest and assign it to the `jvm` variable, as the following example demonstrates:
        - Using Jacl:

```
set jvm [$AdminConfig list JavaVirtualMachine $server1]
```

        - Using Jython:

```
jvm = AdminConfig.list('JavaVirtualMachine', server1)
print jvm
```

          Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaVirtualMachine_1)
```

     c. Modify the JVM to enable debugging, as the following example demonstrates:
        - Using Jacl:

```
$AdminConfig modify $jvm {{debugMode true} {debugArgs "-Djava.compiler=NONE -Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"}}
```

        - Using Jython:

```
AdminConfig.modify(jvm, [['debugMode', 'true'], ['debugArgs',  "-Djava.compiler=NONE -Xdebug
-Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"]])
```

3. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Configuring EJB containers using wsadmin

You can use the AdminConfig object or the wsadmin script library to configure Enterprise JavaBeans (EJB) containers in your configuration.

About this task

There are two ways to perform this task. Complete the steps in this topic to use the AdminConfig object to modify your EJB container configuration. Alternatively, you can use the configureEJBContainer Jython script in the AdminServerManagement script library to configure EJB containers. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure EJB containers using the configureEJBContainer script:

```
AdminServerManagement.configureEJBContainer(nodeName, serverName, ejbName, passivationDir,
 defaultDatasourceJNDIName)
```

For additional information and argument definitions, see the documentation for the AdminServerManagement script library.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the application server of interest.

   The following examples identify the application server and assign it to the `serv1` variable:

   - Using Jacl:

     ```
     set serv1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     serv1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     print serv1
     ```

   The previous commands consist of the following elements:

Table 432. Command elements. The following table explains elements in the Jacl and Jython examples:

Element	Description
`set`	Jacl command
`serv1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the application server configuration
`getid`	AdminConfig command
`/Cell:`*`mycell`*`/Node:`*`mynode`*`/Server:`*`server1`*`/`	The hierarchical containment path of the configuration object
`Cell`	Object type
`mycell`	Optional name of the object
`Node`	Object type
`mynode`	Optional name of the object
`Server`	Object type

Table 432. Command elements (continued). The following table explains elements in the Jacl and Jython examples:

server1	Optional name of the object

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. Identify the EJB container that belongs to the server.

   The following example identifies the EJB container for the server of interest and assigns it to the `ejbc1` variable:

   - Using Jacl:

     ```
     set ejbc1 [$AdminConfig list EJBContainer $serv1]
     ```

   - Using Jython:

     ```
     ejbc1 = AdminConfig.list('EJBContainer', serv1)
     print ejbc1
     ```

   The previous commands consist of the following elements:

Table 433. Command elements. The following table explains elements in the Jacl and Jython examples:

Element	Description
`set`	Jacl command
`ejbc1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	The object that represents the application server configuration
`list`	AdminConfig command
`EJBContainer`	The object type The name of the object type that you specify is the one based on the XML configuration files and does not have to be the same name that the administrative console displays.
`serv1`	Evaluates to the ID of the server of interest

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#EJBContainer_1)
```

4. View each of the attributes of the EJB container.

   The following example displays the EJB container attributes but does not display nested attributes:

   - Using Jacl:

     ```
     $AdminConfig show $ejbc1
     ```

     Example output:

     ```
     {cacheSettings (cells/mycell/nodes/mynode/servers/
     server1|server.xml#EJBCache_1)}
     {components {}}
     {inactivePoolCleanupInterval 30000}
     {parentComponent (cells/mycell/nodes/mynode/servers/
     server1|server.xml#ApplicationServer_1)
     {passivationDirectory ${USER_INSTALL_ROOT}/temp}
     {properties {}}
     {services {(cells/mycell/nodes/mynode/servers/
     server1|server.xml#MessageListenerService_1)}
     {stateManagement (cells/mycell/nodes/mynode/servers/
     server1|server.xml#StateManageable_10)}
     ```

- Using Jython:

```
print AdminConfig.show(ejbc1)
```

Example output:

```
[cacheSettings (cells/mycell/nodes/myode/servers/
server1|server.xml#EJBCache_1)]
[components []]
[inactivePoolCleanupInterval 30000]
[parentComponent (cells/mycell/nodes/myode/servers/
server1|server.xml#ApplicationServer_1)
[passivationDirectory ${USER_INSTALL_ROOT}/temp]
[properties []]
[services [(cells/mycell/nodes/myode/servers/
server1|server.xml#MessageListenerService_1)]
[stateManagement (cells/mycell/nodes/mynode/servers/
server1|server.xml#StateManageable_10)]
```

The previous commands consist of the following elements:

Table 434. Command elements. The following table explains elements in the Jacl and Jython examples:

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`print`	Jython command
`AdminConfig`	The object that represents the application server configuration
`showall`	AdminConfig command
`ejbc1`	evaluates to the ID of the enterprise bean container

The following example displays the EJB container attributes, including nested attributes:

- Using Jacl:

```
$AdminConfig showall $ejbc1
```

Example output:

```
{cacheSettings {{cacheSize 2053}
  {cleanupInterval 3000}}}
{components {}}
{inactivePoolCleanupInterval 30000}
{parentComponent (cells/mycell/nodes/mynode/servers/
server1|server.xml#ApplicationServer_1)}
{passivationDirectory ${USER_INSTALL_ROOT}/temp}
{properties {}}
{services {{{context (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)}
  {listenerPorts {}}
  {properties {}}
  {threadPool {{inactivityTimeout 3500}
    {isGrowable false}
    {maximumSize 50}
    {minimumSize 10}}}}}}
{stateManagement {{initialState START}
  {managedObject (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)}}}
```

- Using Jython:

```
print AdminConfig.showall(ejbc1)
```

Example output:

```
[cacheSettings [[cacheSize 2053]
  [cleanupInterval 3000]]]
[components []]
[inactivePoolCleanupInterval 30000]
[parentComponent (cells/mycell/nodes/mynode/servers/
server1|server.xml#ApplicationServer_1)]
[passivationDirectory ${USER_INSTALL_ROOT}/temp]
[properties []]
[services [[[context (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)]
  [listenerPorts []]
  [properties []]
  [threadPool [[inactivityTimeout 3500]
    [isGrowable false]
    [maximumSize 50]
    [minimumSize 10]]]]]]
[stateManagement {{initialState START]
  [managedObject (cells/mycell/nodes/mynode/servers/
server1|server.xml#EJBContainer_1)]]]
```

The previous commands consist of the following elements:

Table 435. Command elements. The following table explains elements in the Jacl and Jython examples:

Element	Description
$	Jacl operator for substituting a variable name with its value
print	Jython command
AdminConfig	The object that represents the application server configuration
showall	AdminConfig command
ejbc1	evaluates to the ID of the enterprise bean container

5. Modify the attributes.

   The following example modifies the enterprise bean cache settings and it includes nested attributes:

   - Using Jacl:

     ```
     $AdminConfig modify $ejbc1 {{cacheSettings
     {{cacheSize 2500} {cleanupInterval 3500}}}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(ejbc1, [['cacheSettings',
     [['cacheSize', 2500],  ['cleanupInterval', 3500]]]])
     ```

   The previous commands consist of the following elements:

Table 436. Command elements. The following table explains elements in the Jacl and Jython examples:

Element	Description
$	Jacl operator for substituting a variable name with its value
AdminConfig	The object that represents the application server configuration
modify	AdminConfig command
ejbc1	Evaluates to the ID of the enterprise bean container
cacheSettings	The attribute of modify objects
cacheSize	The attribute of modify objects
2500	The value of the cacheSize attribute

Table 436. Command elements (continued). The following table explains elements in the Jacl and Jython examples:

`cleanupInterval`	The attribute of modify objects
`3500`	The value of the cleanupInterval attribute

The following example modifies the cleanup interval attribute:

- Using Jacl:

```
$AdminConfig modify $ejbc1 {{inactivePoolCleanupInterval 15000}}
```

- Using Jython:

```
AdminConfig.modify(ejbc1, [['inactivePoolCleanupInterval', 15000]])
```

The previous commands consist of the following elements:

Table 437. Command elements. The following table explains elements in the Jacl and Jython examples:

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	The object that represents the application server configuration
`modify`	AdminConfig command
`ejbc1`	Evaluates to the ID of the enterprise bean container
`inactivePoolCleanupInterval`	The attribute of modify objects
`15000`	The value of the inactivePoolCleanupInterval attribute

6. Save the changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Configuring timer manager custom properties using the wsadmin tool

You can use the wsadmin tool to set custom properties for the timer manager.

About this task

The **lateTimerTime** custom property represents the number of seconds beyond which a late-firing timer causes an informational message to be logged. The informational message is logged once per timer manager. The default value is 5 seconds and a value of 0 disables this property.

Procedure

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application server, and assign it to the server variable. Use the AdminConfig object, and the getid command to retrieve the configuration ID of the server and assign it to the *<varname>* variable:
   - Using Jacl:

```
set serv1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
serv1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
```

3. Identify the timer manager that belongs to the server, and assign it to the timer manager variable. Use the AdminConfig object, the list command, and the server variable to retrieve the timer manager and assign it to the <varname> variable:
   - Using Jacl:

```
set timermanager1 [$AdminConfig list TimerManagerInfo $serv1]
```

- Using Jython:

```
timermanager1 = AdminConfig.list('TimerManagerInfo', 'serv1')
```

4. Create a new J2EEResourcePropertySet property set for the timer manager, and assign it to the timer manager property set variable. Use the AdminConfig object, the create command, and the timer manager variable to create a new J2EEResourcePropertySet and assign it to the *<varname>* variable:
   - Using Jacl:

```
set timermanagerpropset1 [$AdminConfig create J2EEResourcePropertySet $timermanager1 {}]
```

   - Using Jython:

```
timermanagerpropset1 = AdminConfig.create('J2EEResourcePropertySet', 'timermanager1', [])
```

5. Create a new J2EEResourceProperty for the J2EEResourcePropertySet, and assign it to the timer manager property variable.

   Use the AdminConfig object, the create command, and the property set variable to create a new J2EEResourceProperty for the lateTimerTime custom property and assign it to the *<varname>* variable:
   - Using Jacl:

```
set timermanagerproperty1 [$AdminConfig create J2EEResourceProperty $timermanagerpropset1
{{name "lateTimerTime"} {value "10"} {description "Custom lateTimerTime"} {type "java.lang.String"}
{required "false"}}]
```

   - Using Jython:

```
timermanagerproperty1 = AdminConfig.create('J2EEResourceProperty', 'timermanagerpropset1',
'[[name "lateTimerTime"] [value "10"] [description "Custom lateTimerTime"] [type "java.lang.String"]
[required "false"]]')
```

   The following parameters exist for the new J2EEResourceProperty property:

   Name

   lateTimerTime

   Value

   Number of seconds

   Description

   Specify a description

   Type

   Select java.lang.String
6. Save the configuration changes.

   Enter the following command to save your changes:
   - Using Jacl:

     ```
     $AdminConfig save
     ```

   - Using Jython:

     ```
     AdminConfig.save()
     ```
7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to all active nodes, for example:
   - Using Jacl:

     ```
     $AdminNodeManagement syncActiveNodes
     ```

   - Using Jython:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

Results

You have created and configured custom properties for the timer manager using the wsadmin tool.

Configuring work manager custom properties using the wsadmin tool

You can use the wsadmin tool to set custom properties for the work manager.

About this task

The **lateWorkTime** custom property represents the number of seconds beyond which late-starting work must cause an informational message to be logged. The informational message is logged once per work manager. The default value is 60 seconds and a value of 0 disables this property.

The **lateAlarmTime** custom property represents the number of seconds beyond which a late-firing alarm must cause an informational message to be logged. The informational message is logged once per work manager. The default value is 5 seconds and a value of 0 disables this property.

Procedure

1. Launch the wsadmin scripting tool using the Jython scripting language.
2. Identify the application server, and assign it to the server variable. Use the AdminConfig object, and the getid command to retrieve the configuration ID of the server and assign it to the *<varname>* variable:
   - Using Jacl:

```
set serv1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
serv1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
```

3. Identify the work manager that belongs to the server, and assign it to the work manager variable. Use the AdminConfig object, the list command, and the server variable to retrieve the work manager and assign it to the *<varname>* variable:
   - Using Jacl:

```
set workmanager1 [$AdminConfig list WorkManagerInfo $serv1]
```

   - Using Jython:

```
workmanager1 = AdminConfig.list('WorkManagerInfo', 'serv1')
```

4. Create a new J2EEResourcePropertySet property set for the work manager, and assign it to the work manager property set variable. Use the AdminConfig object, the create command, and the work manager variable to create a new J2EEResourcePropertySet and assign it to the *<varname>* variable:
   - Using Jacl:

```
set workmanagerpropset1 [$AdminConfig create J2EEResourcePropertySet $workmanager1 {}]
```

   - Using Jython:

```
workmanagerpropset1 = AdminConfig.create('J2EEResourcePropertySet', 'workmanager1', [])
```

5. Create a new J2EEResourceProperty for the J2EEResourcePropertySet, and assign it to the work manager property variable.

   Use the AdminConfig object, the create command, and the property set variable to create a new J2EEResourceProperty for the lateWorkTime custom property and assign it to the *<varname>* variable:
   - Using Jacl:

```
set workmanagerproperty1 [$AdminConfig create J2EEResourceProperty $workmanagerpropset1 {{name "lateWorkTime"}
 {value "120"} {description "Custom lateWorkTime"} {type "java.lang.String"} {required "false"}}]
```

   - Using Jython:

```
workmanagerproperty1 = AdminConfig.create('J2EEResourceProperty', 'workmanagerpropset1', '[[name "lateWorkTime"]
 [value "120"] [description "Custom lateWorkTime"] [type "java.lang.String"] [required "false"]]')
```

   The following parameters exist for the new J2EEResourcePropertyproperty:

   Name

   lateWorkTime

   Value

   Number of seconds

Description

Specify a description

Type

Select java.lang.String

Use the AdminConfig object, the create command, and the property set variable to create a new J2EEResourceProperty for the lateAlarmTime custom property and assign it to the <varname> variable:

- Using Jacl:

```
set workmanagerproperty2 [$AdminConfig create J2EEResourceProperty $workmanagerpropset1 {{name "lateAlarmTime"}
 {value "10"} {description "Custom lateAlarmTime"} {type "java.lang.String"} {required "false"}}]
```

- Using Jython:

```
workmanagerproperty2 = AdminConfig.create('J2EEResourceProperty', 'workmanagerpropset1', '[[name "lateAlarmTime"]
 [value "10"] [description "Custom lateAlarmTime"] [type "java.lang.String"] [required "false"]]')
```

The following parameters exist for the new J2EEResourcePropertyproperty:

Name

lateAlarmTime

Value

Number of seconds

Description

Specify a description

Type

Select java.lang.String

6. Save the configuration changes.

   Enter the following command to save your changes:

   - Using Jacl:

     ```
     $AdminConfig save
     ```

   - Using Jython:

     ```
     AdminConfig.save()
     ```

7. In a network deployment environment only, synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to all active nodes, for example:

   - Using Jacl:

     ```
     $AdminNodeManagement syncActiveNodes
     ```

   - Using Jython:

     ```
     AdminNodeManagement.syncActiveNodes()
     ```

Results

You have created and configured custom properties for the work manager using the wsadmin tool.

Configuring the Performance Monitoring Infrastructure using scripting

You can use the wsadmin tool to configure the Performance Monitoring Infrastructure (PMI) in your environment. PMI enables the server to collect performance data from various product components. PMI provides information about average system resource usage statistics, with no correlation between the data across different components.

About this task

There are two ways to perform this task. Complete the steps in this topic to use the AdminConfig object to modify your server configuration. Alternatively, you can use the configurePerformanceMonitoringService Jython script in the AdminServerManagement script library to configure PMI. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure PMI settings using the configurePerformanceMonitoringService script:

```
AdminServerManagement.configurePerformanceMonitoringService(nodeName, serverName, enable, initialSpecLevel,
  otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the application server of interest.

   Use the AdminConfig object and the getid command to retrieve the configuration ID of the application server of interest, and assign it to the `s1` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('Cell:mycell/Node:mynode/Server:server1/')
     ```

   Table 438. Description of elements. The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`s1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the product configuration
`getid`	AdminConfig command
`Cell`	Attribute
`mycell`	Value of the Cell attribute
`Node`	Attribute
`mynode`	Value of the Node attribute
`Server`	Attribute
`server1`	Value of the Server attribute

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

3. Identify the PMI service that belongs to the server.

   Use the AdminConfig object and the list command to identify the PMI service, and assign it to the `pmi` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set pmi [$AdminConfig list PMIService $s1]
     ```

   - Using Jython:

     ```
     pmi = AdminConfig.list('PMIService', s1)
     print pmi
     ```

Table 439. Description of elements. The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`pmi`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object that represents the application server configuration
`list`	AdminConfig command
`PMIService`	AdminConfig object
`s1`	Evaluates to the ID of the application server of interest

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#PMIService_1)
```

4. Modify the PMI configuration attributes.

   Use the AdminConfig object and the modify command to modify the PMI configuration attributes, as the following example demonstrates:

   - Using Jacl:

     ```
     $AdminConfig modify $pmi {{enable true} {statisticSet all}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(pmi, [['enable', 'true'], ['statisticSet','all']]
     ```

   This example enables PMI service and sets the specification levels for all of the components in the server.

   Important: The specification levels are case-sensitive values.

Table 440. Description of specification levels. The following specification levels are valid for the components.

Specification level	Description
none	No statistics are enabled.
basic	Statistics specified in Java Enterprise Edition (Java EE), as well as top statistics like CPU usage and live HTTP sessions are enabled. This set is enabled *out-of-the-box* and provides basic performance data about runtime and application components.
extended	Basic set plus key statistics from various application server components like WLM, and dynamic caching are enabled. This set provides detailed performance data about various runtime and application components.
all	All statistics are enabled.
custom	Enable or disable statistics selectively.

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Logging Tivoli Performance Viewer data using scripting

You can use the wsadmin tool to start and stop Tivoli Performance Viewer logging in your environment.

About this task

Tivoli Performance Viewer provides an easy way to store real-time data for system resources, WebSphere Application Server pools and queues, application-related statistics, and others in log files for later retrieval.

The wsadmin tool helps you start and stop logging using the command line. You can use the command line to start and stop Tivoli Performance Viewer logging.

The following task assumes that you are using Jython script.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the server where you want logging to be started, and assign it to the following variable:

```
o tpvName = AdminControl.completeObjectName("type=TivoliPerfEngine,*" )
o tpvOName = AdminControl.makeObjectName(perfName
```

3. Create a UserPreferences object.

```
o pref = com.ibm.ws.tpv.engine.UserPreferences()
o pref.setServerName("server1")
o pref.setNodeName("mynode")
o pref.setLogFileName("tpv_log_1")
```

4. 4. Create the necessary arguments and invoke monitorServer action on the Tivoli Performance Viewer MBean.

```
o list_p = java.util.ArrayList()
o list_p.add(pref)
o params=jarray.array(list_p,java.lang.Object)

o list_s = java.util.ArrayList()
o list_s.add("com.ibm.ws.tpv.engine.UserPreferences")
o sigs = jarray.array(list_s,java.lang.String)

o print "--- TPV Calling monitorServer ---"
o AdminControl.invoke_jmx(perfOName, "monitorServer", params, sigs )
```

5. When you want to start logging, call the following operation:

```
o print "--- TPV Calling startLogging ---"
o AdminControl.invoke_jmx(perfOName, "startLogging", params, sigs )
```

6. When you want to stop logging , call the following operation:

```
o print "--- TPV : Now Stop Logging ---"
o AdminControl.invoke_jmx(perfOName, "stopLogging", params, sigs )
```

What to do next

By default, the log files are stored in the `profile_root /logs/tpv` directory on the node on which the server is running. Tivoli Performance Viewer automatically compresses the log file when it finishes writing to it to conserve space. At this point, there must only be a single log file in each .zip file and it must have the same name as the .zip file. Complete the following steps to view the log files:

1. Click **Monitoring and Tuning > Performance Viewer > View Logs** in the navigation tree.
2. Select a log file to view using either of the following options:
   - Explicit path to a log File

     Choose a log file from the machine on which the browser is currently running. Use this option if you have created a log file and transferred it to your system. Click **Browse** to open a file browser on the local machine and select the log file to upload.
   - Server file
     a. Specify the path of a log file on the server.
     b. In a stand-alone application server environment, type the path to the log file. The `profile_root /logs/tpv` directory is the default on a Windows system.
     c. Click **View Log**. The log is displayed with log control buttons at the top of the view.

Limiting the growth of JVM log files using scripting

You can use scripting to configure the size of Java virtual machine (JVM) log files. JVM logs record events or information from a running JVM.

Before you begin

There are two ways to perform this task. This topic demonstrates how to use the AdminConfig object to modify your server configuration. Alternatively, you can use the configureJavaProcessLogs Jython script in the AdminServerManagement script library to configure the JVM log settings. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure JVM log settings using the configureJavaProcessLogs script:

```
AdminServerManagement.configureJavaProcessLogs(nodeName, serverName, processLogRoot, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the application server of interest.

   Determine the configuration ID of the application server of interest and assign it to the server1 variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     print s1
     ```

     where:

Table 441. Syntax explanation. The following table describes the elements of the getid command.

`set`	is a Jacl command
`s1`	is a variable name
`$`	is a Jacl operator for substituting a variable name with its value
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`Cell`	is the object type
`mycell`	is the name of the object that will be modified
`Node`	is the object type
`mynode`	is the name of the object that will be modified
`Server`	is the object type
`server1`	is the name of the object that will be modified
`print`	a Jython command

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

3. Identify the stream log of interest.

   Determine the stream log of interest and assign it to the log variable. The following example identifies the output stream log:

   - Using Jacl:

     ```
     set log [$AdminConfig showAttribute $s1 outputStreamRedirect]
     ```

   - Using Jython:

```
log = AdminConfig.showAttribute(s1, 'outputStreamRedirect')
```

The following example identifies the error stream log:

- Using Jacl:

```
set log [$AdminConfig showAttribute $s1 errorStreamRedirect]
```

- Using Jython:

```
log = AdminConfig.showAttribute(s1, 'errorStreamRedirect')
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#StreamRedirect_2)
```

4. List the current values of the stream log.

   Use the following example to display the current values of the stream log of interest:

   - Using Jacl:

```
$AdminConfig show $log
```

   - Using Jython:

```
AdminConfig.show(log)
```

   Example output:

```
{baseHour 24}
{fileName ${SERVER_LOG_ROOT}/SystemOut.log}
{formatWrites true}
{maxNumberOfBackupFiles 1}
{messageFormatKind BASIC}
{rolloverPeriod 24}
{rolloverSize 1}
{rolloverType SIZE}
{suppressStackTrace false}
{suppressWrites false}
```

5. Modify the rotation policy for the stream log.

   The following example sets the rotation log file size to two megabytes:

   - Using Jacl:

```
$AdminConfig modify $log {{rolloverSize 2}}
```

   - Using Jython:

```
AdminConfig.modify(log, [['rolloverSize', 2]])
```

   The following example sets the rotation policy to manage itself. It is based on the age of the file with the rollover algorithm loaded at midnight, and the log file rolling over every 12 hours:

   - Using Jacl:

```
$AdminConfig modify $log {{rolloverType TIME}
{rolloverPeriod 12} {baseHour 24}}
```

   - Using Jython:

```
AdminConfig.modify(log, [['rolloverType', 'TIME'],
['rolloverPeriod', 12], ['baseHour', 24]])
```

   The following example sets the log file to roll over based on both time and size:

   - Using Jacl:

```
$AdminConfig modify $log {{rolloverType BOTH} {rolloverSize 2}
{rolloverPeriod 12} {baseHour 24}}
```

   - Using Jython:

```
AdminConfig.modify(log, [['rolloverType', 'BOTH'], ['rolloverSize', 2],
['rolloverPeriod', 12], ['baseHour', 24]])
```

6. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

ProxyManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage proxy configurations. Use the commands and parameters in the ProxyManagement group to configure proxy servers for web modules.

The ProxyManagement command group for the AdminTask object includes the following commands:

createWebModuleProxyConfig

The createWebModuleProxyConfig command creates a proxy server configuration for a web module.

Target object

Specify the deployment object that represents the application for which the system creates the web module proxy configuration.

Required parameters

-deployedObjectProxyConfigName
Specifies the name of the web module of interest. (String)

Optional parameters

-enableProxy
Specifies whether the system enables the proxy server. Specify `true` to enable the proxy server. (Boolean)

-transportProtocol
Specifies the protocol that the proxy server uses to communicate with the web module. The valid values are `HTTP`, `HTTPS`, and `ClientProtocol`. (String)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createWebModuleProxyConfig myApplication {-deployedObjectProxyConfigName
 MyWebModule -enableProxy true -transportProtocol HTTPS}
```

- Using Jython string:

```
AdminTask.createWebModuleProxyConfig('myApplication', '[-deployedObjectProxyConfigName
 MyWebModule -enableProxy true -transportProtocol HTTPS]')
```

- Using Jython list:

```
AdminTask.createWebModuleProxyConfig(myApplication, ['-deployedObjectProxyConfigName',
 'MyWebModule', '-enableProxy', 'true', '-transportProtocol', 'HTTPS'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createWebModuleProxyConfig {-interactive}
```

- Using Jython:

```
AdminTask.createWebModuleProxyConfig('-interactive')
```

deleteWebModuleProxyConfig

The deleteWebModuleProxyConfig command removes the proxy server configuration for a web module.

Target object

Specify the deployment object that represents the application from which the system deletes the web module proxy configuration.

Required parameters

-deployedObjectProxyConfigName
Specifies the name of the web module of interest. (String)

Optional parameters

None

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteWebModuleProxyConfig myApplication {-deployedObjectProxyConfigName
 MyWebModule}
```

- Using Jython string:

```
AdminTask.deleteWebModuleProxyConfig('myApplication', '[-deployedObjectProxyConfigName
 MyWebModule]')
```

- Using Jython list:

```
AdminTask.deleteWebModuleProxyConfig(myApplication, ['-deployedObjectProxyConfigName',
 'MyWebModule'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteWebModuleProxyConfig {-interactive}
```

- Using Jython:

```
AdminTask.deleteWebModuleProxyConfig('-interactive')
```

getServerSecurityLevel

The getServerSecurityLevel command displays the current security level of the secure proxy server.

Target object

Specify the configuration ID of the secure proxy server of interest.

Optional parameters

-proxyDetailsFormat
Specifies the format of the details to display about the security level of the proxy server. Specify `levels` to display details as a security level for each setting. Specify `values` to display details as the actual setting for each proxy server. (String)

Sample output

The command returns the security level of the secure proxy server. If you specify the optional parameter, the command displays additional information about the security level of the server of interest.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getServerSecurityLevel myProxyServer {-proxyDetailsFormat levels}
```

- Using Jython string:

```
AdminTask.getServerSecurityLevel('myProxyServer', '[-proxyDetailsFormat
 levels]')
```

- Using Jython list:

```
AdminTask.getServerSecurityLevel(myProxyServer, ['-proxyDetailsFormat',
 'levels'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getServerSecurityLevel {-interactive}
```

- Using Jython:

```
AdminTask.getServerSecurityLevel('-interactive')
```

setServerSecurityLevel

The setServerSecurityLevel command modifies the server security level for a secure proxy server.

Target object

Specify the configuration ID of the secure proxy server of interest.

Optional parameters

-proxySecurityLevel
Specifies the level of security to apply to the proxy server. Valid values include `High`, `Medium`, and `Low`. (String)

You can also use this parameter to specify custom security settings by specifying the security setting ID and value, as defined in the following table:

Table 442. Security settings for the secure proxy server.

This table lists the security settings for the secure proxy server.

ID	Description	Valid values
administration	Sets the administration security setting.	Specify `local` to allow local administration. Specify `remote` to allow remote administration.
routing	Sets the routing security setting. Using static routing specifies routing is performed through a flat configuration file using routing precedence that is inherent to the ordering of the directives. Requests can also be routed dynamically through a best match mechanism that determines the installed application or routing rule that corresponds to a specific request.	Specify `static` to use static routing, or specify `dynamic` to use dynamic routing.

Table 442. Security settings for the secure proxy server (continued).

This table lists the security settings for the secure proxy server.

ID	Description	Valid values
startupPermissions	Sets the startup permissions. The overall security level of the secured proxy server can be hardened by reverting the server process to run as an unprivileged user after startup. Although the secured proxy server must be started as a privileged user, changing the server process to run as an unprivileged user provides additional protection for local operating resources.	Specify `unprivileged` to run the server process as an unprivileged user, or specify `privileged` to run the server process as a privileged user.
errorPageHandling	Sets the error page handling. You can define a custom error page for each error code or a group of error codes on errors generated by the proxy server or the application server. This is done using HTTP status codes in responses to generate uniform customized error pages for the application. For security reasons, you can ensure that the error pages are read from the local file system instead of being forwarded to a custom remote application.	Specify `local` to read error pages from the local file system, or specify `remote` to allow the system to read error pages from remote applications.

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setServerSecurityLevel proxyServerID {-proxySecurityLevel
 administration=local;routing=static;startupPermissions=unprivileged ;errorPageHandling=local}
```

- Using Jython string:

```
AdminTask.setServerSecurityLevel('proxyServerID', '[-proxySecurityLevel
 administration=local;routing=static;startupPermissions=unprivileged ;errorPageHandling=local]'))
```

- Using Jython list:

```
AdminTask.setServerSecurityLevel(proxyServerID, ['-proxySecurityLevel',
 'administration=local;routing=static;startupPermissions=unprivileged ;errorPageHandling=local'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setServerSecurityLevel {-interactive}
```

- Using Jython:

```
AdminTask.setServerSecurityLevel('-interactive')
```

Configuring an ORB service using scripting

You can use the wsadmin tool to configure an Object Request Broker (ORB) service in your environment. An ORB manages the interaction between clients and servers, using the Internet InterORB Protocol (IIOP). It enables clients to make requests and receive responses from servers in a network-distributed environment.

About this task

There are two ways to perform this task. Complete the steps in this topic to use the AdminConfig object to modify your ORB configuration. Alternatively, you can use the configureORBService Jython script in the AdminServerManagement script library to configure settings for the ORB service. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure JVM settings using the configureORBService script:

```
AdminServerManagement.configureORBService(nodeName, serverName, requestTimeout, requestRetriesCount,
requestRetriesDelay, connectionCacheMax, connectionCacheMin, locateRequestTimeout, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the application server and assign it to the server variable.

   Use the AdminConfig object and the getid command to retrieve the configuration ID of the server of interest, as the following example demonstrates:

   - Using Jacl:

```
set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
s1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print s1
```

Table 443. AdminConfig getid command description. The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`s1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the WebSphere Application Server configuration
`getid`	AdminConfig command
`Cell`	Object type
`mycell`	Name of the object that will be modified
`Node`	Object type
`mynode`	Name of the object that will be modified
`Server`	Object type
`server1`	Name of the object that will be modified
`print`	Jython command

   Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. Determine the ORB that belongs to the server.

Use the AdminConfig object and the list command to identify the ORB that belongs to the server and assign it to the `orb` variable, as the following example demonstrates:

- Using Jacl:

```
set orb [$AdminConfig list ObjectRequestBroker $s1]
```

- Using Jython:

```
orb = AdminConfig.list('ObjectRequestBroker', s1)
print orb
```

Table 444. AdminConfig list command description. The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`orb`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the WebSphere Application Server configuration
`list`	AdminConfig command
`ObjectRequestBroker`	AdminConfig object
`s1`	Evaluates to the ID of server of interest
`print`	Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#ObjectRequestBroker_1)
```

4. Modify the ORB configuration attributes.

   The following example modifies the connection cache maximum and pass by value attributes. You can modify the example to change the value of other attributes.

   - Using Jacl:

```
$AdminConfig modify $orb {{connectionCacheMaximum 252} {noLocalCopies true}}
```

   - Using Jython:

```
AdminConfig.modify(orb, [['connectionCacheMaximum', 252], ['noLocalCopies',  'true']])
```

Table 445. AdminConfig modify command description. The previous commands consist of the following elements:

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object representing the WebSphere Application Server configuration
`modify`	AdminConfig command
`orb`	Evaluates to the ID of ORB
`connectionCacheMaximum`	Attribute
`252`	Value of the connectionCacheMaximum attribute
`noLocalCopies`	Attribute
`true`	Value of the noLocalCopies attribute

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Configuring processes using scripting

You can use the wsadmin tool to configure processes in your application server configuration. Enhance the operation of an application server by defining command-line information for starting or initializing the application server process. Process definition settings define runtime properties such as the program to run, arguments to run the program, and the working directory.

About this task

There are three ways to perform this task. Complete the steps in this task to use the setProcessDefinition command for the AdminTask object or the AdminConfig object to modify your process definition configuration. Alternatively, you can use the configureProcessDefinition Jython script in the AdminServerManagement script library to configure process definition attributes. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure process definition settings using the configureProcessDefinition script:

```
AdminServerManagement.configureProcessDefintion(nodeName, serverName, otherParamList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

Procedure

1. Start the wsadmin scripting tool.
2. Use the setProcessDefinition command for the AdminTask object or the AdminConfig object to modify your process definition configuration.
   - Use the following example to configure the process definition with the setProcessDefinition command for the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask setProcessDefinition {-interactive}
       ```

     - Using Jython:

       ```
       AdminTask.setProcessDefinition (['-interactive'])
       ```

   - Use the following steps to configure the process definition with the AdminConfig option:
     a. Identify the server and assign it to the `s1` variable, as the following example demonstrates:
        - Using Jacl:

          ```
          set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
          ```

        - Using Jython:

          ```
          s1 = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
          print s1
          ```

Table 446. AdminConfig getid command description. The previous commands consist of the following elements:

Element	Description
`set`	Jacl command
`s1`	Variable name
`$`	Jacl operator for substituting a variable name with its value
`AdminConfig`	Object that represents the WebSphere Application Server configuration
`getid`	AdminConfig command
`Cell`	Object type
`mycell`	Name of the object that will be modified
`Node`	Object type

Table 446. AdminConfig getid command description (continued). The previous commands consist of the following elements:

mynode	Name of the object that will be modified
Server	Object type
server1	Name of the object that will be modified
print	Jython command

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

b. Identify the process definition for the server of interest, and assign it to the `processDef` variable, as the following example displays:

– Using Jacl:

```
set processDef [$AdminConfig list JavaProcessDef $s1]
set processDef [$AdminConfig showAttribute $s1 processDefinitions]
```

– Using Jython:

```
processDef = AdminConfig.list('JavaProcessDef', s1)
print processDef
processDef = AdminConfig.showAttribute(s1, 'processDefinitions')
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#JavaProcessDef_1)
```

c. Modify the configuration attributes for the process definition.

The following example changes the working directory:

– Using Jacl:

```
$AdminConfig modify $processDef {{workingDirectory /home/myProfile/temp/user1}}
```

– Using Jython:

```
AdminConfig.modify(processDef, [['workingDirectory', '/home/myProfile/temp/user1']])
```

The following example modifies the name of the stderr file:

– Using Jacl:

```
set errFile [list stderrFilename \${LOG_ROOT}/server1/new_stderr.log]
set attr [list $errFile]
$AdminConfig modify $processDef [subst {{ioRedirect {$attr}}}]
```

– Using Jython:

```
errFile = ['stderrFilename', '\${LOG_ROOT}/server1/new_stderr.log']
attr = [errFile]
AdminConfig.modify(processDef, [['ioRedirect', [attr]]])
```

The following example modifies the process priority level:

– Using Jacl:

```
$AdminConfig modify $processDef {{execution {{processPriority 15}}}}
```

– Using Jython:

```
AdminConfig.modify(processDef, [['execution', [['processPriority', 15]]]])
```

The following example changes the maximum number of times the product tries to start an application server in response to a start request. If the server cannot be started within the specified number of attempts, an error message is issued that indicates that the application server could not be started.

– Using Jacl:

```
$AdminConfig modify $processDef {{monitoringPolicy {{maximumStartupAttempts 1}}}}
```

– Using Jython:

```
AdminConfig.modify(processDef, [['monitoringPolicy', [['maximumStartupAttempts',  1]]]])
```

3. Save the configuration changes.

Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Configuring the runtime transaction service using scripting

Use the wsadmin tool to configure transaction properties for servers. The transaction service is a server runtime component that coordinates updates to multiple resource managers to ensure atomic updates of data. Transactions are started and ended by applications or the container in which the applications are deployed.

About this task

There are two ways to perform this task. Use the steps in this task to use the AdminControl object to modify your transaction service configuration. Alternatively, you can use the configureTransactionService Jython script in the AdminServerManagement script library to configure the transaction service configuration attributes. You can use the configureRuntimeTransactionService to update the transaction service MBean attributes. The wsadmin tool automatically loads the scripts when the tool starts.

Use the following syntax to configure transaction service settings using the configureTransactionService script:

```
AdminServerManagement.configureTransactionService(nodeName, serverName, totalTranLifetimeTimeout, clientInactivityTimeout,
  maximumTransactionTimeout, heuristicRetryLimit, heuristicRetryWait, propogateOrBMTTranLifetimeTimeout, asyncResponseTimeout,
  otherAttributeList)
```

Use the following syntax to configure runtime transaction service settings using the configureRuntimeTransactionService script:

```
AdminServerManagement.configureRuntimeTransactionService(nodeName, serverName, totalTranLifetimeTimeout,
  clientInactivityTimeout)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

Procedure

1. Identify the transaction service MBean for the application server.

   Use the completeObjectName command for the AdminControl object to return the transaction service MBean for the *server1* server, and to set it to the `ts` variable, as the following example demonstrates:

   - Using Jacl:

```
set ts [$AdminControl completeObjectName cell=mycell,node=mynode,
process=server1,type=TransactionService,*]
```

   - Using Jython:

```
ts = AdminControl.completeObjectName('cell=mycell,node=mynode,
process=server1,type=TransactionService,*')
print ts
```

Table 447. Elements in the completeObjectName command. This tables describes the elements used to return the transaction service MBean for the server in the previous example commands.

Element	Description
`set`	A Jacl command
`ts`	A variable name
`$`	A Jacl operator for substituting a variable name with its value
`AdminControl`	An object that enables the manipulation of MBeans running in a server process
`completeObjectName`	An AdminControl command

Table 447. Elements in the completeObjectName command (continued). This tables describes the elements used to return the transaction service MBean for the server in the previous example commands.

Element	Description
`cell=mycell,node=mynode, process=server1,type=TransactionService`	A fragment of the object name whose complete name is returned by this command. It is used to find the matching object name which is, in this case, the transaction object MBean for the node *mynode*, where *mynode* is the name of the node that you use to synchronize configuration changes. For example: `type=TransactionService, process=server1`. It can be any valid combination of domain and key properties, for example, type, name, cell, node, and process.

Example output:

```
WebSphere:cell=mycell,name=TransactionService,mbeanIdentifier=TransactionService,
type=TransactionService,node=mynode,process=server1
```

2. Modify the runtime transaction service configuration attributes.
   - Using Jacl:

```
$AdminControl setAttributes $ts {{clientInactivityTimeout 30}
{totalTranLifetimeTimeout 180}}
```

   - Using Jython:

```
AdminControl.setAttributes(ts, [['clientInactivityTimeout', 30],
['totalTranLifetimeTimeout', 180]])
```

The following table shows the elements in the previous commands.

Table 448. Elements in the setAttributes command. This tables describes the elements in the setAttributes command.

Element	Description
`$`	Jacl operator for substituting a variable name with its value
`AdminControl`	An object that enables the manipulation of MBeans running in a server process
`setAttributes`	An AdminControl command
`ts`	Evaluates to the ID of the transaction service of interest
`clientInactivityTimeout`	An attribute
`30`	The value of the clientInactivityTimeout attribute specified in seconds. A value of `0` means that there is no timeout limit.
`totalTranLifetimeTimeout`	An attribute
`180`	The value of the totalTranLifetimeTimeout attribute specified in seconds. A value of `0` means that there is no timeout limit.

Configuring the WS-Transaction specification level by using wsadmin scripting

You can configure the default WS-Transaction specification level to use for outbound requests that include a Web Services Atomic Transaction (WS-AT) or Web Services Business Activity (WS-BA) coordination context.

About this task

The product supports the WS-Transaction 1.0, the WS-Transaction 1.1 and the WS-Transaction 1.2 specifications, but when an outbound request is sent, only one specification level can be used. The default

WS-Transaction specification level is used if the specification level that the server requires cannot be determined from the provider policy (the WS-Transaction WS-Policy assertion). This situation might occur when the policy assertion is not available either from the WS-Transaction policy type of the client or from the WSDL of the target web service. Also, this situation might occur when the policy assertion is available, but the client and the target web service support both specification levels.

For details of the specifications, see the topics about Web Services Atomic Transaction support and Web Services Business Activity support in the application server.

You can set the default WS-Transaction specification level by using wsadmin scripting, as described in this task, or by using the administrative console and configuring the relevant transaction property for the application server.

Procedure

1. Start the wsadmin scripting client if it is not already running.
2. Retrieve the configuration ID of the transaction service. In Jacl, use the following code example:

```
set txService $AdminConfig list TransactionService
```

   In Jython, use the following code example:

```
txService = AdminConfig.list("TransactionService")
```

3. Modify the WSTransactionSpecificationLevel attribute to the value you require. In Jacl, to configure the server to use WS-Transaction 1.1, use the following code example:

```
$AdminConfig modify $txService {{WSTransactionSpecificationLevel WSTX_11}}
```

   In Jython, to configure the server to use WS-Transaction 1.0, use the following code example:

```
AdminConfig.modify ($txService,[["WSTransactionSpecificationLevel", "WSTX_10"]])
```

4. Save the configuration changes with the wsadmin tool.

Results

You have configured the default WS-Transaction specification level for the server.

Setting port numbers to the serverindex.xml file using scripting

You can use the wsadmin tool to modify the port numbers specified in the serverindex.xml file. The endpoints of the serverindex.xml file are part of different objects in the configuration.

About this task

There are multiple ways to complete this task. Complete the steps in this task to use the AdminConfig and AdminTask objects to configure ports in your environment. Alternatively, you can use the scripts in the AdminServerManagement script library to configure various ports in your configuration.

Before modifying ports using scripting, you must launch the wsadmin tool.

Procedure

- Modify the BOOTSTRAP_ADDRESS attribute of the server1 process.

  The BOOTSTRAP_ADDRESS attribute is an attribute of the NameServer object that exists inside the server. The naming client uses the attribute to specify the naming server to look up the initial context. The following examples demonstrate how to modify the BOOTSTRAP_ADDRESS attribute:

  – Using the AdminTask object:

    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName BOOTSTRAP_ADDRESS
-host myhost -port 2810}
```

- Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName
BOOTSTRAP_ADDRESS -host myhost -port 2810]')
```

– Using the AdminConfig object. To modify its endpoint, obtain the ID of the NameServer object and issue a modify command, as the following example demonstrates:

- Using Jacl:

```
set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
set ns [$AdminConfig list NameServer $s]
$AdminConfig modify $ns {{BOOTSTRAP_ADDRESS {{port 2810} {host myhost}}}}
```

- Using Jython:

```
s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
ns = AdminConfig.list('NameServer', s)
AdminConfig.modify(ns, [['BOOTSTRAP_ADDRESS', [['host', 'myhost'],  ['port', 2810]]]])
```

• Modify the SOAP_CONNECTOR-ADDRESS attribute of the server1 process.

The SOAP_CONNECTOR-ADDRESS attribute is an attribute of the SOAPConnector object that exists inside the server. HTTP transport uses the SOAP connector port for incoming SOAP requests. Use the following examples modify the SOAP_CONNECTOR-ADDRESS attribute:

– Using the AdminTask object:

- Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName SOAP_CONNECTOR_ADDRESS
-host myhost -port 8881}
```

- Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName SOAP_CONNECTOR_ADDRESS
-host myhost -port 8881]')
```

– To use the AdminConfig object to modify its endpoint, obtain the ID of the SOAPConnector object and issue a **modify** command, as the following example demonstrates:

- Using Jacl:

```
set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
set soap [$AdminConfig list SOAPConnector $s]
$AdminConfig modify $soap {{SOAP_CONNECTOR_ADDRESS {{host myhost} {port 8881}}}}
```

- Using Jython:

```
s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
soap = AdminConfig.list('SOAPConnector', s)
AdminConfig.modify(soap, [['SOAP_CONNECTOR_ADDRESS', [['host', 'myhost'],  ['port', 8881]]]])
```

• Modify the JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS attributes of the server1 process.

The JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS attributes are attributes of the JMSServer object that exists inside the server. The system uses these ports to configure the JMS provider topic connection factory settings. The following examples modify the JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS attributes:

– Using the AdminTask object:

- Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName JMSSERVER_QUEUED_ADDRESS
-host myhost -port 5560}

$AdminTask modifyServerPort server1 {-nodeName mynode -endPointName JMSSERVER_DIRECT_ADDRESS
-host myhost -port 5561}
```

- Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName JMSSERVER_QUEUED_ADDRESS
-host myhost -port 5560]')

AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName JMSSERVER_DIRECT_ADDRESS
-host myhost -port 5561]')
```

– Using the AdminConfig object. To modify its endpoint, obtain the ID of the JMSServer object and issue a **modify** command, for example:

- Using Jacl:

```
set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
set jmss [$AdminConfig list JMSServer $s]
$AdminConfig modify $jmss {{JMSSERVER_QUEUED_ADDRESS {{host myhost} {port 5560}}}}
$AdminConfig modify $jmss {{JMSSERVER_DIRECT_ADDRESS {{host myhost} {port 5561}}}}
```

- Using Jython:

```
s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
jmss = AdminConfig.list('JMSServer', s)
AdminConfig.modify(jmss, [['JMSSERVER_QUEUED_ADDRESS', [['host', 'myhost'], ['port', 5560]]]])
AdminConfig.modify(jmss, [['JMSSERVER_DIRECT_ADDRESS', [['host', 'myhost'], ['port', 5561]]]])
```

- Modify the NODE_DISCOVERY_ADDRESS attribute of node agent process. The NODE_DISCOVERY_ADDRESS attribute is an attribute of the NodeAgent object that exists inside the server. The system uses this port to receive the incoming process discovery messages inside a node agent process. The following examples modify the NODE_DISCOVERY_ADDRESS attribute:
  - Using the AdminTask object:
    - Using Jacl:

```
$AdminTask modifyServerPort nodeagent {-nodeName mynode -endPointName
NODE_DISCOVERY_ADDRESS -host myhost -port 7272}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('nodeagent', '[-nodeName mynode -endPointName
NODE_DISCOVERY_ADDRESS -host myhost -port 7272]')
```

  - Using the AdminConfig object. To modify its endpoint, obtain the ID of the NodeAgent object and issue a **modify** command, for example:
    - Using Jacl:

```
set nodeAgentServer [$AdminConfig getid /Cell:mycell/Node:mynode/Server:nodeagent/]
set nodeAgent [$AdminConfig list NodeAgent $nodeAgentServer]
$AdminConfig modify $nodeAgent {{NODE_DISCOVERY_ADDRESS {{host myhost} {port 7272}}}}
```

    - Using Jython:

```
nodeAgentServer = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:nodeagent/')
nodeAgent = AdminConfig.list('NodeAgent', nodeAgentServer)
AdminConfig.modify(nodeAgent, [['NODE_DISCOVERY_ADDRESS', [['host', 'myhost'], ['port', 7272]]]])
```

- Modify the CELL_DISCOVERY_ADDRESS attribute of deployment manager process. The CELL_DISCOVERY_ADDRESS attribute is an attribute of the deploymentManager object that exists inside the server. The system uses this port to receive the incoming process discovery messages inside a deployment manager process. Use the following examples modify the CELL_DISCOVERY_ADDRESS attribute:
  - Using the AdminTask object:
    - Using Jacl:

```
$AdminTask modifyServerPort dmgr {-nodeName managernode -endPointName
CELL_MULTICAST_DISCOVERY_ADDRESS -host myhost -port 7272}

$AdminTask modifyServerPort dmgr {-nodeName managernode -endPointName
CELL_DISCOVERY_ADDRESS -host myhost -port 7278}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('dmgr', '[-nodeName managernode -endPointName
CELL_MULTICAST_DISCOVERY_ADDRESS -host myhost -port 7272]')

AdminTask.modifyServerPort ('dmgr', '[-nodeName managernode -endPointName
CELL_DISCOVERY_ADDRESS -host myhost -port 7278]')
```

  - To use the AdminConfig attribute to modify its endpoint, obtain the ID of the deploymentManager object and issue a modify command, for example:
    - Using Jacl:

```
set netmgr [$AdminConfig getid /Cell:mycell/Node:managernode/Server:dmgr/]
set deploymentManager [$AdminConfig list CellManager $netmgr]
$AdminConfig modify $deploymentManager {{CELL_MULTICAST_DISCOVERY_ADDRESS {{host myhost} {port 7272}}}}
$AdminConfig modify $deploymentManager {{CELL_DISCOVERY_ADDRESS {{host myhost} {port 7278}}}}
```

    - Using Jython:

```
netmgr = AdminConfig.getid('/Cell:mycell/Node:managernode/Server:dmgr/')
deploymentManager = AdminConfig.list('CellManager', netmgr)
AdminConfig.modify(deploymentManager, [['CELL_MULTICAST_DISCOVERY_ADDRESS', [['host', 'myhost'],
['port', 7272]]]])
AdminConfig.modify(deploymentManager, [['CELL_DISCOVERY_ADDRESS', [['host', 'myhost'], ['port', 7278]]]])
```

- Modify the WC_defaulthost attribute of the server1 process. Use the following examples modify WC_defaulthost endpoint:
  - Using the AdminConfig object:
    - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_defaulthost"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 5555]]
           break
        }
     }
  }
}
```

    - Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_defaulthost":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5555]])
           break
```

- Modify the WC_defaulthost_secure attribute of the server1 process. Use the following examples to modify a WC_defaulthost_secure endpoint:
  - Using the AdminTask object:
    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_defaulthost_secure
-host myhost -port 5544}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_defaulthost_secure
-host myhost -port 5544]')
```

  - Using the AdminConfig object:
    - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_defaulthost_secure"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 5544]]
```

```
            break
         }
      }
   }
}
```

- Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_defaulthost_secure":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5544]])
           break
```

- Modify the WC_adminhost attribute of the server1 process.

  To modify a WC_adminhost endpoint, use one of the following examples:

  – Using the AdminTask object:

    - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_adminhost -host myhost -port 6666}
```

    - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_adminhost -host myhost
 -port 6666]')
```

  – Using the AdminConfig object:

    - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_adminhost"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 6666]]
           break
        }
     }
  }
}
```

    - Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries =
 AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_adminhost":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 6666]])
           break
```

- Modify the WC_adminhost_secure attribute of server1 process.

To modify a WC_adminhost_secure endpoint, use one of the following examples:

– Using the AdminTask object:

  - Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_adminhost_secure -host
 myhost -port 5566}
```

  - Using Jython:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_adminhost_secure
-host myhost -port 5566]')
```

– Using the AdminConfig object:

  - Using Jacl:

```
set serverName server1
set node [$AdminConfig getid /Node:myNode/]
set serverEntries [$AdminConfig list ServerEntry $node]

foreach serverEntry $serverEntries {
  set sName [$AdminConfig showAttribute $serverEntry serverName]
  if {$sName == $serverName} {
     set specialEndPoints [lindex [$AdminConfig showAttribute $serverEntry specialEndpoints] 0]
     foreach specialEndPoint $specialEndPoints {
        set endPointNm [$AdminConfig showAttribute $specialEndPoint endPointName]
        if {$endPointNm == "WC_adminhost_secure"} {
           set ePoint [$AdminConfig showAttribute $specialEndPoint endPoint]
           $AdminConfig modify $ePoint [list [list host myhost] [list port 5566]]
           break
        }
     }
  }
}
```

  - Using Jython:

```
serverName = "server1"
node = AdminConfig.getid('/Node:myNode/')
serverEntries = AdminConfig.list('ServerEntry', node).split(java.lang.System.getProperty('line.separator'))

for serverEntry in serverEntries:
  sName = AdminConfig.showAttribute(serverEntry, "serverName")
  if sName == serverName:
     sepString = AdminConfig.showAttribute(serverEntry, "specialEndpoints")
     sepList = sepString[1:len(sepString)-1].split(" ")
     for specialEndPoint in sepList:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_adminhost_secure":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5566]])
           break
```

What to do next

Save the configuration changes.

Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Use the syncActiveNode or syncNode scripts in the AdminNodeManagement script library to propagate the configuration changes to node or nodes.

- Use the syncActiveNodes script to propagate the changes to each node in the cell, as the following example demonstrates:

  ```
  AdminNodeManagement.syncActiveNodes()
  ```

- Use the syncNode script to propagate the changes to a specific node, as the following example demonstrates:

  ```
  AdminNodeManagement.syncNode("myNode")
  ```

Disabling components using scripting

You can disable components by invoking operations with scripting and the wsadmin tool. This topic describes how to disable the nameServer component of a configured server. You can modify the examples in this topic to disable other components.

About this task

There are two ways to complete this task. This topic uses the AdminConfig object to stop components in your environment. Alternatively, you can use the configureStateManageable script in the AdminServerManagement script library to enable and disable components. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure PMI settings using the configureStateManageable script:

```
AdminServerManagement.configureStateManageable(nodeName, serverName, parentType, initialState)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the server component and assign it to the nameServer variable.
   - Using Jacl:

```
set nameServer [$AdminConfig list NameServer $server]
```

   - Using Jython:

```
nameServer = AdminConfig.list('NameServer', server)
print nameServer
```

   Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#NameServer_1)
```

3. List the components belonging to the server.

   List the components that are associated with the server, and assign the components to the components variable, as the following example demonstrates:

   - Using Jacl:

```
set components [$AdminConfig list Component $server]
```

   - Using Jython:

```
components = AdminConfig.list('Component', server)
print components
```

   The components variable contains a list of components.

   Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#ApplicationServer_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#EJBContainer_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#NameServer_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#WebContainer_1)
```

4. Identify the nameServer component.

   Parse the components to identify the nameServer component, and assign it to the nameServer variable. Since the name server component is the third element in the list, retrieve this element by using an index of 2, as the following example demonstrates:

   - Using Jacl:

```
set nameServer [lindex $components 2]
```

   - Using Jython:

```
# get line separator
import  java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayComponents = components.split(lineSeparator)
nameServer = arrayComponents[2]
print nameServer
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#NameServer_1)
```

5. Disable the nameServer component.

   Modify the nested initialState attribute belonging to the stateManagement attribute to disable the nameServer component, as the following example demonstrates:

   - Using Jacl:

```
$AdminConfig modify $nameServer {{stateManagement {{initialState STOP}}}}
```

   - Using Jython:

```
AdminConfig.modify(nameServer, [['stateManagement', [['initialState', 'STOP']]]])
```

6. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Disabling the trace service using scripting

You can disable the services of a configured server with scripting and the wsadmin tool.

About this task

Perform the following steps to disable the trace service of a configured server. You can modify this example to disable a different service.

Procedure

1. Start the wsadmin scripting tool.
2. Identify the server and assign it to the server variable. For example:

   - Using Jacl:

```
set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
```

   - Using Jython:

```
server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
print server
```

   Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

3. List all the services belonging to the server and assign them to the services variable. The following example returns a list of services:

   - Using Jacl:

```
set services [$AdminConfig list Service $server]
```

   - Using Jython:

```
services = AdminConfig.list('Service', server)
print services
```

   Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#AdminService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#DynamicCache_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#MessageListenerService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#ObjectRequestBroker_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#PMIService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#RASLoggingService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#SessionManager_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#TraceService_1)
(cells/mycell/nodes/mynode/servers/server1|server.xml#TransactionService_1)
```

4. Identify the trace service and assign it to the traceService variable.

Since trace service is the seventh element in the list, retrieve this element by using index 6.

- Using Jacl:

```
set traceService [$AdminConfig list TraceService $server]
```

- Using Jython:

```
traceService = AdminConfig.list('TraceService', server)
print traceService
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#TraceService_1)
```

5. Disable the trace service by modifying the enable attribute. For example:
   - Using Jacl:

```
$AdminConfig modify $traceService {{enable false}}
```

   - Using Jython:

```
AdminConfig.modify(traceService, [['enable', 'false']])
```

6. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Configuring servlet caching using wsadmin scripting

You can configure servlet caching with scripting and the wsadmin tool. The dynamic cache service works within an application server Java virtual machine (JVM), intercepting calls to cacheable objects. For example, it intercepts calls through a servlet service method or a command execute method, and either stores the output of the object to the cache or serves the content of the object from the dynamic cache.

Before you begin

Before you can configure servlet caching, you must configure dynamic cache. Use the configureDynamicCache Jython script in the AdminServerManagement script library to configure dynamic caching. The wsadmin tool automatically loads the script when the tool starts. Use the following syntax to configure dynamic caching using the configureDynamicCache script:

```
AdminServerManagement.configureDynamicCache(nodeName, serverName, defaultPriority, cacheSize,
   externalCacheGroupName, externalCacheGroupType, otherAttributeList)
```

For additional information and argument definitions, see the documentation for the AdminServerMananagment script library.

About this task

After a servlet is invoked and completes generating the output to cache, a cache entry is created containing the output and the side effects of the servlet. These side effects can include calls to other servlets or JavaServer Pages (JSP) files or metadata about the entry, including timeout and entry priority information. Configure servlet caching to save the output of servlets and JavaServer Pages (JSP) files to the dynamic cache.

Note: If you use the wsadmin tool to enable servlet caching, verify that portlet fragment caching is also enabled. Similarly, if you use the wsadmin tool to disable servlet caching, verify that portlet fragment caching is also disabled. The settings for these two caching functions must remain synchronized. If you enable or disable servlet caching using the administrative console, synchronization performed automatically.

To see a list of parameters associated with dynamic caching, use the **attributes** command. For example:

```
$AdminConfig attributes DynamicCache
```

Procedure

1. Start the wsadmin scripting tool.
2. Retrieve the configuration ID of the server object.

   The following example sets the first server found to the `s1` variable:

   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('/Server:server1/')
     ```

3. Retrieve the web containers for the server of interest. and assign them to the `wc` variable.

   The following example sets the web container to the `wc` variable:

   - Using Jacl:

     ```
     set wc [$AdminConfig list WebContainer $s1]
     ```

   - Using Jython:

     ```
     wc = AdminConfig.list('WebContainer', s1)
     ```

4. Set a variable with the new value for the enableServletCaching attribute.

   Set the enableServletCaching attribute to `true` and assign it to the `serEnable` variable, as the following example demonstrates:

   - Using Jacl:

     ```
     set serEnable "{enableServletCaching true}"
     ```

   - Using Jython:

     ```
     serEnable = [['enableServletCaching', 'true']]
     ```

5. Enable dynamic caching.

   Use the AdminConfig object to modify the application server configuration, as the following example demonstrates:

   - Using Jacl:

     ```
     $AdminConfig modify $wc $serEnable
     ```

   - Using Jython:

     ```
     AdminConfig.modify(wc, serEnable)
     ```

Modifying variables using wsadmin scripting

Use scripting and the wsadmin tool to modify variables in the application server.

About this task

Complete the following steps to modify an application server variable:

Procedure

1. Start the wsadmin scripting tool.
2. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask setVariable {-interactive}
       ```

     - Using Jython:

       ```
       AdminTask.setVariable (['-interactive'])
       ```

   - Using the AdminConfig object. The following examples modify the DB2_JDBC_DRIVER_PATH variable on the node level:
     - Using Jacl:

```
set varName DB2_JDBC_DRIVER_PATH
set newVarValue C:/SQLLIB/java

set node [$AdminConfig getid /Node:myNode/]

set varSubstitutions [$AdminConfig list VariableSubstitutionEntry $node]

foreach varSubst $varSubstitutions {
   set getVarName [$AdminConfig showAttribute $varSubst symbolicName]
   if {[string compare $getVarName $varName] == 0} {
      $AdminConfig modify $varSubst [list [list value $newVarValue]]
      break
   }
}
```

– Using Jython:

```
varName = "DB2_JDBC_DRIVER_PATH"
newVarValue = "C:/SQLLIB/java"

node = AdminConfig.getid("/Node:myNode/")

varSubstitutions = AdminConfig.list("VariableSubstitutionEntry",node).split
(java.lang.System.getProperty("line.separator"))

for varSubst in varSubstitutions:
   getVarName = AdminConfig.showAttribute(varSubst, "symbolicName")
   if getVarName == varName:
      AdminConfig.modify(varSubst,[["value", newVarValue]])
      break
```

3. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Increasing the Java virtual machine heap size using scripting

Some servers might specify a Java virtual machine (JVM) heap size greater than the default. You can increase the heap size of the JVM using the administrative console, the wsadmin tool, or a Java client.

Procedure

1. Set attributes that control the heap size for the JVM that is associated with the server.

   Use the administrative console or the wsadmin tool to control the heap size.

2. Increase the heap size of the JVM.

   Use the AdminTask object to modify the heap size, as the following examples demonstrate:

   - Using Jython:

     ```
     AdminTask.setJVMMaxHeapSize('-serverName server1 -nodeName node1 -maximumHeapSize heap_size')
     ```

   - Using Jacl:

     ```
     $AdminTask setJVMMaxHeapSize {-serverName server1 -nodeName node1 -maximumHeapSize heap_size}
     ```

PortManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure servers with the wsadmin tool. The commands and parameters in the PortManagement group can be used to list and modify application and server ports.

The PortManagement command group for the AdminTask object includes the following commands:

- “listApplicationPorts” on page 471
- “listServerPorts” on page 471

• “modifyServerPort” on page 472

listApplicationPorts

Use the **listApplicationPorts** command to list the ports in order to access a particular application.

Target object

The application name for which the list of ports is generated. (String)

Required parameters

None.

Return values

The ports that are used by the application that you specified.

Batch mode example usage

- Using Jacl:
  ```
  $AdminTask listApplicationPorts {}
  ```
- Using Jython string:
  ```
  AdminTask.listApplicationPorts ()
  ```

Interactive mode example usage

- Using Jacl:
  ```
  $AdminTask listApplicationPorts {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listApplicationPorts ('[-interactive]')
  ```

listServerPorts

Use the **listServerPorts** command to list the ports that are used by the server that you specify.

Target object

The server name. (String)

Optional parameters

-nodeName
The name of the node. This parameter is only required when the server name is not unique in the cell. (String, optional)

Batch mode example usage

- Using Jacl:
  ```
  $AdminTask listServerPorts server1 {-nodeName myNode}
  ```
- Using Jython string:
  ```
  AdminTask.listServerPorts ('server1', '[-nodeName myNode]')
  ```

Interactive mode example usage

- Using Jacl:
  ```
  $AdminTask listServerPorts {-interactive}
  ```

- Using Jython string:

```
AdminTask.listServerPorts ('[-interactive]')
```

modifyServerPort

Use the **modifyServer Port** command to modify the port that is used by the server.

Target object

The name of the server for which the port is modified.

Required parameters

-nodeName
The name of the server node. This parameter is required only if the server name is not unique in the cell. (String, required)

-endPointName
The name of the port to modify. (String, required)

Optional parameters

-host
The new value for the host name of the endpoint. (String, optional)

-port
The new value for the port number of the endpoint. (Integer, optional)

-modifyShared
Set this parameter to `true` to modify the port of interest if the port is shared between multiple transport channel chains. If this parameter is not specified, the command will not modify the port if it is used in more than one transport channel chain. (Boolean, optional)

Batch mode example usage

- Using Jacl:

```
$AdminTask modifyServerPort server1 {-nodeName myNode -endPointName
port1 -port 5566 -modifyShared true}
```

- Using Jython string:

```
AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName port1
-port 5566 -modifyShared true]')
```

- Using Jython list:

```
AdminTask.modifyServerPort ('server1', ['-nodeName', 'myNode', '-endPointName',
'port1', '-port', '5566 -modifyShared true'])
```

Interactive mode example usage

- Using Jacl:

```
$AdminTask modifyServerPort {-interactive}
```

- Using Jython string:

```
AdminTask.modifyServerPort ('[-interactive]')
```

- Using Jython list:

```
AdminTask.modifyServerPort (['-interactive'])
```

DynamicCache command group for the AdminTask object

You can use the Jython scripting language to manage the dynamic cache service. Use the commands in the DynamicCache command group to create object and service cache instances with the wsadmin tool.

Use the following commands to manage your dynamic cache configuration:

- "createObjectCacheInstance"
- "createServletCacheInstance"

createObjectCacheInstance

The **createObjectCacheInstance** command creates an object cache instance in your configuration. An object cache is a location where the dynamic cache stores, distributes, and shares data.

Target object

Specify the instance of the cache provider that the system stores the object cache instance objects under.

Required parameters

-name
Specifies the name of the object cache instance to create. (String, required)

-jndiName
Specifies the Java Naming and Directory Interface (JNDI) name for the object cache instance to create. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createObjectCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml#
#CacheProvider_1055745612404)','[-name objectName -jndiName myJNDI]')
```

- Using Jython list:

```
AdminTask.createObjectCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml#
#CacheProvider_1055745612404)',['-name', 'objectName', '-jndiName', 'myJNDI'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createObjectCacheInstance('-interactive')
```

createServletCacheInstance

The **createServletCacheInstance** command creates a servlet cache instance in your configuration. A servlet cache instance specifies the location where the dynamic cache stores, distributes, and shares data.

Target object

Specify the instance of the cache provider that the system stores the object cache instance objects under.

Required parameters

-name
Specifies the name of the servlet cache instance to create. (String, required)

-jndiName
Specifies the Java Naming and Directory Interface (JNDI) name for the servlet cache instance to create. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createServletCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml
l#CacheProvider_1055745612404)','[-name servletName -jndiName myJNDI]')
```

- Using Jython list:

```
AdminTask.createServletCacheInstance('CacheProvider(cells/myCell/nodes/myNode/servers/myServer|resources-pme502.xml
l#CacheProvider_1055745612404)',['-name', 'servletName', '-jndiName', 'myJNDI'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createServletCacheInstance('-interactive')
```

VariableConfiguration command group for the AdminTask object

You can use the Jython or Jacl scripting languages to configure servers with the wsadmin tool. The commands and parameters in the VariableConfiguration group can be used to remove variable definitions from the system, to set values for variables, or to query for variable values with a specific scope.

The VariableConfiguration command group for the AdminTask object includes the following commands:

- "removeVariable"
- "setVariable" on page 475
- "showVariables" on page 476

removeVariable

Use the **remove Variable** command to remove a variable definition from the system. A variable is a configuration property that you can use to provide a parameter for some values in the system.

Target object

None

Parameters and return values

-variableName
The name of the variable. (String, required)

-scope
The scope of the variable definition. The default is Cell. (String, optional)

The syntax of the scope parameter is `Type=value`

Supported types are Cell, Node, Servers, Application and Cluster, for example:

- `Node=node1`
- `Node=node1, Server=server1`
- `Application=app1`
- `Cluster=cluster1`
- `Cell=cell1`

-node
The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeVariable {-interactive}
```

- Using Jython string:

```
AdminTask.removeVariable ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeVariable (['-interactive'])
```

setVariable

Use the **set Variable** command to set the value for a variable. A variable is a configuration property that you can use to provide a parameter for some values in the system.

Target object

None

Parameters and return values

-variableName
The name of the variable. (String, required)

-scope
The scope of the variable definition. The default is Cell. (String, optional)

The syntax of the scope parameter is `Type=value`, for example:

- `Node=node1`
- `Node=node1, Server=server1`
- `Application=app1`
- `Cluster=cluster1`
- `Cell=cell1`

.

-node
The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional)

-variableValue
The value of the variable. (String, optional)

-variableDescription
The description of the variable. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setVariable {-variableName varname1 -scope Cell=localhost Node01Cell,Node=
localhostNode01}
```

- Using Jython string:

  ```
  AdminTask.setVariable(' [-variableName varname1 -scope Cell=local hostNode01Cell,Node=
  localhostNode01]')
  ```

- Using Jython list:

  ```
  AdminTask.setVariable (['-variableName', 'varname1', '-scope', 'Cell=localhostNode 01Cell,Node=
  local hostNode01'])
  ```

Interactive mode example usage:

- Using Jacl:

  ```
  $AdminTask setVariable {-interactive}
  ```

- Using Jython string:

  ```
  AdminTask.setVariable ('[-interactive]')
  ```

- Using Jython list:

  ```
  AdminTask.setVariable (['-interactive'])
  ```

showVariables

Use the **show Variables** command to list variable values under a scope.

Target object

None

Parameters and return values

- scope
The scope of the variable definition. The default is Cell. (String, optional)

The syntax of the scope parameter is `Type=value`

Supported types are Cell, Node, Servers, Application and Cluster, for example:

- `Node=node1`
- `Node=node1, Server=server1`
- `Application=app1`
- `Cluster=cluster1`
- `Cell=cell1`

-node
The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional)

-variableName
The name of the variable. If you specify this parameter, the value of this variable is returned. If you do not specify this parameter, all variables defined under the scope will return in list format where each element is a variable name and value pair. (String, optional)

Examples

Interactive mode example usage:

- Using Jacl:

  ```
  $AdminTask showVariables {-interactive}
  ```

- Using Jython string:

  ```
  AdminTask.showVariables ('[-interactive]')
  ```

- Using Jython list:

```
AdminTask.showVariables (['-interactive'])
```

Chapter 16. Setting up intermediary services using scripting

Use the wsadmin tool and the Jython scripting language to configure intermediary services, such as web servers, and proxy servers.

Procedure

- Regenerate the node plug-in configuration. You can use scripting and the wsadmin tool to regenerate the plug-in configuration for a web server.
- Create virtual hosts using templates. Use scripting to create a new virtual host from a new or preexisting template. Virtual hosts let you manage a single application server on a single machine as if the application server were multiple application servers each on their own host machine.

Regenerating the node plug-in configuration using scripting

You can use scripting and the wsadmin tool to regenerate the node plug-in configuration.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 article for more information.

About this task

Perform the following steps to regenerate the node plug-in configuration:

Procedure

1. Identify the plug-in and assign it to the `generator` variable, for example:

   Using Jython:

   ```
   generator = AdminControl.completeObjectName('type=PluginCfgGenerator,*')
   ```

   Using Jacl:

   ```
   set generator [$AdminControl completeObjectName type=PluginCfgGenerator,*]
   ```

   Additionally, you can specify the optional **node** parameter. In a stand alone environment, specify the node of the application server.
2. Regenerate the node plug-in for a given web server definition.

   Using Jython:

   ```
   AdminControl.invoke(generator, 'generate', "profile_root/config mycell myWebServerNode myWebServerName true true")
   ```

   Using Jacl:

   ```
   $AdminControl invoke $generator generate "profile_root/config mycell myWebServerNode myWebServerName true true"
   ```

Example

The following application-centric examples use the **generate**, **propagate**, and **propagateKeyring** operations for a given web server definition:

Using Jython:

```
AdminControl.invoke(generator,'generate',"profile_root/config
01Cell03  01Node03 webserver1 true")

AdminControl.invoke(generator,'propagate',"profile_root/config
01Cell03  01Node03 webserver1")

AdminControl.invoke(generator,'propagateKeyring',"profile_root/config
01Cell03  01Node03 webserver1"
```

Using Jacl:

```
$AdminControl invoke $generator generate "profile_root/config  01Cell03  01Node03 webserver1 true"

$AdminControl invoke $generator propagate  "profile_root/config  01Cell03  01Node03 webserver1"

$AdminControl invoke $generator propagateKeyring "profile_root/config  01Cell03  01Node03 webserver1"
```

The following information explains the possible parameters that the **generate** operation accepts:

```
public void generate(java.lang.String
  configuration_root, java.lang.String myCellName,
 java.lang.String myNodeName, java.lang.String myServerName, java.lang.Boolean
 propagate, java.lang.Boolean propagateKeyring)
```

where:

configuration_root
is the root directory path for the configuration repository to be scanned.

myCellName
is the name of the cell in the configuration repository to restrict generation to.

myNodeName
is the name of the node in the configuration repository to restrict generation to.

myServerName
is the name of the server to restrict generation to.

propagate
is a boolean variable that specifies to propagate the configuration file.

propagateKeyring
is a boolean variable that specifies to propagate the keyring file.

The following network-centric example uses the **generate** operation to generate the plug-in configuration file for the cell:

Using Jython:

```
AdminControl.invoke(generator,'generate',"profile_root/config 01Cell03 null null plugin-cfg.xml")
```

Using Jacl:

```
$AdminControl invoke $generator generate "profile_root/config 01Cell03 null null plugin-cfg.xml"
```

The following information explains the possible parameters that the **generate** operation accepts:

```
public void generate(java.lang.String  app_server_root, java.lang.String
  configuration_root, java.lang.String myCellName,
 java.lang.String myNodeName, java.lang.String
 myServerName,java.lang.String myOutputFileName)
```

where:

app_server_root
is the root directory for the application server to run the command against.

configuration_root
is the root directory path for the configuration repository to be scanned.

myCellName
is the name of the cell in the configuration repository to restrict generation to.

myNodeName
is the name of the node in the configuration repository to restrict generation to.

myServerName
is the name of the server to restrict generation to.

myOutputFileName
is the path and filename of the generated plug-in configuration file.

Creating new virtual hosts using templates with scripting

Use scripting to create a new virtual host from a new or preexisting template.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Some configuration object types have templates that you can use when you create a virtual host. You can create a new virtual host using a preexisting template or by creating a new custom template. Perform the following steps to create a new virtual host using a template:

Procedure

1. If you want to create a new custom template, perform the following steps:
   a. Copy and paste the following file into a new file, *myvirtualhostname.xml*:
      app_server_root/config/templates/default/virtualhosts.xml
   b. Edit and customize the new *myvirtualhostname.xml* file.
   c. Place the new file in the following directory:
      app_server_root/config/templates/custom/

   If you want the new custom template to appear with the list of templates, use the AdminConfig object **reset** command. For example:
   - Using Jacl:
     ```
     $AdminConfig reset
     ```
   - Using Jython:
     ```
     AdminConfig.reset()
     ```

   The administrative console does not support the use of custom templates. The new template that you create will not be visible in the administrative console panels.
2. Use the AdminConfig object **listTemplates** command to list available templates, for example:
   - Using Jacl:
     ```
     $AdminConfig listTemplates VirtualHost
     ```
   - Using Jython:
     ```
     print AdminConfig.listTemplates('VirtualHost')
     ```

   Example output:
   ```
   default_host(templates/default:virtualhosts.xml#VirtualHost_1)
   my_host(templates/custom:virtualhostname.xml#VirtualHost_1)
   ```
3. Create a new virtual host. For example:
   - Using Jacl:
     ```
     set cell [$AdminConfig getid /Cell:NetworkDeploymentCell/]
     set vtempl [$AdminConfig listTemplates VirtualHost my_host]
     $AdminConfig createUsingTemplate VirtualHost $cell {{name newVirHost}} $vtempl
     ```
   - Using Jython:
     ```
     cell = AdminConfig.getid('/Cell:NetworkDeploymentCell/')
     vtempl = AdminConfig.listTemplates('VirtualHost', 'my_host')
     AdminConfig.createUsingTemplate('VirtualHost', cell, [['name', 'newVirHost']], vtempl)
     ```

4. Save the configuration changes. See the topic about saving configuration changes with the wsadmin tool for more information.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
The default Application Client for IBM WebSphere Application Server profile root is the `/QIBM/UserData/WebSphere/AppClient/V8/client/profiles/`*`profile_name`* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppServer/V8/Base` directory.

java_home

Table 449. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit`
64–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit`

plugins_profile_root
The default Web Server Plug-ins profile root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/`*`profile_name`* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the `/QIBM/ProdData/WebSphere/Plugins/V8/webserver` directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver` directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to `.exe`, `.dll`, `.so` objects) for the installed product. The product library name is `QWAS8`*`x`* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is `QWAS8A`. The *`app_server_root`*`/properties/product.properties` file contains the value for the product library of the installation, `was.install.library`, and is located under the *`app_server_root`* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/`*`profile_name`* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS8`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base` directory.

The `profiles` and `profileRegistry` subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Chapter 17. Managing servers and nodes with scripting

Use the wsadmin tool to administer your administrative architecture and runtime settings.

Procedure

- Use the following topics to manage server configurations with the wsadmin tool:
  -
  - "Stopping servers using scripting" on page 487
  - "Querying server state using scripting" on page 488
  - "Listing running applications on running servers using wsadmin scripting" on page 488
  - "Starting listener ports using scripting" on page 491
  - "Managing generic servers using scripting" on page 492
  - "Setting development mode for server objects using scripting" on page 493
  - "Disabling parallel startup using scripting" on page 493
  - "Obtaining server version information with scripting" on page 494
- Use the following topics to view wsadmin command reference information for server and node management:
  - "UnmanagedNodeCommands command group for the AdminTask object using wsadmin scripting" on page 546
  - "ManagedObjectMetadata command group for the AdminTask object" on page 505
  - "Utility command group of the AdminTask object" on page 502
  - "ConfigArchiveOperations command group for the AdminTask object using wsadmin scripting" on page 548
- Use the following topics to use properties files to manage your environment:
  - "Creating server, cluster, application, or authorization group objects using properties files and wsadmin scripting" on page 563
  - "Deleting server, cluster, application, or authorization group objects using properties files" on page 564
  - "Creating and deleting configuration objects using properties files and wsadmin scripting" on page 562

Stopping a node using wsadmin scripting

Use scripting to stop a node agent.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting article for more information.

About this task

Stopping the node agent on a remote machine process is an asynchronous action where the stop is initiated, and then control returns to the command line. Perform the following task to stop a node:

Procedure

1. Identify the node that you want to stop and assign it to a variable:

   Using Jacl:

   ```
   set na [$AdminControl queryNames type=NodeAgent,node=mynode,*]
   ```

Using Jython:

```
na = AdminControl.queryNames('type=NodeAgent,node=mynode,*')
```

2. Stop the node:

   Using Jacl:

   ```
   $AdminControl invoke $na stopNode
   ```

   Using Jython:

   ```
   AdminControl.invoke(na, 'stopNode')
   ```

Starting servers using scripting

You can use scripting and the wsadmin tool to start servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scriptingChapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 article for more information.

Procedure

Use the **startServer** command to start the server. This command has several syntax options. For example:

- To start a server on a WebSphere Application Server single server edition, choose one of the following options:
  - The following examples specify the server name only:

    Using Jacl:

    ```
    $AdminControl startServer serverName
    ```

    Using Jython:

    ```
    AdminControl.startServer('serverName')
    ```

  - The following example starts an application server with the node specified:
    - Using Jacl:

      ```
      $AdminControl startServer server1 mynode
      ```

    - Using Jython:

      ```
      print AdminControl.startServer('server1', 'mynode')
      ```

    Example output:

    ```
    WASX7319I: The serverStartupSyncEnabled attribute is set to false.  A start
    will be attempted for server "server1" but the configuration information for
    node "mynode" may not be current.
    WASX7262I: Start completed for server "server1" on node "mynode"
    ```

  - The following example specify the server name and wait time:
    - Using Jacl:

      ```
      $AdminControl startServer serverName 10
      ```

    - Using Jython:

      ```
      AdminControl.startServer('serverName', 10)
      ```

    where *10* is the maximum number of seconds waiting for the server to start.
- To start a server on a WebSphere Application Server network deployment edition, choose one of the following options:
  - The following example specifies the server name and the node name:
    - Using Jacl:

      ```
      $AdminControl startServer serverName nodeName
      ```

- Using Jython:

  ```
  AdminControl.startServer('serverName', 'nodeName')
  ```

– The following example specifies the server name, the node name, and the wait time:

- Using Jacl:

  ```
  $AdminControl startServer serverName nodeName 10
  ```

- Using Jython:

  ```
  AdminControl.startServer('serverName', 'nodeName', 10)
  ```

  where *10* is the number of seconds that the process should wait before starting the server.

Stopping servers using scripting

You can stop servers using scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Restriction: Concurrently stopping servers using Jacl or Jython threading (multiple threads) methods is not supported in WebSphereApplication Server Version 7.0.

Procedure

Use the **stopServer** command to stop the server. This command has several syntax options. For example:

- To stop a server on a WebSphere Application Server single server edition, choose one of the following options:
  – The following examples specify the server name only:

    Using Jacl:

    ```
    $AdminControl stopServer serverName
    ```

    Using Jython:

    ```
    AdminControl.stopServer('serverName')
    ```

  – The following examples stop an application server with the node specified:

    - Using Jacl:

      ```
      $AdminControl stopServer serverName mynode
      ```

    - Using Jython:

      ```
      print AdminControl.stopServer('serverName', 'mynode')
      ```

    Example output:

    ```
    WASX7337I: Invoked stop for server "serverName" Waiting for stop completion.
    WASX7264I: Stop completed for server "serverName" on node "mynode"
    ```

  – The following examples specify the server name and immediate:

    - Using Jacl:

      ```
      $AdminControl stopServer serverName immediate
      ```

    - Using Jython:

      ```
      AdminControl.stopServer('serverName', immediate)
      ```

- To stop a server on a WebSphere Application Server network deployment edition, choose one of the following options:
  – The following example specifies the server name and the node name:

    - Using Jacl:

```
$AdminControl stopServer serverName nodeName
```

- Using Jython:

```
AdminControl.stopServer('serverName', 'nodeName')
```

– The following example specifies the server name, the node name, and immediate:

- Using Jacl:

```
$AdminControl stopServer serverName nodeName immediate
```

- Using Jython:

```
AdminControl.stopServer('serverName', 'nodeName', immediate)
```

Querying server state using scripting

You can use the wsadmin tool and scripting to query server states.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

When querying the server state, the following command steps return a value of `STARTED` if the server is started. If the server is stopped, the command does not return a value.

Procedure

- Identify the server and assign it to the server variable. The following example returns the server MBean that matches the partial object name string:
  – Using Jacl:

```
set server [$AdminControl completeObjectName cell=mycell,node=mynode,
name=server1,type=Server,*]
```

  – Using Jython:

```
server = AdminControl.completeObjectName('cell=mycell,node=mynode,
name=server1,type=Server,*')
print server
```

  Example output:

```
WebSphere:cell=mycell,name=server1,mbeanIdentifier=server.xml#Server_1,
type=Server,node=mynode,process=server1,processType=ManagedProcess
```

  If the server is stopped, the **completeObjectName** command returns an empty string ('').
- Query for the state attribute. In addition to using the previous step, you can also query for the server state attribute. For example:
  – Using Jacl:

```
$AdminControl getAttribute $server state
```

  – Using Jython:

```
print AdminControl.getAttribute(server, 'state')
```

  The **getAttribute** command returns the value of a single attribute.

  Example output:

```
STARTED
```

Listing running applications on running servers using wsadmin scripting

Use the wsadmin tool and scripting to list all the running applications on all the running servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Use the following example to list all the running applications on all the running servers on each node of each cell.

Procedure

- Using Jacl:

```
*Provide this example as a Jacl script file and run it with  the "-f" option:
1  #-----------------------------------------------------------------
2  # lines 4 and 5 find all the cell and process them one at a time
3  #-----------------------------------------------------------------
4  set cells [$AdminConfig list Cell]
5  foreach cell $cells {
6     #-------------------------------------------------------------------------
7     # lines 10 and 11 find all the nodes belonging to the cell and
8     # process them at a time
9     #-------------------------------------------------------------------------
10    set nodes [$AdminConfig list Node $cell]
11    foreach node $nodes {
12       #---------------------------------------------------------------
13       # lines 16-20 find all the running servers belonging to the cell
14       # and node, and process them one at a time
15       #---------------------------------------------------------------
16       set cname [$AdminConfig showAttribute $cell name]
17       set nname [$AdminConfig showAttribute $node name]
18       set servs [$AdminControl queryNames type=Server,cell=$cname,node=$nname,*]
19       puts "Number of running servers on node $nname: [llength $servs]"
20       foreach server $servs {
21           #-----------------------------------------------------------
22           # lines 25-31 get some attributes from the server to display;
23           # invoke an operation on the server JVM to display a property.
24           #-----------------------------------------------------------
25           set sname [$AdminControl getAttribute $server name]
26           set ptype [$AdminControl getAttribute $server processType]
27           set pid   [$AdminControl getAttribute $server pid]
28           set state [$AdminControl getAttribute $server state]
29           set jvm [$AdminControl queryNames type=JVM,cell=$cname,
       node=$nname,process=$sname,*]
30           set osname [$AdminControl invoke $jvm getProperty os.name]
31           puts "  $sname ($ptype) has pid $pid; state: $state; on $osname"
32
j3           #-----------------------------------------------------------
34           # line 37-42 find the applications running on this server and
35           # display the application name.
35           #-----------------------------------------------------------
37           set apps [$AdminControl queryNames type=Application,
       cell=$cname,node=$nname,process=$sname,*]
38           puts "  Number of applications running on $sname: [llength $apps]"
39           foreach app $apps {
40              set aname [$AdminControl getAttribute $app name]
41              puts "    $aname"
42           }
43           puts "-----------------------------------------------------"
44           puts ""
```

```
45
46        }
47     }
48  }
```

- Using Jython:

  * Provide this example as a Jython script file and run it with the "-f" option:

```
1   #-------------------------------------------------------------
2   # lines 7 and 8 find all the cell and process them one at a time
3   #-------------------------------------------------------------
4   # get line separator
5   import  java.lang.System  as  sys
6   lineSeparator = sys.getProperty('line.separator')
7   cells = AdminConfig.list('Cell').split(lineSeparator)
8   for cell in cells:
9        #-------------------------------------------------------------
10     # lines 13 and 14 find all the nodes belonging to the cell and
11     # process them at a time
12     #-------------------------------------------------------------
13     nodes = AdminConfig.list('Node', cell).split(lineSeparator)
14     for node in nodes:
15        #-------------------------------------------------------------
16        # lines 19-23 find all the running servers belonging to the cell
17        # and node, and process them one at a time
18        #-------------------------------------------------------------
19        cname = AdminConfig.showAttribute(cell, 'name')
20        nname = AdminConfig.showAttribute(node, 'name')
21        servs = AdminControl.queryNames('type=Server,cell=' + cname +
       ',node=' + nname + ',*').split(lineSeparator)
22        print "Number of running servers on node " +
       nname + ": %s \n" % (len(servs))
23        for server in servs:
24          #-----------------------------------------------------
25          # lines 28-34 get some attributes from the server to display;
26          # invoke an operation on the server JVM to display a property.
27          #-----------------------------------------------------
28          sname = AdminControl.getAttribute(server, 'name')
29          ptype = AdminControl.getAttribute(server, 'processType')
30          pid   = AdminControl.getAttribute(server, 'pid')
31          state = AdminControl.getAttribute(server, 'state')
32           jvm = AdminControl.queryNames('type=JVM,cell=' +
       cname + ',node=' + nname + ',process=' + sname + ',*')
33          osname = AdminControl.invoke(jvm, 'getProperty', 'os.name')
34           print " " + sname + " " +  ptype + " has pid " + pid +
          "; state: " + state + "; on " +
           osname + "\n"
35
36           #-----------------------------------------------------
37           # line 40-45 find the applications running on this server and
38          # display the application name.
39          #-----------------------------------------------------
40          apps = AdminControl.queryNames('type=Application,cell=' +
      cname + ',node=' + nname + ',process=' + sname + ',*').
         split(lineSeparator)
41          print "Number of applications running on " + sname +
         ": %s \n" % (len(apps))
42          for app in apps:
43             aname = AdminControl.getAttribute(app, 'name')
44             print aname + "\n"
45         print "-----------------------------------------------------"
46         print "\n"
```

Results

Example output:

```
Number of running servers on node mynode: 2
mynode (NodeAgent) has pid 3592; state: STARTED; on Windows Vista
Number of applications running on mynode: 0
-----------------------------------------------------

server1 (ManagedProcess) has pid 3972; state: STARTED; on Windows Vista
Number of applications running on server1: 0
-----------------------------------------------------

Number of running servers on node mynodeManager: 1
dmgr (DeploymentManager) has pid 3308; state: STARTED; on Windows Vista
Number of applications running on dmgr: 2
adminconsole
filetransfer
-----------------------------------------------------
```

Starting listener ports using scripting

These steps demonstrate how to start a listener port on an application server using scripting.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Perform the following steps to start a listener port on an application server. The following example returns a list of listener port MBeans:

Procedure

1. Identify the listener port MBeans for the application server and assign it to the lPorts variable.
   - Using Jacl:

     ```
     set lPorts [$AdminControl queryNames type=ListenerPort,
     cell=mycell,node=mynode,process=server1,*]
     ```

   - Using Jython:

     ```
     lPorts = AdminControl.queryNames('type=ListenerPort,
     cell=mycell,node=mynode,process=server1,*')
     print lPorts
     ```

     Example output:

     ```
     WebSphere:cell=mycell,name=ListenerPort,mbeanIdentifier=server.xml#
     ListenerPort_1,type=ListenerPort,node=mynode,process=server1
     WebSphere:cell=mycell,name=listenerPort,mbeanIdentifier=ListenerPort,
     type=server.xml#ListenerPort_2,node=mynode,process=server1
     ```

2. Start the listener port if it is not started. For example:
   - Using Jacl:

     ```
     foreach lPort $lPorts {
          set state [$AdminControl getAttribute $lport started]
          if {$state == "false"} {
             $AdminControl invoke $lPort start
          }
       }
     ```

   - Using Jython:

```
# get line separator
import  java
lineSeparator = java.lang.System.getProperty('line.separator')

lPortsArray = lPorts.split(lineSeparator)
for lPort in lPortsArray:
 state = AdminControl.getAttribute(lPort, 'started')
 if state == 'false':
  AdminControl.invoke(lPort, 'start')
```

These pieces of Jacl and Jython code loop through the listener port MBeans. For each listener port MBean, get the attribute value for the started attribute. If the attribute value is set to `false`, then start the listener port by invoking the start operation on the MBean.

Managing generic servers using scripting

You can use WebSphere Application Server to define, start, stop, and monitor generic servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

A generic server is a server that the WebSphere Application Server manages but did not supply.

Procedure

- To define a generic server, use the following example:
  - Using Jacl:

    ```
    $AdminTask createGenericServer mynode {-name generic1 -ConfigProcDef
    {{"/mydir1/myStartCommand" "arg1 arg2" "" "" "/tmp/workingDirectory"
    "/mydir2/stopCommand" "argy argz"}}}
    $AdminConfig save
    ```

  - Using Jython:

    ```
    AdminTask.createGenericServer('mynode', '[-name generic1 -ConfigProcDef
    [[/mydir1/myStartCommand "a b c" "" "" /tmp/workingDirectory
    /mydir2/myStopCommand "x y z"]]]')
    AdminConfig.save()
    ```

- To start a generic server, use the `launchProcess` parameter, for example:
  - Using Jacl:

    ```
    set nodeagent [$AdminControl queryNames *:*,type=NodeAgent]
    $AdminControl invoke $nodeagent launchProcess generic1
    ```

  - Using Jython:

    ```
    nodeagent = AdminControl.queryNames ('*:*,type=NodeAgent')
    AdminControl.invoke(nodeagent, 'launchProcess', 'generic1')
    ```

  Example output:

  ```
  true
  ```

  or

  ```
  false
  ```

- To stop a generic server, use the `terminate` parameter, for example:
  - Using Jacl:

    ```
    set nodeagent [$AdminControl queryNames *:*,type=NodeAgent]
    $AdminControl invoke $nodeagent terminate generic1
    ```

  - Using Jython:

```
nodeagent = AdminControl.queryNames ('*:*,type=NodeAgent')
AdminControl.invoke(nodeagent, 'terminate', 'generic1')
```

Example output:

```
true
```

or

```
false
```

- To monitor the server state, use the `getProcessStatus` parameter, for example:
  - Using Jacl:

    ```
    $AdminControl invoke $nodeagent getProcessStatus generic1
    ```

    Using Jython:

    ```
    AdminControl.invoke(nodeagent, 'getProcessStatus', 'generic1')
    ```

    Example output:

    ```
    RUNNING
    ```

    or

    ```
    STOPPED
    ```

Setting development mode for server objects using scripting

You can use scripting and the wsadmin tool to configure development mode for server objects.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Perform the following steps to set the development mode for a server object:

Procedure

1. Locate the server object. The following example selects the first server found:
   - Using Jacl:

     ```
     set server [$AdminConfig getid /Server:server1/]
     ```

   - Using Jython:

     ```
     server = AdminConfig.getid('/Server:server1/')
     ```

2. Enable development mode:
   - Using Jacl:

     ```
     $AdminConfig modify $server "{developmentMode true}"
     ```

   - Using Jython:

     ```
     AdminConfig.modify(server, [['developmentMode', 'true']])
     ```

3. Save the configuration changes. See the topic about saving configuration changes with the wsadmin tool for more information.

Disabling parallel startup using scripting

You can use scripting to disable parallel startup of servers.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Perform the following steps to disable parallel startup:

Procedure

1. Locate the server object. The following example selects the first server found:
   - Using Jacl:
     ```
     set server[$AdminConfig getid /Server:server1/]
     ```
   - Using Jython:
     ```
     server = AdminConfig.getid('/Server:server1/']
     ```
2. Enable development mode. For example:
   - Using Jacl:
     ```
     $AdminConfig modify $server "{parallelStartEnabled false}"
     ```
   - Using Jython:
     ```
     AdminConfig.modify(server, [['parallelStartEnabled', 'false']])
     ```
3. Save the configuration changes. See the topic about saving configuration changes with the wsadmin tool for more information.

Obtaining server version information with scripting

Use the wsadmin tool and scripting to obtain server version information.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic about starting the wsadmin scripting client using wsadmin scripting for more information.

About this task

Perform the following steps to query the server version information:

Procedure

1. Identify the server and assign it to the server variable.
   - Using Jacl:
     ```
     set server [$AdminControl completeObjectName type=Server,name=server1,node=mynode,*]
     ```
   - Using Jython:
     ```
     server = AdminControl.completeObjectName('type=Server,name=server1,node=mynode,*')
     print server
     ```

   Example output:

   ```
   WebSphere:cell=mycell,name=server1,mbeanIdentifier=server.xml#Server_1,
   type=Server,node=mynode,process=server1,processType=ManagedProcess
   ```
2. Query the server version. The server version information is stored in the serverVersion attribute. The **getAttribute** command returns the attribute value of a single attribute, passing in the attribute name.
   - Using Jacl:
     ```
     $AdminControl getAttribute $server serverVersion
     ```
   - Using Jython:

```
print AdminControl.getAttribute(server, 'serverVersion')
```

NodeGroupCommands command group for the AdminTask object using wsadmin scripting

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the NodeGroupCommands group can be used to create and manage node groups and node group members.

The NodeGroupCommands command group for the AdminTask object includes the following commands:

- "addNodeGroupMember"
- "createNodeGroup" on page 496
- "createNodeGroupProperty" on page 496
- "listNodeGroupProperties" on page 497
- "listNodeGroups" on page 498
- "listNodes" on page 498
- "modifyNodeGroup" on page 499
- "modifyNodeGroupProperty" on page 499
- "removeNodeGroup" on page 500
- "removeNodeGroupMember" on page 501
- "removeNodeGroupProperty" on page 501

addNodeGroupMember

The **addNode Group Member** command adds a member to a node group. Nodes can be members of more than one node group. The command does validity checking to ensure the following:

- Distributed and z/OS nodes are not combined in the same node group.

Target object

The target object is the node group where the member will be created. This target object is required.

Parameters and return values

-nodeName
The name of the node that you want to add to a node group. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask addNodeGroupMember  WBINodeGroup {-nodeName WBINode}
  ```
- Using Jython string:
  ```
  AdminTask.addNodeGroupMember ('WBINodeGroup', '[-nodeName  WBINode]')
  ```
- Using Jython list:
  ```
  AdminTask.addNodeGroupMember ('WBINodeGroup', ['-nodeName',  'WBINode'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask addNodeGroupMember  {-interactive}
  ```
- Using Jython string:

```
AdminTask.addNodeGroupMember  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.addNodeGroupMember  (['-interactive'])
```

createNodeGroup

The **createNode Group** command creates a new node group. A node group consists of a group of nodes that are referred to as node group members. Optionally, you can create a short name and a description for the new node group.

Target object

The node group name to be created. This target object is required.

Parameters and return values

-shortName
The short name of the node group. This parameter is optional.

-description
The description of the node group. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createNodeGroup  WBINodeGroup
```

- Using Jython string:

```
AdminTask.createNodeGroup ('WBINodeGroup')
```

- Using Jython list:

```
AdminTask.createNodeGroup ('WBINodeGroup')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createNodeGroup  {-interactive}
```

- Using Jython string:

```
AdminTask.createNodeGroup  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createNodeGroup  (['-interactive'])
```

createNodeGroupProperty

The **createNode Group Property** command creates custom properties for a node group.

Target object

The name of the node group. This target object is required.

Parameters and return values

-name
The name of the custom property to create. This parameter is required.

-value
The value of the custom property. This parameter is optional.

-description
The description of the custom property. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask createNodeGroup Property WBINodeGroup  {-name Channel -value  "channel1"}
```
- Using Jython string:
```
AdminTask.createNodeGroup Property('WBINodeGroup',  '[-name Channel -value  channel1]')
```
- Using Jython list:
```
AdminTask.createNodeGroup Property('WBINodeGroup',  ['-name', 'Channel',  '-value', 'channel1'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask createNodeGroup Property {-interactive}
```
- Using Jython string:
```
AdminTask.createNodeGroup Property ('[-interactive]')
```
- Using Jython list:
```
AdminTask.createNodeGroup Property (['-interactive'])
```

listNodeGroupProperties

The **listNode Group Properties** command displays all of the custom properties of a node group.

Target object

The target object is name of the node group. This target object is required.

Parameters and return values

- Parameters: None
- Returns: A list of all of the custom properties of a node group.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask listNodeGroup Properties WBINodeGroup
```
- Using Jython string:
```
AdminTask.listNodeGroup Properties('WBINodeGroup')
```
- Using Jython list:
```
AdminTask.listNodeGroup Properties('WBINodeGroup')
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask listNodeGroup Properties {-interactive}
```
- Using Jython string:
```
AdminTask.listNodeGroup Properties ('[-interactive]')
```
- Using Jython list:
```
AdminTask.listNodeGroup Properties (['-interactive'])
```

listNodeGroups

The **listNode Groups** command returns the list of node groups from the configuration repository. You can pass an optional node name to the command that returns the list of node groups where the node resides.

Target object

The target object is name of the node. This target object is optional.

Parameters and return values

- Parameters: None
- Returns: A list of the node groups in the cell.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listNodeGroups  $AdminTask listNodeGroups  nodeName
  ```
- Using Jython string:
  ```
  AdminTask.listNodeGroups   AdminTask.listNodeGroups ('nodeName')
  ```
- Using Jython list:
  ```
  AdminTask.listNodeGroups   AdminTask.listNodeGroups ('nodeName')
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listNodeGroups  {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listNodeGroups  ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listNodeGroups  (['-interactive'])
  ```

listNodes

The **listNodes** command displays all of the nodes in the cell.

Target object

The target object is name of the node group. This target object is optional.

Parameters and return values

- Parameters: None
- Returns: A list of all the nodes in the cell

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask listNodes
  ```
- Using Jython string:
  ```
  AdminTask.listNodes()
  ```
- Using Jython list:

```
AdminTask.listNodes()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listNodes  {-interactive}
```

- Using Jython string:

```
AdminTask.listNodes  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listNodes  (['-interactive'])
```

modifyNodeGroup

The **modify Node Group** command modifies the configuration of a node group. The node group name cannot be changed. However, its short name and description are supported. Also, its node membership can be modified.

Target object

The target object is the node group name. This target object is required.

Parameters and return values

-shortName
The short name of the node group. This parameter is optional.

-description
The description of the node group. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask modifyNodeGroup  WBINodeGroup {-shortName  WBIGroup -description  "Default node group"}
```

- Using Jython string:

```
AdminTask.modifyNodeGroup  WBINodeGroup('[-shortName  WBIGroup -description  "WBI" node group]')
```

- Using Jython list:

```
AdminTask.modifyNodeGroup  WBINodeGroup(['-shortName',  'WBIGroup', '-description',  "WBI",
 'node', 'group'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask modifyNodeGroup  {-interactive}
```

- Using Jython string:

```
AdminTask.modifyNodeGroup  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.modifyNodeGroup  (['-interactive'])
```

modifyNodeGroupProperty

The **modify Node Group Property** command modifies custom properties for a node group

Target object

The name of the node group. This target object is required.

Parameters and return values

-name
The name of the custom property to modify. This parameter is required.

-value
The value of the custom property. This parameter is optional.

-description
The description of the custom property. This parameter is optional.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask modifyNodeGroup Property WBINodeGroup {-name  Channel -value "channel1"}
```
- Using Jython string:
```
AdminTask.modifyNode GroupProperty('WBINodeGroup',  '[-name Channel  -value channel1]')
```
- Using Jython list:
```
AdminTask.modifyNode GroupProperty('WBINodeGroup',  ['-name', 'Channel',  '-value', 'channel1'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask modifyNodeGroup Property {-interactive}
```
- Using Jython string:
```
AdminTask.modifyNodeGroup Property ('[-interactive]')
```
- Using Jython list:
```
AdminTask.modifyNodeGroup Property (['-interactive'])
```

removeNodeGroup

The **remove Node Group** command removes the configuration of a node group. You can remove a node group if it does not contain any members. Also, the default node group cannot be removed.

Target object

The name of the node group to be removed. This target object is required.

Parameters and return values

- Parameters: None
- Returns: The node group object ID.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask removeNodeGroup  WBINodeGroup
```
- Using Jython string:
```
AdminTask.removeNodeGroup ('WBINodeGroup')
```
- Using Jython list:
```
AdminTask.removeNodeGroup ('WBINodeGroup')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeNodeGroup  {-interactive}
```

- Using Jython string:

```
AdminTask.removeNodeGroup  ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeNodeGroup  (['-interactive'])
```

removeNodeGroupMember

The **removeNode Group Member** command removes the configuration of a node group member.

- A node must always be a member of at least one node group.
- You cannot remove a node from a node group that is part of a cluster in that node group.

Target object

The target object is the node group containing the member to be removed. This target object is required.

Parameters and return values

-nodeName
The name of the node to remove from a node group. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeNode GroupMember WBINodeGroup  {-nodeName WBINode}
```

- Using Jython string:

```
AdminTask.removeNode GroupMember('WBINodeGroup',  '[-nodeName WBINode]')
```

- Using Jython list:

```
AdminTask.removeNode GroupMember('WBINode Group', ['-nodeName',  'WBINode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeNodeGroup Member {-interactive}
```

- Using Jython string:

```
AdminTask.removeNodeGroup Member ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeNodeGroup Member (['-interactive'])
```

removeNodeGroupProperty

The **removeNode Group Property** command removes custom properties of a node group.

Target object

The name of the node group. This target object is required.

Parameters and return values

-name
The name of the custom property to remove. This parameter is required.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask removeNodeGroup Property WBINodeGroup  {-name Channel}
```

- Using Jython string:

```
AdminTask.removeNodeGroup Property('WBINodeGroup',  '[-name Channel]')
```

- Using Jython list:

```
AdminTask.removeNodeGroup Property('WBINodeGroup',  ['-name', 'Channel'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask removeNodeGroup Property {-interactive}
```

- Using Jython string:

```
AdminTask.removeNodeGroup Property ('[-interactive]')
```

- Using Jython list:

```
AdminTask.removeNodeGroup Property (['-interactive'])
```

Utility command group of the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the Utility group can be used to change the host name of a node, to query for the name of the deployment manager, and to determine if the system is a single server or network deployment.

The Utility command group for the AdminTask object includes the following commands:

- "changeHostName"
- "getDmgrProperties" on page 503
- "isFederated" on page 504

changeHostName

Use the **changeHost Name** command to change the host name of a node.

Target object

None

Parameters and return values

-hostName
The new host name. (String, required)

-nodeName
The name of the node whose host name will be changed. (String, required)

Optional parameters:

-systemName
The name of the z/OS system on which this node will run. This field is only required if a node is to be moved from one system to another, for example, form system SYSA to system SYSB. If you are not sure of the value you should specify for this parameter, issue the IPLINFO command on your z/OS system, and use the name that displays in the **Sysname=** field as the value for this parameter.

Note: When you run the changeHostName command interactively, the systemName parameter appears, but is only to be used for a z/OS system. The systemName parameter represents the z/OS system name that is defined in the `server.xml` file as a value for property, `was.ConfiguredSystemName`. When the systemName parameter is selected, this property is changed.

-regenDefaultCert
A request to regenerate default certificates. The only valid value for this parameter is "true" if you want to regenerate default certificates. Any other value assumes to NOT regenerate default certificates. The string argument is processed as a boolean. The boolean returned represents the value true if the string argument is not null and is equal, ignoring case, to the string "true". The regenDefaultCert parameter operates like `AdminTask.createChainedCertificate` and has the following default values:

- -keyStoreName "NodeDefaultKeyStore"
- -keyStoreScope "(node):" + nodeName
- -certificateAlias" "default_" + hostName
- -certificateCommonName" nodeName
- -certificateOrganization" "IBM"
- -certificateOrganizationalUnit" nodeName
- -certificateCountry "US"

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask changeHostName {-hostName host_name -nodeName node_name
-systemName system_name}
```

- Using Jython string:

```
AdminTask.changeHostName('-hostName host_name -nodeName node_name
-systemName system_name')
```

- Using Jython list:

```
AdminTask.changeHostName(['-hostName', 'host_name', '-nodeName', 'node_name',
'-systemName', 'system_name'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask changeHostName {-interactive}
```

- Using Jython string:

```
AdminTask.changeHostName ('[-interactive]')
```

- Using Jython list:

```
AdminTask.changeHostName (['-interactive'])
```

getDmgrProperties

Use the **getDmgr Properties** command to return the name of the deployment manager.

Target object

None

Parameters and return values

- Parameters: None

- Returns: The name of the deployment manager in a network deployment system. Returns an empty string if the system is a single server.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getDmgrProperties {}
```

- Using Jython:

```
AdminTask.getDmgrProperties()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getDmgrProperties {-interactive}
```

- Using Jython string:

```
AdminTask.getDmgrProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getDmgrProperties (['-interactive'])
```

isFederated

Use the **isFederated** command to check if the system is a single server or network deployment.

Target object

None

Parameters and return values

- Parameters: None
- Returns: Boolean. `true` if the system is a network deployment system. Otherwise it returns `false`.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask isFederated {}
```

- Using Jython string:

```
AdminTask.isFederated ()
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask isFederated {-interactive}
```

- Using Jython string:

```
AdminTask.isFederated ('[-interactive]')
```

- Using Jython list:

```
AdminTask.isFederated (['-interactive'])
```

ManagedObjectMetadata command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the ManagedObjectMetadata group can be used to retrieve configuration and metadata information for a specified node.

The ManagedObjectMetadata command group for the AdminTask object includes the following commands:

compareNodeVersion

The **compareNode Version** command compares the WebSphere Application Server version given a node that you specify and an input version.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

-version
A version number that you want to compare to the WebSphere Application Server version number.

Examples

Batch mode example usage:

- Using Jacl:
```
$AdminTask compareNode Version {-nodeName node1 -version 5}
```
- Using Jython string:
```
AdminTask.compareNode Version('[-nodeName node1 -version 5]')
```
- Using Jython list:
```
AdminTask.compareNode Version(['-nodeName', 'node1', '-version', '5'])
```

Interactive mode example usage:

- Using Jacl:
```
$AdminTask compareNode Version {-interactive}
```
- Using Jython string:
```
AdminTask.compareNode Version ('[-interactive]')
```
- Using Jython list:
```
AdminTask.compareNode Version (['-interactive'])
```

getAvailableSDKsOnNode

Run the getAvailableSDKsOnNode command to return a list of the names of the installed software development kits that a node can use. This command lists the software development kits that have been installed and are available for use by the node.

You might run this command before setting an SDK using the setter commands in the AdminSDKCmds command group. See AdminSDKCmds command group for the AdminTask object.

Target object

None

Required parameters

-nodeName
Specifies the name of the node for which you want a list of available software development kits. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getAvailableSDKsOnNode {-nodeName myNode}
```

- Using Jython string:

```
AdminTask.getAvailableSDKsOnNode('[-nodeName myNode]')
```

- Using Jython list:

```
AdminTask.getAvailableSDKsOnNode(['-nodeName', 'myNode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getAvailableSDKsOnNode {-interactive}
```

- Using Jython:

```
AdminTask.getAvailableSDKsOnNode('[-interactive]')
```

getMetadataProperties

The **getMetadata Properties** command obtains all metadata for the node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getMetadata Properties {-nodeName node1}
```

- Using Jython string:

```
AdminTask.getMetadata Properties('[-nodeName node1]')
```

- Using Jython list:

```
AdminTask.getMetadata Properties(['-nodeName', 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getMetadataProperties {-interactive}
```

- Using Jython string:

```
AdminTask.getMetadataProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getMetadataProperties (['-interactive'])
```

getMetadataProperty

The **getMetadata Property** command obtains metadata with the specified key for the node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

-propertyName
Metadata property key.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getMetadataProperty {-nodeName node1 -property Name
 com.ibm.websphere.base ProductVersion}
```

- Using Jython string:

```
AdminTask.getMetadataProperty ('[-nodeName node1 -propertyName
com.ibm. websphere.baseProductVersion]')
```

- Using Jython list:

```
AdminTask.getMetadataProperty (['-nodeName', 'node1', '-propertyName',
 'com.ibm. websphere.baseProductVersion'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getMetadataProperty {-interactive}
```

- Using Jython string:

```
AdminTask.getMetadataProperty ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getMetadataProperty (['-interactive'])
```

getNodeBaseProductVersion

The **getNode Base Product Version** command returns the version of the WebSphere Application Server for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeBaseProduct Version {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeBaseProduct Version('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeBaseProduct Version(['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeBaseProduct Version {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeBaseProduct Version ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeBaseProduct Version (['-interactive'])
  ```

getNodeMajorVersion

The **getNode Major Version** command returns the major version of the WebSphere Application Server for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMajorVersion {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMajorVersion ('[-nodeName node1]')
  ```

- Using Jython list:
  ```
  AdminTask.getNodeMajorVersion (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMajor Version {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMajor Version ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeMajor Version (['-interactive'])
  ```

getNodeMinorVersion

The **getNode Minor Version** command returns the minor version of the WebSphere Application Server for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMinorVersion {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMinorVersion ('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeMinorVersion (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeMinor Version {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeMinor Version ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeMinor Version (['-interactive'])
  ```

getNodePlatformOS

The **getNode Platform OS** command returns the operating system name for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodePlatformOS {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodePlatformOS ('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodePlatformOS (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodePlatformOS {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodePlatformOS ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodePlatformOS (['-interactive'])
  ```

getNodeSysplexName

The **getNode Sysplex Name** command returns the sysplex name for a node that you specify.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeSysplexName {-nodeName node1}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeSysplexName ('[-nodeName node1]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeSysplexName (['-nodeName', 'node1'])
  ```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask getNodeSysplexName {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.getNodeSysplexName ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.getNodeSysplexName (['-interactive'])
  ```

getSDKPropertiesOnNode

Run the getSDKPropertiesOnNode command to return a list of the software development kit properties for a node. This command lists properties of the software development kits that have been installed and are available for use by the node.

When the -sdkName option is not specified, the command returns all properties for all available software development kits. When the -sdkAttributes option is specified, the command returns only properties for the specified SDK attributes.

You might run this command before setting an SDK using the setter commands in the AdminSDKCmds command group. See AdminSDKCmds command group for the AdminTask object.

Target object

None

Required parameters

-nodeName
Specifies the name of the node for which you want a list of installed SDK properties. (String, required)

Optional parameters

-sdkName
Specifies the name of an SDK whose properties you want returned. (String, optional)

-sdkAttributes
Specifies a list of the SDK attributes whose properties you want returned. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getSDKPropertiesOnNode {-nodeName myNode}

$AdminTask getSDKPropertiesOnNode {-nodeName myNode -sdkName 1.6_32}

$AdminTask getSDKPropertiesOnNode {-nodeName myNode -sdkAttributes
 {location}}

$AdminTask getSDKPropertiesOnNode {-nodeName myNode -sdkName 1.6_32
 -sdkAttributes {location version}}
```

- Using Jython string:

```
AdminTask.getSDKPropertiesOnNode('[-nodeName myNode]')

AdminTask.getSDKPropertiesOnNode('[-nodeName myNode -sdkName 1.6_32]')

AdminTask.getSDKPropertiesOnNode('[-nodeName myNode -sdkAttributes
[location version]]')

AdminTask.getSDKPropertiesOnNode('[-nodeName myNode -sdkName 1.6_32
-sdkAttributes [location version]]')
```

- Using Jython list:

```
AdminTask.getSDKPropertiesOnNode(['-nodeName', 'myNode'])

AdminTask.getSDKPropertiesOnNode(['-nodeName', 'myNode','-sdkName',
'1.6_32'])

AdminTask.getSDKPropertiesOnNode(['-nodeName', 'myNode', '-sdkAttributes',
'[location version]'])

AdminTask.getSDKPropertiesOnNode(['-nodeName', 'myNode', '-sdkName',
 '1.6_32', '-sdkAttributes', '[location version]'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getSDKPropertiesOnNode {-interactive}
```

- Using Jython:

```
AdminTask.getSDKPropertiesOnNode('[-interactive]')
```

isNodeZOS

The **isNodeZOS** command tests if a node that you specify is running on the z/OS platform. This command does not apply to distributed platforms or to WebSphere Application Server-Express.

Target object

None

Parameters and return values

-nodeName
The name of the node associated with the metadata you want this command to return.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask isNodeZOS {-nodeName node1}
```

- Using Jython string:

```
AdminTask.isNodeZOS('[ -nodeName node1]')
```

- Using Jython list:

```
AdminTask.isNodeZOS([' -nodeName', 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask isNodeZOS {-interactive}
```

- Using Jython string:

```
AdminTask.isNodeZOS ('[-interactive]')
```

- Using Jython list:

```
AdminTask.isNodeZOS (['-interactive'])
```

AdminSDKCmds command group for the AdminTask object

You can use commands and parameters in the AdminSDKCmds group in the Jython or Jacl scripting languages to manage software development kit configurations.

Using the commands and parameters in the AdminSDKCmds group for the AdminTask object, you can see what software development kits are not used by a node, get or set the default software development kit (SDK) for a node, and get or set an SDK for a server.

The AdminSDKCmds command group for the AdminTask object includes the following commands:

Every WebSphere Application Server version and operating system has a default SDK. For example, the default SDK for WebSphere Application Server Version 8 on workstations might be `1.6_32` for 32-bit operating systems and `1.6_64` for 64-bit operating systems. The name of an SDK that is installed at a particular computer location must be unique. On workstations, the default SDK is installed in a directory name that starts with `${WAS_HOME}/java` for both 32- and 64-bit operating systems; for example, `${WAS_HOME}/java`, `${WAS_HOME}/java_1.6_32`, or `${WAS_HOME}/java_1.6_64`.

For the OS/400® operating system, the software development kits are part of the operating system installation and are not part of the WebSphere Application Server installation. For the OS/390® operating system, symbolic links to the software development kits are in `${WAS_HOME}/java*` directories.

Attention: If the managesdk command is used to change the SDK for a profile from a 31-bit (z/OS) or 32-bit (IBM i) SDK to a 64-bit SDK, and you are using third-party resource adapters, consider the following information to avoid potential problems. This information does not apply to any of the built-in resource adapters shipped with the WebSphere Application Server product, including the IBM WebSphere Relational Resource Adapter, the IBM WebSphere MQ Resource Adapter, or the IBM SIB JMS Resource Adapter as they have been fully tested to work with all IBM SDKs. Because resource adapters can use non-Java libraries containing platform-specific native code, it is possible that changing the SDK from 31-bit (z/OS) or 32-bit (IBM i) to 64-bit, or from 64-bit to 31-bit or 32-bit, might result in the resource adapter not functioning properly. If a third-party resource adapter is installed, either stand-alone or embedded in an enterprise application, on a server for which you intend to change the SDK, verify with the supplier of that resource adapter that any native libraries it uses are compatible with the selected SDK.

getNodeDefaultSDK

Use the getNodeDefaultSDK command to return the values of the default software development kit (SDK) for a node. The returned values include the Java home and SDK name.

Target object

None

Required parameters

-nodeName
Specifies the name of the node whose default SDK values you want returned. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getNodeDefaultSDK {-nodeName myNode}
```

- Using Jython string:

```
AdminTask.getNodeDefaultSDK('[-nodeName myNode]')
```

- Using Jython list:

```
AdminTask.getNodeDefaultSDK(['-nodeName', 'myNode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getNodeDefaultSDK {-interactive}
```

- Using Jython:

```
AdminTask.getNodeDefaultSDK('[-interactive]')
```

getSDKVersion

Run the getSDKVersion command to return the version number of the software development kit in use.

Target object

None

Required parameters

None

Optional parameters

-nodeName
Specifies the name of the node whose SDK version you want returned. Do not specify a -clusterName value with a -nodeName value. (String, optional)

-serverName
Specifies the name of the server whose SDK version you want returned. If you specify a -serverName value, specify a -nodeName value as well and do not specify a -clusterName value. (String, optional)

-clusterName
Specifies the name of the cluster whose SDK version you want returned. If you specify a -clusterName value, do not specify a -nodeName or -serverName value. (String, optional)

-highest
Specifies whether to return the highest SDK version number. By default, the lowest SDK version number is returned. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getSDKVersion {-nodeName myNode -serverName myServer}
```

- Using Jython string:

```
AdminTask.getSDKVersion('[-nodeName myNode -serverName myServer]')
```

- Using Jython list:

```
AdminTask.getSDKVersion(['-nodeName', 'myNode', ' -serverName', 'myServer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getSDKVersion {-interactive}
```

- Using Jython:

```
AdminTask.getSDKVersion('[-interactive]')
```

getServerSDK

Use the getServerSDK command to return the values of the software development kit for a server. If a valid SDK value is set for the server, the returned values include the Java home and SDK name of the default SDK for the server.

If no SDK value is set for the server, the command returns nothing for the Java home value because a `variables.xml` file does not exist for the server or a JAVA_HOME entry does not exist in the `variables.xml` file. For the SDK name value, the command returns the node SDK name because the node SDK is the default SDK for a server when a valid SDK has not yet been set using the setServerSDK command.

Target object

None

Required parameters

-nodeName
Specifies the name of the node on which the server runs. (String, required)

-serverName
Specifies the name of the server whose SDK values you want returned. (String, required)

Optional parameters

-checkOnly
Specifies whether to check only the variable. (Boolean, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getServerSDK {-nodeName myNode -serverName myServer}
```

- Using Jython string:

```
AdminTask.getServerSDK('[-nodeName myNode -serverName myServer]')
```

- Using Jython list:

```
AdminTask.getServerSDK(['-nodeName', 'myNode', '–serverName', 'myServer'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getServerSDK {-interactive}
```

- Using Jython:

```
AdminTask.getServerSDK('[-interactive]')
```

getUnusedSDKsOnNode

Run the getUnusedSDKsOnNode command to return a list of the names of software development kits that a node is not using.

Target object

None

Required parameters

-nodeName
Specifies the name of the node whose unused SDK names you want returned. (String, required)

Optional parameters

None

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getUnusedSDKsOnNode {-nodeName myNode}
```

- Using Jython string:

```
AdminTask.getUnusedSDKsOnNode('[-nodeName myNode]')
```

- Using Jython list:

```
AdminTask.getUnusedSDKsOnNode(['-nodeName', 'myNode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getUnusedSDKsOnNode {-interactive}
```

- Using Jython:

```
AdminTask.getUnusedSDKsOnNode('[-interactive]')
```

setNodeDefaultSDK

Use the setNodeDefaultSDK command to assign a default software development kit for a node. For the command, specify either the SDK Java home or the SDK name, but not both.

Note: If you change the node SDK, ensure that the options and properties for the Java command are compatible with the new SDK. See Configuring the JVM.

Target object

None

Required parameters

-nodeName
Specifies the name of the node for which you want to set a default SDK. (String, required)

Optional parameters

To set a node default SDK, specify the required -nodeName parameter with either -javahome or -sdkName. Both the -javahome and -sdkName parameters are optional but you must specify either one of the parameters.

To clear all SDK settings for all servers on a node, specify the required -nodeName parameter with either -javahome or -sdkName and with the optional -clearServerSDKs parameter set to `true`.

-javahome
Specifies the Java home of the SDK that you want the node to use. If you specify a -javahome value, do not specify a value for the -sdkName parameter. (String, optional)

-sdkName
Specifies the name of the SDK that you want the node to use. If you specify a value for this -sdkName parameter, do not specify a value for the -javahome parameter. (String, optional)

-clearServerSDKs
Specifies to clear any SDK value settings for all servers on a node. (Boolean, optional)

To clear SDK value settings for all servers on a node, specify `true` for -clearServerSDKs. After the server SDK value settings are cleared, servers use the SDK value setting for the node.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setNodeDefaultSDK {-nodeName myNode -sdkName 1.6_32}
$AdminTask setNodeDefaultSDK {-nodeName myNode -sdkName 1.6_32 -clearServerSDKs true}
```

- Using Jython string:

```
AdminTask.setNodeDefaultSDK('[-nodeName myNode -sdkName 1.6_32]')
AdminTask.setNodeDefaultSDK('[-nodeName myNode -sdkName 1.6_32 -clearServerSDKs true]')
```

- Using Jython list:

```
AdminTask.setNodeDefaultSDK(['-nodeName', 'myNode', '-javahome', '${JAVA_LOCATION_1.6_32}'])
AdminTask.setNodeDefaultSDK(['-nodeName', 'myNode', '-sdkName', '1.6_32', '-clearServerSDKs', 'true'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setNodeDefaultSDK {-interactive}
```

- Using Jython:

```
AdminTask.setNodeDefaultSDK('[-interactive]')
```

setServerSDK

Use the setServerSDK command to assign a software development kit for a server. The command creates a `variables.xml` file for the server that designates the SDK. For the command, specify either the cluster or both the node and server. Optionally specify either the SDK Java home or the SDK name, but not both.

To clear the server SDK assignment, do not specify values for SDK Java home or the SDK name. For example, if server1 is assigned SDK 1.6_32, run setServerSDK without the -javahome and -sdkName parameters to have no SDK assigned for server1.

Note: If you change the server SDK, ensure that the options and properties for the Java command are compatible with the new SDK. See Configuring the JVM.

Target object

None

Required parameters

None

Optional parameters

-nodeName
Specifies the name of the node on which the server runs. If you specify a -nodeName value, specify a -serverName value as well and do not specify a -clusterName value. (String, optional)

-serverName
Specifies the name of the server for which you want to set a SDK. If you specify a -serverName value, specify a -nodeName value as well and do not specify a -clusterName value. (String, optional)

-clusterName
Specifies the name of the cluster for which you want to set a SDK. If you specify a -cluster value, do not specify a -nodeName or -serverName value. (String, optional)

-javahome
Specifies the Java home of the SDK that you want the server to use. If you specify a -javahome value, do not specify a value for the -sdkName parameter. (String, optional)

-sdkName
Specifies the name of the SDK that you want the server to use. If you specify a value for this -sdkName parameter, do not specify a value for the -javahome parameter. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setServerSDK {-nodeName myNode -serverName myServer -sdkName 1.6_32}
```

- Using Jython string:

```
AdminTask.setServerSDK('[-nodeName myNode -serverName myServer -sdkName 1.6_32]')
```

- Using Jython list:

```
AdminTask.setServerSDK(['-nodeName', 'myNode', '–serverName', 'myServer', '-javahome', '${JAVA_LOCATION_1.6_32}'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setServerSDK {-interactive}
```

- Using Jython:

```
AdminTask.setServerSDK('[-interactive]')
```

ServerManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the ServerManagement group can be used to create and manage application server, web server, proxy server, generic server and Java virtual machine (JVM) configurations.

The ServerManagement command group for the AdminTask object includes the following commands:

createApplicationServer

Use the **createApplicationServer** command to create a new application server.

Target object

Specifies the name of the node (String, required)

Required parameters

-name
The name of the server that you want to create. (String, required)

Optional parameters

-templateName
The name of the template from which to base the server. (String, optional)

-genUniquePorts
Specifies whether the system generates unique HTTP ports for the server. The default value is true. Specify false if you do not want to generate unique HTTP ports for the server. (Boolean)

-templateLocation
The configuration Id that represents the location of a template. To list all of the available templates, you can use the following command:

```
print AdminTask.listServerTemplates('-serverType WEB_SERVER')
 Example usage:
 -templateLocation IHS(templates/servertypes/WEB_SERVER/servers/IHS|server.xml)
```

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createApplicationServer ndnode1 {-name test1 -templateName default}
```

- Using Jython string:

```
AdminTask.createApplicationServer(ndnode1, '[-name test1 -templateName
 default]")
```

- Using Jython list:

```
AdminTask.createApplicationServer(ndnode1, ['-name', 'test1',  '-templateName',
 'default'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createApplicationServer {-interactive}
```

- Using Jython:

```
AdminTask.createApplicationServer ('-interactive')
```

createApplicationServerTemplate

The **createApplicationServerTemplate** command creates a new application server template.

Target object

None

Required parameters

-templateName
Specifies the name of the application server template that you want to create. (String, required)

-serverName
Specifies the name of the server from which to base the template. (String, required)

-nodeName
Specifies the node that corresponds to the server from which to base the template. (String, required)

Optional parameters

-description
Specifies the description of the template. (String)

-templateLocation
Specifies a configuration Id that represents the location to place the template. (String)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=
configuration_id
```

The configuration Id value is the application server template, which is `templates\servertypes\APPLICATION_SERVER|servertype-metadata.xml`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createApplicationServerTemplate {-templateName newTemplate -serverName
 server1  -nodeName ndnode1 -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=APPLICATION_SERVER,_Websphere_Config_Data_Id=
 templates/servertypes/APPLICATION_SERVER|servertype-metadata.xml}
```

- Using Jython string:

```
AdminTask.createApplicationServerTemplate('[-templateName newTemplate -serverName
 server1 -nodeName ndnode1  -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=
 APPLICATION_SERVER,_Websphere_Config_Data_Id=templates/servertypes/APPLICATION_SERVER|servertype-
metadata.xml]')
```

- Using Jython list:

```
AdminTask.createApplicationServerTemplate(['-templateName',  'newTemplate', '-serverName',
 'server1', '-nodeName',  'ndnode1', '-description', "This is my new template"])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createApplicationServerTemplate {-interactive}
```

- Using Jython string:

```
AdminTask.createApplicationServerTemplate ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createApplicationServerTemplate (['-interactive'])
```

createGenericServer

Use the **createGenericServer** command to create a new generic server in the configuration. A generic server is a server that the WebSphere Application Server manages, but did not supply. The **createGenericServer** command provides an additional step, ConfigProcDef, that you can use to configure the parameters that are specific to generic servers.

Target object

Specifies the name of the node (String, required)

Required parameters

-name
The name of the server that you want to create.

Optional parameters

-templateName
Picks up a server template. This step provides a list of application server templates for the node and server type. The default value is the default templates for the server type. (String, optional)

-genUniquePorts
Specifies whether the system generates unique HTTP ports for the server. The default value is true. Specify false if you do not want to generate unique HTTP ports for the server. (Boolean)

-templateLocation
The configuration Id that represents the location of a template. Specify the `_Websphere_Config_Data_Id=templates/servertypes/GENERIC_SERVER|servertype-metadata.xml` configuration Id to create a generic server. (ObjectName)

-startCommand
Indicates the path to the command that will run when this generic server is started. (String, optional)

-startCommandArgs
Indicates the arguments to pass to the startCommand when the generic server is started. (String, optional)

-executableTargetKind
Specifies whether a Java class name (use `JAVA_CLASS`) or the name of an executable JAR file (use `EXECUTABLE_JAR`) will be used as the executable target for this process. This field should be left blank for binary executables. This parameter is only applicable for Java processes. (String optional)

-executableTarget
Specifies the name of the executable target (a Java class containing a main() method or the name of an executable JAR), depending on the executable target type. This field should be left blank for binary executables. This parameter is only applicable for Java processes. (String, optional)

-workingDirectory
Specifies the working directory for the generic server.

-stopCommand
Indicates the path to the command that will run when this generic server is stopped. (String, optional)

-stopCommandArgs
Indicates the arguments to pass to the stopCommand parameter when the generic server is stopped. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createGenericServer jim667BaseNode {-name jgeneric  -ConfigProcDef
 {{"/usr/bin/myStartCommand" "arg1 arg2" "" ""  "/tmp/workingDirectory" "/tmp/stopCommand"
"argy argz"}}}
```

- Using Jython string:

```
AdminTask.createGenericServer('jim667BaseNode', '[-name jgeneric  -ConfigProcDef
 [[/usr/bin/myStartCommand "arg1 arg2" "" ""  /tmp/workingDirectory /tmp/StopCommand
"argy argz"]]]')
```

- Using Jython list:

```
AdminTask.createGenericServer('jim667BaseNode', ['-name', 'jgeneric',
 '-ConfigProcDef',  [['/usr/bin/myStartCommand', "arg1 arg2" "" "", '/tmp/workingDirectory',
'/tmp/StopCommand', "argy
 argz"]]])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createGenericServer {-interactive}
```

- Using Jython string:

```
AdminTask.createGenericServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createGenericServer (['-interactive'])
```

createGenericServerTemplate

The **createGenericServerTemplate** command creates a new generic server template.

Target object

None

Required parameters

-templateName
Specifies the name of the generic server template that you want to create. (String, required)

-serverName
Specifies the name of the server from which to base the template. (String, required)

-nodeName
Specifies the node that corresponds to the server from which to base the template. (String, required)

Optional parameters

-description
Specifies the description of the template. (String)

-templateLocation
Specifies a configuration Id that represents the location to place the template. (String)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=
configuration_id
```

The configuration Id value is the generic server template, which is `templates\servertypes\GENERIC_SERVER|servertype-metadata.xml`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createGenericServerTemplate {-templateName newTemplate -serverName
 server1  -nodeName ndnode1 -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=GENERIC_SERVER,_Websphere_Config_Data_Id=
 templates/servertypes/GENERIC_SERVER|servertype-metadata.xml}
```

- Using Jython string:

```
AdminTask.createGenericServerTemplate('[-templateName newTemplate -serverName
 server1 -nodeName ndnode1  -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=
 GENERIC_SERVER,_Websphere_Config_Data_Id=templates/servertypes/GENERIC_SERVER|servertype-metadata.xml]')
```

- Using Jython list:

```
AdminTask.createGenericServerTemplate(['-templateName',  'newTemplate', '-serverName',
 'server1', '-nodeName',  'ndnode1', '-description', "This is my new template"])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createGenericServerTemplate {-interactive}
```

- Using Jython string:

```
AdminTask.createGenericServerTemplate ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createGenericServerTemplate (['-interactive'])
```

createGenericServerTemplate

Use the createGenericServerTemplate command to create a server template based on a server configuration.

Target object

None.

Required parameters

-serverName
Specifies the name of the server of interest. (String, required)

-nodeName
Specifies the name of the node of interest. (String, required)

-templateName
Specifies the name of the template to create. (String, required)

Optional parameters

-description
Provides a description for the template to be created. (String, optional)

-templateLocation
Specifies a configuration Id that represents the location of the template. If this parameter is not specified, the system uses the default location. (String, optional)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=
configuration_id
```

The configuration Id can be one of the following values:

- To create a web server template:
  `templates\servertypes\WEB_SERVER|servertype-metadata.xml`
- To create an application server template:
  `templates\servertypes\APPLICATION_SERVER|servertype-metadata.xml`
-

- To create a generic server template:
  `templates\servertypes\GENERIC_SERVER|servertype-metadata.xml`
- To create a proxy server template:
  `templates\servertypes\PROXY_SERVER|servertype-metadata.xml`

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createGenericServerTemplate {-interactive}
```

- Using Jython:

```
AdminTask.createGenericServerTemplate('-interactive')
```

createProxyServer

Use the **createProxyServer** command to create a new proxy server in the configuration. The proxy server is a specific type of application server that routes HTTP requests to content servers that perform the work. The proxy server is the initial point of entry, after the firewall, for requests into the enterprise.

Target object

Specifies the name of the node (String, required)

Required parameters

-name
The name of the server to create. (String)

Optional parameters

-templateName
Picks up a server template. This step provides a list of application server templates for the node and server type. The default value is the default templates for the server type. (String)

-genUniquePorts
Specifies whether the system generates unique HTTP ports for the server. The default value is true. Specify false if you do not want to generate unique HTTP ports for the server. (Boolean)

-templateLocation
Specifies the location of the template on your system. Use the system defined location if you are unsure of the location. (String)

-clusterName
Specifies the name of the cluster to which the system assigns the server. (String)

Optional steps

-ConfigCoreGroup

coregroupName
Specifies the name of the core group to configure. (String)

-selectProtocols

list Specifies a list of protocols that the proxy server supports. (java.util.List)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createProxyServer myNode {-name myProxyServer -templateName myTemplate
 -ConfigCoreGroup [-coregroupName [myCoreGroup]] -selectProtocols [-list [HTTP, SIP]]}
```

- Using Jython string:

```
AdminTask.createProxyServer('myNode', '[-name myProxyServer -templateName myTemplate
 -ConfigCoreGroup [-coregroupName [myCoreGroup]] -selectProtocols [-list [HTTP, SIP]]]')
```

- Using Jython list:

```
AdminTask.createProxyServer(myNode, ['-name', 'myProxyServer', '-templateName', 'myTemplate',
 '-ConfigCoreGroup', '[-coregroupName [myCoreGroup]]', '-selectProtocols', '[-list [HTTP, SIP]]'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createProxyServer {-interactive}
```

- Using Jython:

```
AdminTask.createProxyServer('-interactive')
```

createProxyServerTemplate

Use the **createProxyServerTemplate** command to create a new proxy server template based on an existing proxy server configuration.

Target object

None

Required parameters

-templateName
Specifies the name of the proxy server template to create. (String)

-serverName
Specifies the name of the proxy server of interest. (String)

-nodeName
Specifies the name of the node of interest. (String)

Optional parameters

-description
Specifies a description for the new server template. (String)

-templateLocation
Specifies a configuration Id that represents the location to place the template. Use the system defined location if you are unsure of the location. (String)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=
configuration_id
```

The configuration Id value is the proxy server template, which is `templates\servertypes\PROXY_SERVER|servertype-metadata.xml`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createProxyServerTemplate {-templateName proxyServerTemplate -serverName proxyServer1
 -nodeName myNode}
```

- Using Jython string:

```
AdminTask.createProxyServerTemplate('[-templateName proxyServerTemplate -serverName proxyServer1
 -nodeName myNode]')
```

- Using Jython list:

```
AdminTask.createProxyServerTemplate(['-templateName', 'proxyServerTemplate', '-serverName',
 'proxyServer1', '-nodeName', 'myNode'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createProxyServerTemplate {-interactive}
```

- Using Jython:

```
AdminTask.createProxyServerTemplate('[-interactive]')
```

createServerType

depfeat: The createServerType command is deprecated. No alternative command is provided for this deprecation.

Use the createServerType command to define a server type.

Target object

None.

Required parameters

-version
Specifies the product version. (String, required)

-serverType
Specifies the server type of interest. (String, required)

-createTemplateCommand
Specifies the command to use to create a server template. (String, required)

-createCommand
Specifies the command to use to create a server. (String, required)

-configValidator
Specifies the name of the configuration validator class. (String, required)

Optional parameters

-defaultTemplateName
Specifies the name of the default template. The default value is `default`. (String, optional)

Sample output

The command returns the object name of the server type that was created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createServerType {-version version -serverType serverType
 -createTemplateCommand name -createCommand name}
```

- Using Jython string:

```
AdminTask.createServerType('[-version version -serverType serverType
 -createTemplateCommand name -createCommand name]')
```

- Using Jython list:

```
AdminTask.createServerType(['-version', 'version', '-serverType',
 'serverType','-createTemplateCommand', 'name', '-createCommand', 'name'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createServerType {-interactive}
```

- Using Jython:

```
AdminTask.createServerType('-interactive')
```

createWebServer

Use the **createWebServer** command to create a web server definition. This command creates a web server definition using a template and configures the web server definition properties. Web server definitions generate and propagate the `plugin-config.xml` file for each web server. For IBM HTTP Server only, you can use web server definitions to administer and configure IBM HTTP Server web servers with the administrative console. These functions include: Start, Stop, View logs, View and Edit configuration files.

Target object

Specifies the name of the node (String, required).

Required parameters

-name
Specifies the name of the server. (String, required)

-serverConfig
Specifies the web server definition properties. Use this parameter and associated options to specify configuration properties for the IBM HTTP Server. Specify the following values in order in a list with the -serverConfig parameter:

webPort
Specifies the port number of the web server. This option is required for all web servers. (Integer, required)

webInstallRoot
Specifies the install path directory for the web server. This option is required for IBM HTTP Server Admin Function. (String, required)

pluginInstallRoot
Specifies the installation root directory where the plug-in for the web server is installed. This option is required for all web servers. (String, required)

configurationFile
Specifies the file path for the IBM HTTP Server. This option is required for view and edit of the IBM HTTP Server Configuration file only. (String, required)

serviceName
Specifies the windows service name on which to start the IBM HTTP Server. This option is required for start and stop of the IBM HTTP Server web server only. (String, required)

errorLogfile
Specifies the path for the IBM HTTP Server error log (`error.log`) (String, required)

accessLogfile
Specifies the path for the IBM HTTP Server access log (access.log). (String, required)

webProtocol
Specifies the IBM HTTP Server administration server running with an unmanaged or remote web server. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)

webAppMapping
Specifies configuration information for web application mapping. (String, required)

-remoteServerConfig
Specifies additional web server definition properties that are only necessary if the IBM HTTP Server web server is installed on a machine remote from the application server. Specify the following values in order in a list with the remoteServerConfig parameter:

adminPort
Specifies the port of the IBM HTTP Server administrative server. The administration server is installed on the same machine as the IBM HTTP Server and handles administrative requests to the IBM HTTP Server web server. (String, required)

adminProtocol
Specifies the administrative protocol title. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)

adminUserID
Specifies the user ID, if authentication is activated on the Administration server in the admin configuration file (`admin.conf`). This value should match the authentication in the admin.conf file. (String, optional)

adminPasswd
Specifies the password for the user ID. The password is generated by the htpasswd utility in the `admin.passwd` file. (String, optional)

Optional parameters

-templateName
Specifies the name of the template that you want to use. Templates include the following: `IHS`, `iPlanet`, `IIS`, `DOMINO`, `APACHE`. The default template is `IHS`. (String, required)

-genUniquePorts
Specifies whether the system generates unique HTTP ports for the server. The default value is true. Specify false if you do not want to generate unique HTTP ports for the server. (Boolean)

-templateLocation
The configuration Id that represents the location of a template. Specify the `_Websphere_Config_Data_Id=templates/servertypes/WEB_SERVER|servertype-metadata.xml` configuration Id to create a generic server. (ObjectName)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createWebServer myNode {-name wsname -serverConfig {{80
 /opt/path/to/ihs   /opt/path/to/plugin  /opt/path/to/plugin.xml  "windows service"
 /opt/path/to/error.log   /opt/path/to/access.log  HTTP}} -remoteServerConfig {{
8008 user  password HTTP}}
```

- Using Jython list:

```
print AdminTask.createWebServer(myNode, ['-name', wsname,
 '-serverConfig',[['80','/opt/path/to/ihs', '/opt/path/to/plugin',
'/opt/path/to/plugin.xml',  'windows service','/opt/path/to/error.log',
'/opt/path/to/access.log','HTTP']],'-remoteServerConfig',  [['8008','user',
'password','HTTP']]])
```

- Using Jython string:

```
AdminTask.createWebServer('myNode','-name wsname -serverConfig [80
 /opt/path/to/ihs  /opt/path/to/plugin /opt/path/to/plugin.xml "windows service"
/opt/path/to/error.log /opt/path/to/access.log HTTP]
-remoteServerConfig [8008 user password HTTP]')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createWebServer -interactive
```

- Using Jython string:

```
AdminTask.createWebServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createWebServer (['-interactive'])
```

createWebServerTemplate

The **createWebServerTemplate** command creates a new web server template.

Target object

None

Required parameters

-templateName
Specifies the name of the web server template that you want to create. (String, required)

-serverName
Specifies the name of the server from which to base the template. (String, required)

-nodeName
Specifies the node that corresponds to the server from which to base the template. (String, required)

Optional parameters

-description
Specifies the description of the template. (String)

-templateLocation
Specifies a configuration Id that represents the location to place the template. (String)

The following example displays the format of the configuration Id, where the display name is optional:

```
Websphere:_Websphere_Config_Data_Display_Name=display_name,_Websphere_Config_Data_Id=configuration_id
```

The configuration Id value is the Webserver template, which is `templates\servertypes\WEB_SERVER|servertype-metadata.xml`

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createWebServerTemplate {-templateName newTemplate -serverName
 server1  -nodeName ndnode1 -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=WEB_SERVER,_Websphere_Config_Data_Id=
 templates/servertypes/WEB_SERVER|servertype-metadata.xml}
```

- Using Jython string:

```
AdminTask.createWebServerTemplate('[-templateName newTemplate -serverName
 server1 -nodeName ndnode1  -description "This is my new template" -templateLocation
 Websphere:_Websphere_Config_Data_Display_Name=
 WEB_SERVER,_Websphere_Config_Data_Id=templates/servertypes/WEB_SERVER|servertype-metadata.xml]')
```

- Using Jython list:

```
AdminTask.createWebServerTemplate(['-templateName',  'newTemplate', '-serverName',
 'server1', '-nodeName',  'ndnode1', '-description', "This is my new template"])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createWebServerTemplate {-interactive}
```

- Using Jython string:

```
AdminTask.createWebServerTemplate ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createWebServerTemplate (['-interactive'])
```

deleteServer

Use the **deleteServer** command to delete a server.

Target object

None

Required parameters

-serverName
The name of the server to delete. (String, required)

-nodeName
The name of the node for the server that you want to delete. (String, required)

Optional parameters

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteServer {-serverName server_name -nodeName node_name}
```

- Using Jython string:

```
AdminTask.deleteServer('[-serverName server_name -nodeName node_name]')
```

- Using Jython list:

```
AdminTask.deleteServer(['-serverName', 'server_name', '-nodeName',
 'node_name'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteServer {-interactive}
```

- Using Jython string:

```
AdminTask.deleteServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteServer (['-interactive'])
```

deleteServerTemplate

Use the **deleteServerTemplate** command to delete a server template. You cannot delete templates that are defined by the system. You can only delete server templates that you created. This command deletes the directory that hosts the server template.

Target object

The name of the template to delete. (ObjectName, required)

Required parameters

None.

Optional parameters

None.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask deleteServerTemplate
 template_name(templates/serverTypes/APPLICATION_SERVER/servers /newTemplate|server.xml
#Server_1105015708079)
```

- Using Jython string:

```
AdminTask.deleteServerTemplate('template_name(templates/serverTypes/APPLICATION_SERVER/servers
 /newTemplate|server.xml#Server_1105015708079)')
```

- Using Jython list:

```
AdminTask.deleteServerTemplate('template_name(templates/serverTypes/APPLICATION_SERVER/servers
 /newTemplate|server.xml#Server_1105015708079)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask deleteServerTemplate {-interactive}
```

- Using Jython string:

```
AdminTask.deleteServerTemplate ('[-interactive]')
```

- Using Jython list:

```
AdminTask.deleteServerTemplate (['-interactive'])
```

getJavaHome

Use the **getJavaHome** command to get the Java home value.

Target object

None.

Required parameters

-serverName
Specifies the name of the server. (String, required)

-nodeName
Specifies the name of the node. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getJavaHome {-nodeName mynode -serverName myserver}
```

- Using Jython string:

```
AdminTask.getJavaHome ('[-nodeName mynode -serverName myserver]')
```

- Using Jython list:

```
AdminTask.getJavaHome (['-nodeName' 'mynode' '-serverName' 'myserver'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getJavaHome {-interactive}
```

- Using Jython string:

```
AdminTask.getJavaHome ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getJavaHome (['-interactive'])
```

getServerType

The **getServerType** command returns the type of the server that you specify.

Target object

None

Optional parameters

-serverName
The name of the server. (String)

-nodeName
The name of the node. (String)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getServerType {-serverName test2 -nodeName ndnode1}
```

- Using Jython string:

```
AdminTask.getServerType('[-serverName test2 -nodeName ndnode1]')
```

- Using Jython list:

```
AdminTask.getServerType(['-serverName', 'test2', '-nodeName', 'ndnode1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getServerType {-interactive}
```

- Using Jython string:

```
AdminTask.getServerType ('[-interactive]')
```

- Using Jython list:

```
AdminTask.getServerType (['-interactive'])
```

listServers

The **listServers** command returns a list of servers.

Target object

None

Optional parameters

serverType
Specifies the type of the server. Used to filter the results. (String, optional)

nodeName
Specifies the name of the node. Used to filter the results. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listServers {-serverType APPLICATION_SERVER -nodeName ndnode1}
```

- Using Jython string:

```
AdminTask.listServers('[-serverType APPLICATION_SERVER -nodeName ndnode1]')
```

- Using Jython list:

```
AdminTask.listServers(['-serverType', 'APPLICATION_SERVER',  '-nodeName',
 'ndnode1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listServers {-interactive}
```

- Using Jython string:

```
AdminTask.listServers ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listServers (['-interactive'])
```

listServerTemplates

Use the **listServerTemplates** command to list server templates.

Target object

None

Optional parameters

-version
The version of the template that you want to list. (String, optional)

-serverType
Specify this option if you want to list templates for a specific server type. (String, optional)

-name
Specify this option to look for a specific template. (String, optional)

-queryExp
A key and value pair that you can use to find templates by properties. For example, `com.ibm.websphere.nodeOperatingSystem=os390` (String[], optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listServerTemplates {-version 6.0.0.0 -serverType
 APPLICATION_SERVER}
```

- Using Jython string:

```
AdminTask.listServerTemplates('[-version 6.0.0.0 -serverType
 APPLICATION_SERVER]')
```

- Using Jython list:

```
AdminTask.listServerTemplates(['-version', '6.0.0.0',  '-serverType',
 'APPLICATION_SERVER'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listServerTemplates {-interactive}
```

- Using Jython string:

```
AdminTask.listServerTemplates ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listServerTemplates (['-interactive'])
```

listServerTypes

Use the **listServerTypes** command to display all the current server types. For example, `APPLICATION_SERVER, WEB_SERVER, GENERIC_SERVER`

Target object

The node name for which you want to list the valid types. For example, the types that are only valid on z/OS will appear on a z/OS node. (String, optional)

Parameters and return values

- Parameters: None
- Returns: A list of server types that you can define on a node. If you do not specify the target object, this command returns all of the server types defined in the entire cell.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listServerTypes ndnode1
```

- Using Jython string:

```
AdminTask.listServerTypes(ndnode1)
```

- Using Jython list:

```
AdminTask.listServerTypes(ndnode1)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listServerTypes {-interactive}
```

- Using Jython string:

```
AdminTask.listServerTypes ('[-interactive]')
```

- Using Jython list:

```
AdminTask.listServerTypes (['-interactive'])
```

setJVMDebugMode

Use the **setJVMDebugMode** command to set the Java virtual machine (JVM) debug mode for the application server.

Target object

None

Required parameters

-serverName
The name of the server whose JVM properties will be modified. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-debugMode
Specifies whether to run the JVM in debug mode. The default is not to enable debug mode. (Boolean, required)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMDebugMode {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMDebugMode ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMDebugMode (['-interactive'])
```

setGenericJVMArguments

Use the **setGenericJVMArguments** command passes command line arguments to the Java virtual machine (JVM) code that starts the application server process.

Target object

None

Required parameters

-serverName
Specifies the name of the server that contains the JVM properties that are modified. If only one server exists in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

Optional parameters

-genericJvmArguments
Specifies that the command line arguments pass to the Java virtual machine code that starts the application server process. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setGenericJVMArguments {-serverName server1 -nodeName node1}
```

- Using Jython string:

```
AdminTask.setGenericJVMArguments('[-serverName server1 -nodeName node1]')
```

- Using Jython list:

```
AdminTask.setGenericJVMArguments(['-serverName', 'server1',  '-nodeName',
 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setGenericJVMArguments {-interactive}
```

- Using Jython string:

```
AdminTask.setGenericJVMArguments ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setGenericJVMArguments (['-interactive'])
```

setJVMInitialHeapSize

Use the **setJVMInitialHeapSize** command to set the Java Virtual Machine (JVM) initial heap size for the application server.

Target object

None

Parameters and return values

-serverName
The name of the server whose JVM properties are modified. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-initialHeapSize
Specifies the initial heap size available to the JVM code, in megabytes. (Integer, required)

Examples

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMInitialHeapSize {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMInitialHeapSize ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMInitialHeapSize (['-interactive'])
```

setJVMMaxHeapSize

Use the **setJVMMaxHeapSize** command to set the Java virtual machine (JVM) maximum heap size for the application server.

Target object

None

Parameters and return values

-serverName
The name of the server whose JVM properties are modified. If there is only one server in the configuration, (String, required)

-nodeName
The node name where the server locates. If the server name is unique in the cell, this parameter is optional. (String, required)

-maximumHeapSize
Specifies the maximum heap size available to the JVM code, in megabytes. (Integer, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMMaxHeapSize {-serverName server1 -nodeName node1 -maximumHeapSize
 10}
```

Configuration note: With the Jacl scripting language, the *subst* command enables you to substitute a previously set value for a variable in the command. For example, you can set the JVM maximum heap size using the command commands:

```
set nodeparm "node1"
$AdminTask setJVMMaxHeapSize [subst {-serverName server1 -nodeName $nodeparm
-maximumHeapSize 100}]
```

- Using Jython string:

```
AdminTask.setJVMMaxHeapSize('[-serverName server1 -nodeName node1 -maximumHeapSize
 10]')
```

- Using Jython list:

```
AdminTask.setJVMMaxHeapSize(['-serverName', 'server1', '-nodeName', 'node1',
 '-maximumHeapSize', '10'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMMaxHeapSize {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMMaxHeapSize ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMMaxHeapSize (['-interactive'])
```

setJVMProperties

Use the **setJVMProperties** command to set the Java virtual machine (JVM) configuration for the application server.

Target object

None

Required parameters

-serverName
Specifies the name of the server for which the JVM properties will be modified. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the entire cell, this parameter is optional. (String, required)

Optional parameters

-classpath
Specifies the standard class path in which the Java virtual machine (JVM) code looks for classes. (String, optional)

-bootClasspath
Bootstrap classes and resources for JVM code. This option is only available for JVM instructions that support bootstrap classes and resources. You can separate multiple paths by a colon (:) or semi-colon (;), depending on the operating system of the node. (String, optional)

-verboseModeClass
Specifies whether to use verbose debug output for class loading. The default is not to enable verbose class loading. (Boolean, optional)

-verboseModeGarbageCollection
Specifies whether to use verbose debug output for garbage collection. The default is not to enable verbose garbage collection. (Boolean, optional)

-verboseModeJNI
Specifies whether to use verbose debug output for native method invocation. The default is not to enable verbose Java Native Interface (JNI) activity. (Boolean, optional)

-initialHeapSize
Specifies the initial heap size in megabytes that is available to the JVM code. (Integer, optional)

-maximumHeapSize
Specifies the maximum heap size available in megabytes to the JVM code. (Integer, optional)

-runHProf
This parameter only applies to WebSphere Application Server version. It specifies whether to use HProf profiler support. To use another profiler, specify the custom profiler settings using the hprofArguments parameter. The default is not to enable HProf profiler support. (Boolean, optional)

-hprofArguments
This parameter only applies to WebSphere Application Server version. It specifies command-line profiler arguments to pass to the JVM code that starts the application server process. You can specify arguments when HProf profiler support is enabled. (String, optional)

-debugMode
Specifies whether to run the JVM in debug mode. The default is not to enable debug mode support. (Boolean, optional)

-debugArgs
Specifies the command line debug arguments to pass to the JVM code that starts the application server process. You can specify arguments when the debug mode is enabled. (String, optional)

-genericJvmArguments
Specifies the command line arguments to pass to the JVM code that starts the application server process. (String, optional)

-executableJarFileName
Specifies a full path name for an executable JAR file that the JVM code uses. (String, optional)

-disableJIT
Specifies whether to disable the just in time (JIT) compiler option of the JVM code. (Boolean, optional)

-osName
Specifies the JVM settings for a given operating system. When started, the process uses the JVM settings for the operating system of the node. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMProperties {-serverName server1 -nodeName node1}
```

- Using Jython string:

```
AdminTask.setJVMProperties('[-serverName server1 -nodeName node1]')
```

- Using Jython list:

```
AdminTask.setJVMProperties(['-serverName', 'server1', '-nodeName', 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMProperties {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMProperties (['-interactive'])
```

setJVMSystemProperties

Use the **setJVMSystemProperties** command to set the Java virtual machine (JVM) system property for the process of the application server.

Target object

None

Required parameters

-serverName
Specifies the name of the server whose JVM system properties will be set. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-propertyName
Specifies the property name. (String, required)

-propertyValue
Specifies the property value. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setJVMSystemProperties {-serverName server1 -nodeName node1
 -propertyName  test.property -propertyValue testValue}
```

- Using Jython string:

```
AdminTask.setJVMSystemProperties('[-serverName server1 -nodeName node1
 -propertyName  test.property -propertyValue testValue]')
```

- Using Jython list:

```
AdminTask.setJVMSystemProperties(['-serverName', 'server1',  '-nodeName', 'node1',
 '-propertyName', 'test.property',  '-propertyValue', 'testValue'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setJVMSystemProperties {-interactive}
```

- Using Jython string:

```
AdminTask.setJVMSystemProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setJVMSystemProperties (['-interactive'])
```

setProcessDefinition

Use the **setProcessDefinition** command to set the process definition of an application server.

Target object

None

Required parameters

-serverName
The name of the server for which you want to modify the process definition. If there is only one server in the entire configuration, this parameter is optional. (String, required)

-nodeName
The node name where the server resides. If the server name is unique in the entire cell, this parameter is optional. (String, required)

Optional parameters

-executableName
Specifies the executable name that is invoked to start the process. This parameter is only applicable to WebSphere Application Server version. (String, optional)

-executableArguments
Specifies the arguments that are passed to the process when it is started. This parameter is only applicable to WebSphere Application Server version. (String, optional)

-workingDirectory
Specifies the file system directory that the process uses for the current working directory. (String, optional)

-executableTargetKind
Specifies the type of the executable target. Valid values include JAVA_CLASS and EXECUTABLE JAR. (String, optional)

-executableTarget
Specifies the name of the executable target. The executable target is a Java class containing a main() method, or the name of an executable JAR file. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setProcessDefinition {-serverName server1 -nodeName node1}
```

- Using Jython string:

```
AdminTask.setProcessDefinition('[-serverName server1 -nodeName node1]')
```

- Using Jython list:

```
AdminTask.setProcessDefinition(['-serverName', 'server1', '-nodeName',
 'node1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setProcessDefinition {-interactive}
```

- Using Jython string:

```
AdminTask.setProcessDefinition ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setProcessDefinition (['-interactive'])
```

setTraceSpecification

Use the **setTraceSpecification** command to set the trace specification for the server. If the server is running new trace specification the change takes effect immediately. This command also saves the trace specification in configuration.

Target object

None

Required parameters

-serverName
Specifies the name of the server whose trace specification will be set. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

-traceSpecification
Specifies the trace specification. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask setTraceSpecification {-serverName server1 -nodeName node1
 -traceSpecification  com.ibm.*=all=enabled}
```

- Using Jython string:

```
AdminTask.setTraceSpecification('[-serverName server1 -nodeName node1
 -traceSpecification  com.ibm.*=all=enabled]')
```

- Using Jython list:

```
AdminTask.setTraceSpecification(['-serverName', 'server1',  '-nodeName', 'node1',
 '-traceSpecification',  'com.ibm.*=all=enabled'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask setTraceSpecification {-interactive}
```

- Using Jython string:

```
AdminTask.setTraceSpecification ('[-interactive]')
```

- Using Jython list:

```
AdminTask.setTraceSpecification (['-interactive'])
```

showJVMProperties

Use the **showJVMProperties** command to list the Java virtual machine (JVM) configuration for the server of the application process.

Target object

None

Required parameters

-serverName
Specifies the name of the Server whose JVM properties are shown. If there is only one server in the entire configuration, then this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server locates. If the server name is unique in the entire cell, then this parameter is optional. (String, required)

-propertyName
If you specify this parameter, the value of this property is returned. If you do not specify this parameter, all JVM properties will return in list format. Each element in the list is a property name and value pair. (String, optional)

Optional parameters

-propertyName
If you specify this parameter, the value of this property is returned. If you do not specify this parameter, all JVM properties will return in list format. Each element in the list is a property name and value pair. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showJVMProperties {-serverName server1 -nodeName node1  -propertyName
 test.property}
```

- Using Jython string:

```
AdminTask.showJVMProperties('[-serverName server1 -nodeName node1  -propertyName
 test.property]')
```

- Using Jython list:

```
AdminTask.showJVMProperties(['-serverName', 'server1', '-nodeName', 'node1',
 '-propertyName', 'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showJVMProperties {-interactive}
```

- Using Jython string:

```
AdminTask.showJVMProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showJVMProperties (['-interactive'])
```

showJVMSystemProperties

Use the **showJVMSystemProperties** command to show the Java virtual machine (JVM) system properties for the process of the application server.

Target object

None

Required parameters

-serverName
Specifies the name of the server whose JVM properties will be shown. If there is only one server in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

Optional parameters

-propertyName
If you specify this parameter, the value of specified property is returned. If you do not specify this parameter, all properties will return in a list where each element is a property name and value pair. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showJVMSystemProperties {-serverName server1 -nodeName node1
 -propertyName test.property}
```

- Using Jython string:

```
AdminTask.showJVMSystemProperties('[-serverName server1 -nodeName node1
 -propertyName test.property]')
```

- Using Jython list:

```
AdminTask.showJVMSystemProperties(['-serverName', 'server1',  '-nodeName', 'node1',
 '-propertyName',  'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showJVMSystemProperties {-interactive}
```

- Using Jython string:

```
AdminTask.showJVMSystemProperties ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showJVMSystemProperties (['-interactive'])
```

showProcessDefinition

Use the **showProcessDefinition** command to show the process definition of the server.

Target object

None

Required parameters

-serverName
Specifies the name of the server for which the process definition is shown. If only one server exists in the configuration, this parameter is optional. (String, required)

-nodeName
Specifies the node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required)

Optional parameters

-propertyName
If you do not specify this parameter, all the process definitions of the server are returned in a list format where each element in the list is property name and value pair. If you specify this parameter, the property value of the property name that you specified is returned. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showProcessDefinition {-serverName server1 -nodeName node1 -propertyName
 test.property}
```

- Using Jython string:

```
AdminTask.showProcessDefinition('[-serverName server1 -nodeName node1 -propertyName
 test.property]')
```

- Using Jython list:

```
AdminTask.showProcessDefinition(['-serverName', 'server1',  '-nodeName', 'node1',
 '-propertyName',  'test.property'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showProcessDefinition {-interactive}
```

- Using Jython string:

```
AdminTask.showProcessDefinition ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showProcessDefinition (['-interactive'])
```

showServerInfo

The **showServerInfo** command returns the information for a server that you specify.

Target object

The configuration ID of the server. (required)

Parameters and return values

- Parameters: None
- Returns: A list of metadata.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showServerInfo server1(cells/WAS00Network
 /nodes/ndnode1/servers/server1|server.xml)
```

- Using Jython string:

```
AdminTask.showServerInfo('server1(cells/WAS00Network
 /nodes/ndnode1/servers/server1|server.xml)')
```

- Using Jython list:

```
AdminTask.showServerInfo('server1(cells/WAS00Network
 /nodes/ndnode1/servers/server1|server.xml)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showServerInfo {-interactive}
```

- Using Jython string:

```
AdminTask.showServerInfo ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showServerInfo (['-interactive'])
```

showServerTypeInfo

The **showServerTypeInfo** command displays information about a specific server type.

Target object

Specifies a server type. For example: `APPLICATION_SERVER` (String, required)

Optional parameters

-version
Specifies the version of the templates that you want to list. For example, 6.0.0.0. (String, optional)

-serverType
Specifies if you want to list templates for a specific server type. (String, optional)

-name
Specifies whether to look for a specific template. (String, optional)

-queryExp
Specifies a key and value pair that you can use to find templates by properties. For example, com.ibm.websphere.nodeOperatingSystem=os390. (String[], optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showServerTypeInfo APPLICATION_SERVER
```

- Using Jython string:

```
AdminTask.showServerTypeInfo(APPLICATION_SERVER)
```

- Using Jython list:

```
AdminTask.showServerTypeInfo(APPLICATION_SERVER)
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showServerTypeInfo {-interactive}
```

- Using Jython string:

```
AdminTask.showServerTypeInfo ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showServerTypeInfo (['-interactive'])
```

showTemplateInfo

Use the **showTemplateInfo** command to display the metadata information for a specific server template.

Target object

Specifies the configuration Id of the server type. (String, required)

Parameters and return values

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask showTemplateInfo
 default(templates/servertypes/APPLICATION_SERVER/servers/default|server.xml)  {isSystemTemplate true}
{name default}  {com.ibm.websphere.baseProductVersion 6.0.0}
{description {The WebSphere  Default Server Template}}
 {com.ibm.websphere.baseProductMinorVersion 0.0} {com.ibm.websphere.baseProductMajorVersion 6}
 {com.ibm.websphere.nodeOperatingSystem {}}{isDefaultTemplate true}
```

- Using Jython string:

```
AdminTask.showTemplateInfo(default(templates/serverTypes/APPLICATION_SERVER/servers/default|server.xml))
 '[[isSystemTemplate  true] [com.ibm.websphere.baseProductVersion 6.0.0] [name default]
[com.ibm.websphere.baseProductMinorVersion 0.0] [description  The WebSphere Default Server Template]
  [isDefaultTemplate true] [com.ibm.websphere.nodeOperatingSystem] [com.ibm.websphere.baseProductMajorVersion 6]]'
```

- Using Jython list:

```
AdminTask.showTemplateInfo(default(templates/serverTypes/APPLICATION_SERVER/servers/default|server.xml))
 [['isSystemTemplate',  'true'], ['com.ibm.websphere.baseProductVersion', '6.0.0'], ['name', 'default']
 ['com.ibm.websphere.baseProductMinorVersion',  '0.0'], ['description', 'The WebSphere
Default Server Template']  ['isDefaultTemplate', 'true'], ['com.ibm.websphere.nodeOperatingSystem'],
  ['com.ibm.websphere.baseProductMajorVersion',  '6']]
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask showTemplateInfo {-interactive}
```

- Using Jython string:

```
AdminTask.showTemplateInfo ('[-interactive]')
```

- Using Jython list:

```
AdminTask.showTemplateInfo (['-interactive'])
```

UnmanagedNodeCommands command group for the AdminTask object using wsadmin scripting

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The commands and parameters in the UnmanagedNodeCommands group can be used to create and query for managed and unmanaged nodes. An unmanaged node is a node that does not have a node agent or a deployment manager.

The UnmanagedNodeCommands command group for the AdminTask object includes the following commands:

- "createUnmanagedNode"
- "listManagedNodes" on page 547
- "listUnmanagedNodes" on page 547
- "removeUnmanagedNode" on page 548

createUnmanagedNode

Use the **create Unmanaged Node** command to create a new unmanaged node in the configuration. An unmanaged node is a node that does not have a node agent or a deployment manager. Unmanaged nodes can contain web servers, such as IBM HTTP Server.

Target object

None

Parameters and return values

-nodeName
The name that will represent the node in the configuration repository. (String, required)

-hostName
The host name of the system associated with this node. (String, required)

-nodeOperatingSystem
The operating system in use on the system associated with this node. Valid entries include the following: `os400`, `aix`, `hpux`, `linux`, `solaris`, `windows`, and `os390`.(String required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createUnmanagedNode {-nodeName myNode -hostName myHost
-nodeOperatingSystem linux}
```

- Using Jython string:

```
AdminTask.createUnmanagedNode ('[-nodeName jjNode -hostName jjHost
-nodeOperatingSystem linux]')
```

- Using Jython list:

```
AdminTask.createUnmanagedNode (['-nodeName', 'jjNode', '-hostName', 'jjHost',
'-nodeOperatingSystem', 'linux'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createUnmanaged Node {-interactive}
```

- Using Jython string:

```
AdminTask.createUnmanaged Node ('[-interactive]')
```

- Using Jython list:

```
AdminTask.createUnmanaged Node (['-interactive'])
```

listManagedNodes

Use the **listManaged Nodes** command to list the managed nodes, nodes that have a node agent defined, in a configuration.

Target object

None

Parameters and return values

- Parameters: None
- Returns: List

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listManagedNodes
```

- Using Jython string:

```
AdminTask.listManagedNodes()
```

- Using Jython list:

```
AdminTask.listManagedNodes()
```

listUnmanagedNodes

Use the **list Unmanaged Nodes** command to list the unmanaged nodes in a configuration.

Target object

None

Parameters and return values

- Parameters: None
- Returns: List

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listUnmanagedNodes
```

- Using Jython string:

```
AdminTask.listUnmanagedNodes()
```

- Using Jython list:

```
AdminTask.listUnmanagedNodes()
```

Interactive mode example usage:

- Using Jacl:
  ```
  $AdminTask listUnmanagedNodes {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.listUnmanagedNodes ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.listUnmanagedNodes (['-interactive'])
  ```

removeUnmanagedNode

Use the **remove Unmanaged Node** command to remove an unmanaged node from the configuration.

Target object

None

Parameters and return values

-nodeName
The name of the unmanaged node. (String, required)

Examples

Batch mode example usage:
- Using Jacl:
  ```
  $AdminTask removeUnmanagedNode {-nodeName myNode }
  ```
- Using Jython string:
  ```
  AdminTask.removeUnmanagedNode('[-nodeName myNode]')
  ```
- Using Jython list:
  ```
  AdminTask.removeUnmanagedNode(['-nodeName', 'myNode'])
  ```

Interactive mode example usage:
- Using Jacl:
  ```
  $AdminTask removeUnmanagedNode {-interactive}
  ```
- Using Jython string:
  ```
  AdminTask.removeUnmanagedNode ('[-interactive]')
  ```
- Using Jython list:
  ```
  AdminTask.removeUnmanagedNode (['-interactive'])
  ```

ConfigArchiveOperations command group for the AdminTask object using wsadmin scripting

You can use the Jython or Jacl scripting languages to configure servers in your environment. The commands and parameters in the ConfigArchiveOperations group can be used to export or import server configurations and entire cell configurations.

The ConfigArchiveOperations command group for the AdminTask object includes the following commands:
- “exportProxyProfile” on page 549
- “exportProxyServer” on page 549
- “exportServer” on page 550
- “exportWasprofile” on page 551
- “importProxyProfile” on page 552

- “importProxyServer” on page 552
- “importServer” on page 554
- “importWasprofile” on page 555

exportProxyProfile

Use the exportProxyProfile command to export the entire cell configuration of a secure proxy server to a configuration archive. The exportProxyProfile command does not work between the distributed and z/OS operating systems.

Target object

None.

Required parameters

-archive
Specifies the fully qualified file path of the exported configuration archive. (String, required)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:
  ```
  AdminTask.exportProxyProfile('-archive c:/myCell.car')
  ```
- Using Jython list:
  ```
  AdminTask.exportProxyProfile(['-archive', 'c:/myCell.car'])
  ```

Interactive mode example usage

- Using Jython:
  ```
  AdminTask.exportProxyProfile('-interactive')
  ```

exportProxyServer

Use the exportProxyServer command to export the secure proxy server configuration to a node that is defined in the configuration archive. The command exports the metadata file of the node where the server resides. You can use this information later when you import the configuration archive to verify that the target node is compatible to the node from which you are exporting the server.

The exportProxyServer command virtualizes the server configuration and exports a server to a configuration archive. This process breaks any existing associations between the server configurations in the configuration archive and the configurations in the system.

Target object

None

Required parameters

-archive
Specifies the fully qualified path of the exported configuration archive. (String, required)

-serverName
Specifies the secure proxy server name. (String, required)

Optional parameters

-nodeName
Specifies the node name of the secure proxy server. This parameter is only required if the secure proxy server name is not unique across the cell. (String, optional)

Return value

The command does not return output.

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.exportProxyServer('[-archive c:/myProxyServer.car -nodeName node1
 -serverName server1]')
```

- Using Jython list:

```
AdminTask.exportProxyServer(['-archive', 'c:/myProxyServer.car', '-nodeName',
 'node1', '-serverName', 'server1'])
```

- Using Jython string:

```
AdminTask.exportProxyServer('[-archive /myProxyServer.car -nodeName node1
 -serverName server1]')
```

- Using Jython list:

```
AdminTask.exportProxyServer(['-archive', '/myProxyServer.car', '-nodeName', 'node1',
 '-serverName', 'server1'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.exportServer ('-interactive')
```

exportServer

Use the exportServer command to export the server configuration to a node that is defined in the configuration archive. Use the exportProxyServer command to export a proxy server configuration.

The exportServer command virtualizes the server configuration and exports a server to a configuration archive. This process breaks any existing associations between the server configurations in the configuration archive and the configurations in the system. This process also removes applications from the server that you specify, breaks the relationship between the server that you specify and the core group of the server, cluster, or service integration bus member.

The exportServer command exports the metadata file of the node where the server resides. You can use this information later when you import the configuration archive to verify that the target node is compatible to the node from which you are exporting the server.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified path of the exported configuration archive. (String, required)

-nodeName
Specifies the node name of the server. This parameter is only required when the server name is not unique across the cell. (String, optional)

-serverName
Specifies the server name. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask exportServer {-archive c:/myServer.car -nodeName node1 -serverName
 server1}
```

- Using Jython string:

```
AdminTask.exportServer ('[-archive c:/myServer.car -nodeName node1 -serverName
 server1]')
```

- Using Jython list:

```
AdminTask.exportServer (['-archive', 'c:/myServer.car', '-nodeName', 'node1',
 '-serverName', 'server1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask exportServer {-interactive}
```

- Using Jython string:

```
AdminTask.exportServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.exportServer (['-interactive'])
```

exportWasprofile

Use the exportWasprofile command to export the entire cell configuration to a configuration archive. The exportWasprofile command does not work between the distributed and z/OS platforms.

Restriction: Only a base server configuration with a single node is supported for the exportWasprofile command.
Use the exportProxyProfile command to export a secure proxy server configuration.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified file path of the exported configuration archive. (String, required)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask exportWasprofile {-archive c:/myCell.car}
```

- Using Jython string:

```
AdminTask.exportWasprofile('[-archive c:/myCell.car]')
```

- Using Jython list:

```
AdminTask.exportWasprofile(['-archive', 'c:/myCell.car'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask exportWasprofile {-interactive}
```

- Using Jython string:

```
AdminTask.exportWasprofile ('[-interactive]')
```

- Using Jython list:

```
AdminTask.exportWasprofile (['-interactive'])
```

importProxyProfile

Use the importProxyProfile command to import a cell configuration in the configuration archive to the system. Only a base single server configuration is supported for this command. The importProxyProfile command does not work between the distributed and z/OS platforms.

Target object

None.

Required parameters

-archive
Specifies the fully qualified file path of the exported configuration archive. (String, required)

Optional parameters

-deleteExistingServers
Specifies whether to replace the existing secure proxy servers in the profile with the servers in the imported proxy profile. Specify `true` to overwrite the existing servers. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.importProxyProfile('-archive /myCell.car -deleteExistingServers
 true')
```

- Using Jython list:

```
AdminTask.importProxyProfile('-archive', '/myCell.car', '-deleteExistingServers', 'true')
```

Interactive mode example usage

- Using Jython:

```
AdminTask.importProxyProfile('-interactive')
```

importProxyServer

Use the importProxyServer command to import a secure proxy server that resides in a configuration archive to the system. This command imports all the server scope configurations defined in the configuration archive to system configuration.

Target object

None

Required parameters

-archive
Specifies the fully qualified path of the configuration archive to import. (String, required)

Optional parameters

-nodeInArchive
Specifies the node name of the server defined in the configuration archive. Specify a value for this parameter if multiple nodes exist in the configuration archive. (String, optional)

-serverInArchive
Specifies the name of the secure proxy server defined in the configuration archive. Specify a value for this parameter if multiple secure proxy servers exist in the archive. (String, optional)

-deleteExistingServer
Specifies whether to delete and replace an existing server if it has the same name as the server to import. Set the value of this command to `true` to overwrite existing servers with the same name. (String, optional)

-nodeName
Specifies the name of the node to which the secure proxy server is imported. This parameter is only required if the secure proxy server name is not unique across the cell. (String, optional)

-serverName
Specifies the secure proxy server name. If the server name that you specify matches an existing server name under the node, an exception is created. (String, optional)

-coreGroup
Specifies the core group name to which the secure proxy server belongs. (String, optional)

Return value

The command does not return output.

Examples

Batch mode example usage:

- Using Jython string:

```
AdminTask.importProxyServer('[-archive c:/myProxyServer.car -nodeName node1
 -serverInArchive server1 -deleteExistingServer true]')
```

- Using Jython list:

```
AdminTask.importProxyServer(['-archive', 'c:/myProxyServer.car', '-nodeName',
 'node1', '-serverInArchive', 'server1', '-deleteExistingServer', 'true'])
```

- Using Jython string:

```
AdminTask.importProxyServer('[-archive /myProxyServer.car -nodeName node1
 -serverInArchive server1 -deleteExistingServer true]')
```

- Using Jython list:

```
AdminTask.importProxyServer(['-archive', '/myProxyServer.car', '-nodeName', 'node1',
 '-serverInArchive', 'server1', 'server1', '-deleteExistingServer', 'true'])
```

Interactive mode example usage:

- Using Jython:

```
AdminTask.importProxyServer('-interactive')
```

importServer

Use the importServer command to import a server that resides in a configuration archive to the system. This command imports all the server scope configurations defined in the configuration archive to system configuration. Use the importProxyServer command to import a secure proxy server configuration.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified path of the configuration archive. (String, required)

-nodeInArchive
Specifies the node name of the server defined in the configuration archive. (String, optional if there is only one node defined in the configuration archive, required if there are multiple nodes defined in the configuration archive)

-serverInArchive
Specifies the name of the server defined in the configuration archive. (String, optional if there is only one server defined on the specified *nodeInConfiguration* archive, required if there are multiple servers defined under the specified *nodeInConfiguration* archive)

-nodeName
Specifies the node name where the server is imported. (String, optional if there is only one node)

-serverName
Specifies the server name where the server is imported. If the server name that you specify matches an existing server name under the node, an exception is created. (String, optional, default: serverInArchive)

-coreGroup
Specifies the core group name to which the server should belong. (String, optional)

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask importServer {-archive c:/myServer.car -nodeInArchive node1
 -serverInArchive server1}
```

- Using Jython string:

```
AdminTask.importServer ('[-archive c:/myServer.car -nodeInArchive node1
 -serverInArchive server1]')
```

- Using Jython list:

```
AdminTask.importServer (['-archive', 'c:/myServer.car', '-nodeInArchive', 'node1',
 '-serverInArchive', 'server1'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask importServer {-interactive}
```

- Using Jython string:

```
AdminTask.importServer ('[-interactive]')
```

- Using Jython list:

```
AdminTask.importServer (['-interactive'])
```

importWasprofile

Use the importWasprofile command to import a cell configuration in the configuration archive to the system. Only a base server configuration with single node is supported for this command. Use the importProxyProfile command to import a secure proxy server profile.

The importWasprofile command does not work between distributed and z/OS operating systems.

Target object

None

Parameters and return values

-archive
Specifies the fully qualified file path of the configuration archive. (String, required)

-deleteExistingServers
When set to `true`, specifies to remove existing servers from the target profile and import the configuration archive onto the target profile. (Boolean, optional)

Default value is `false`, which specifies to not replace servers.

Examples

Batch mode example usage:

Interactive mode example usage:

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
The default Application Client for IBM WebSphere Application Server profile root is the `/QIBM/UserData/WebSphere/AppClient/V8/client/profiles/`*`profile_name`* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppServer/V8/Base` directory.

java_home

Table 450. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit
64–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit

plugins_profile_root
The default Web Server Plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/*profile_name* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V8/webserver directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V8/webserver directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS8*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is QWAS8A. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, was.install.library, and is located under the *app_server_root* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/*profile_name* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is QWAS8. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V8/Base directory.

The profiles and profileRegistry subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Chapter 18. Using properties files to manage system configuration

Use the wsadmin tool and properties files to administer your administrative architecture and runtime settings.

About this task

You can use properties files to manage your environment and configuration objects. You can extract configuration objects in simple properties file format, modify the extracted properties file, and apply the modified properties file to update the system configuration.

In addition to updating system properties, you can do the following:
- Extract properties required to run an administrative command.
- Run an administrative command using an extracted properties file.
- Extract or modify properties for any WCCM object type.
- Extract or modify all properties of an object type.
- Delete or remove a property and modify a property using a single properties file.
- Delete a configuration object in the same properties file that is used to create or modify properties.

A properties file extracted from a configuration contains the following information about the configuration:
- Required properties for creating a new object of any type.
- Default values for a property.
- Range of values for a property.

Properties files are portable. You can extract a properties file from one cell, modify some environment-specific variables at the bottom of the extracted properties file, and then apply the modified properties file to another cell.

To use non-English, localized special characters in properties files, save the properties files using UTF-8 encoding. Using any other encoding might cause errors when the product resolves the non-English resources during properties file validation or cause incorrect value changes in the configuration files after applying the properties files.

Procedure
- Manage environment configurations with properties files.
- Extract properties files.
- Validate properties files.
- Apply properties files.
- Run administrative commands using properties files.
- Extract or modify properties for a WCCM object type.
- Apply portable properties files across multiple environments.
- Manage specific configuration objects with properties files.

What to do next

Save the changes to your configuration.

Managing environment configurations with properties files using wsadmin scripting

Use this topic to modify your environment using properties files. You can use the wsadmin tool to generate, validate and apply properties files in your application server, profile, node, or other resource configurations.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can extract the configuration attributes and values from your environment to properties files. You can use this functionality for various purposes, including:

- To modify your existing configuration in one location, instead of configuring multiple administrative console panels or running many commands
- To improve the application development life cycle

You can use this topic to manage the following resources in your environment:

- Application servers
- Nodes
- Profiles
- Virtual hosts
- Authorization tables
- Data replication domains
- Variable maps
- Java™ Database Connectivity (JDBC) providers
- Uniform Resource Locator (URL) providers
- Mail providers
- Resource environment providers
- Java 2 Connector (J2C) resource adapters

Complete the following steps to extract a properties file for an application server, edit the properties, and apply them to your configuration. You can also use interactive mode with these commands, as the following syntax demonstrates:

```
AdminTask.commandName('-interactive')
```

Procedure

Modify an application server configuration, and apply the changes using a properties file.

1. Launch the wsadmin tool.
2. Extract the application server configuration to modify.

   Use the extractConfigProperties command to extract the object configuration, as the following Jython example demonstrates:

```
AdminTask.extractConfigProperties('-propertiesFileName ConfigProperties_server1.props
-configData Server=server1')
```

   The system extracts the properties file, which contains each of the configuration objects and attributes for the `server1` application server.
3. Open the properties file, and manually edit the attribute values of interest.

   Note: Because you are manually editing the properties file, make a back-up copy of the properties file before you edit it.

The following sample is a section of an application server properties file:

```
#
# Configuration properties file for cells/myCell/nodes/myNode/servers/server1|server.xml#
# Extracted on Thu Sep 06 00:27:26 CDT 2007
#
#
# Section 1.0 ## cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
# SubSection 1.0 # Server Section
#
ResourceType=Server
ImplementingResourceType=Server
ResourceId=cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
#Properties
#
shortName=null
serverType=APPLICATION_SERVER
developmentMode=false #boolean
name=server1
parallelStartEnabled=true #boolean
clusterName=C
modelId=null
uniqueId=null
#
```

To modify the application server to run in development mode and disable parallel start, modify the `developmentMode` and `parallelStartEnabled` properties, as the following example demonstrates:

```
#
# Configuration properties file for cells/myCell/nodes/myNode/servers/server1|server.xml#
# Extracted on Thu Sep 06 00:27:26 CDT 2007
#
#
# Section 1.0 ## cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
# SubSection 1.0 # Server Section
#
ResourceType=Server
ImplementingResourceType=Server
ResourceId=cells/myCell/nodes/myNode/servers/server1|server.xml#server1
#
#
#Properties
#
shortName=null
serverType=APPLICATION_SERVER
developmentMode=true #boolean
name=server1
parallelStartEnabled=false #boolean
clusterName=C
modelId=null
uniqueId=null
#
```

4. Validate the properties file.

   Note: As a best practice, use the validateConfigProperties command to validate the modified properties file before applying the changes, as the following Jython example demonstrates:

   ```
   AdminTask.validateConfigProperties('-propertiesFileName ConfigProperties_server1.props
    -reportFile report.txt')
   ```

   The command returns a value of `true` if the system successfully validates the properties file. The command returns a value of `false` if the system does not validate the file.

5. Apply the changes to the application server.

   Use the applyConfigProperties command to apply the changes to the application server.

   ```
   AdminTask.applyConfigProperties('-propertiesFileName ConfigProperties_server1.props
   -validate true')
   ```

6. Save your configuration changes.

   ```
   AdminConfig.save()
   ```

Creating, modifying, and deleting configuration objects using one properties file

You can specify to create, modify, and delete objects in one properties file. You run the applyConfigProperties command to apply the configuration changes.

Before you begin

Determine the changes that you want to make to configuration objects.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to create, modify, and delete configuration objects from your environment.

You can create, delete, and modify objects using one properties file. Specify in the header of the properties section `DELETE=true` to delete an entire object or `DELTEPROP=true` to delete an object property and then run the applyConfigProperties command to apply the properties file. With this approach, you do not need to run the deleteConfigProperties command to delete an object.

Procedure

1. Start the wsadmin scripting tool.
2. Extract a properties file for the subtype of interest from your configuration.

   Use the extractConfigProperties command to extract the properties file for the resource of interest. The following example extracts the properties for the JDBCProvider resource to the `derby.props` file:

```
AdminTask.extractConfigProperties('[-propertiesFileName derby.props -configData
 Server=server1 -filterMechanism SELECTED_SUBTYPES -selectedSubTypes [JDBCProvider]]')
```

   The command generates a template file similar to the following sample template:

```
#
# SubSection 1.0 # JDBCProvider attributes
#
ResourceType=JDBCProvider
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:ServerCluster=!{clusterName}:JDBCProvider=Derby JDBC
 Provider (XA)
#

#
#Properties
#
classpath={${DERBY_JDBC_DRIVER_PATH}/derby.jar}
name=Derby JDBC Provider (XA) #required
implementationClassName=org.apache.derby.jdbc.EmbeddedXADataSource #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Built-in Derby JDBC Provider (XA)
providerType=Derby JDBC Provider (XA) #readonly
xa=true #boolean,default(false)
```

3. Edit the extracted properties file to specify that it create, modify, or delete configuration objects.

   To create a new object or modify or delete an existing object, edit the extracted properties file. You can specify one or more create, modify, and delete operations in the same properties file.

   - To create a new object, specify unique properties for an object. Set the ResourceId attribute to a value that does not exist in your configuration.

     The following example creates a new DataSource object, DefaultDatasource_1, which does not exist in the configuration:

```
#
# Create a new object
#
ResourceType=DataSource
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=Derby JDBC Provider:DataSource=jndiName#DefaultDatasource_1
#

#
#Properties
```

```
#
name=Default Datasource1 #required
jndiName=DefaultDatasource_1
manageCachedHandles=false #boolean,default(false)
provider=Derby JDBC Provider #ObjectName(JDBCProvider),readonly
description=Datasource for the WebSphere Default Application
logMissingTransactionContext=true #boolean,default(true)
```

- To modify an existing object, change one or more object properties.

  The following example changes the `description` property of the DefaultDatasource_1 object by adding _1 to the end of the description:

```
#
# Modify a property
#
ResourceType=DataSource
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=Derby JDBC Provider:DataSource=jndiName#DefaultDatasource_1
#

#
#Properties
#
description=Datasource for the WebSphere Default Application_1
```

- To delete an existing object property, specify `DELETEPROP=true` in the header of the properties file.

  The following example deletes the `description` property:

```
#
# Delete a property
#
ResourceType=DataSource
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=Derby JDBC Provider:DataSource=jndiName#DefaultDatasource_1
DELETEPROP=true
#

#
#Properties
#
description=Datasource for the WebSphere Default Application_1
```

- To delete an existing object, specify `DELETE=true` in the header of the properties file.

  The following example deletes the `DefaultDatasource` object:

```
#
# Delete an existing object
#
ResourceType=DataSource
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=Derby JDBC Provider:DataSource=jndiName#DefaultDatasource
DELETE=true
#

#
#Properties
#
name=Default Datasource #required
jndiName=DefaultDatasource
```

4. Run the applyConfigProperties command to apply the properties file and change your configuration.

   The following example command applies the `derby.props` properties file:

```
AdminTask.applyConfigProperties('[-propertiesFileName derby.props]')
```

   The command automatically validates the properties file, then applies the changes to your configuration.

   Note: If you run the applyConfigProperties command before you add the `DELETE=true` attribute and value to the properties file, the command resets each property to the default value. The system completely removes properties that do not have default values.

Results

The administrative command runs and applies the properties file.

What to do next

Save the changes to your configuration.

Creating and deleting configuration objects using properties files and wsadmin scripting

Use this topic to use an extracted properties file to create or delete configuration objects that are not server, cluster, application, or authorization group object types.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to create and delete configuration objects from your environment.

Procedure

1. Start the wsadmin scripting tool.
2. Extract a properties file for the subtype of interest from your configuration.

   Use the extractConfigProperties command to extract the properties file for the resource of interest. The following example extracts the properties for the ThreadPool resource:

```
AdminTask.extractConfigProperties('[-propertiesFileName threadPool.props -configData
 Server=server1 -filterMechanism SELECTED_SUBTYPES -selectedSubTypes [ThreadPool]]')
```

   The command generates a template file similar to the following sample template:

```
# # SubSection 1.0.1.4 # Thread pools # ResourceType=ThreadPool
 ImplementingResourceType=Server ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:Thr
 eadPoolManager=ID#ThreadPoolManager_1:ThreadPool=ID#builtin_ThreadPool_4 #  # #Properties #
 maximumSize=20 #integer name=Default inactivityTimeout=5000 #integer minimumSize=5
#integer isGrowable=false #boolean
```

3. Create or delete configuration objects.

   To create a new thread pool or delete the existing thread pool, modify the ResourceId attribute.

   - To create a new thread pool, set the ResourceId attribute to a value that does not exist in your configuration. In the following example, note that the `ThreadPool=ID#builtin_ThreadPool_4` ResourceId is replaced with the `ThreadPool=ID#ThreadPool_99999` ResourceId, which does not exist in the configuration:

```
 # # SubSection 1.0.1.4 # Thread pools # ResourceType=ThreadPool
 ImplementingResourceType=Server ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:Thr
 eadPoolManager=ID#ThreadPoolManager_1:ThreadPool=ID#ThreadPool_99999 #  # #Properties # maximumSize=20
 #integer name=myThreadPool inactivityTimeout=5000 #integer minimumSize=5 #integer isGrowable=false #Boolean
```

     Run the applyConfigProperties command to apply the properties file to your configuration, as the following command demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName threadPool.props]')
```

     The command automatically validates the properties file, then uses the modified values in the file to create a new thread pool in your configuration.

   - To delete the thread pool, specify the `DELETE=true` property in the header of the properties file, as the following example demonstrates:

```
# # SubSection 1.0.1.4 # Thread pools # ResourceType=ThreadPool
 ImplementingResourceType=Server ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:Thr
 eadPoolManager=ID#ThreadPoolManager_1:ThreadPool=myThreadPool DELETE=true #  # #Properties # maximumSize=20
#integer  name=myThreadPool inactivityTimeout=5000 #integer minimumSize=5 #integer isGrowable=false #boolean
```

     Run the deleteConfigProperties command to use the properties file to remove the thread pool from your configuration, as the following command demonstrates:

```
AdminTask.deleteConfigProperties('[-propertiesFileName threadPool.props]')
```

     The command automatically validates the properties file, then uses the new attribute and value in the file to remove the thread pool from your configuration.

     Note: If you run the deleteConfigProperties command before you add the `DELETE=true` attribute and value to the properties file, the command resets each property to the default value. The system completely removes properties that do not have default values.

4. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Creating server, cluster, application, or authorization group objects using properties files and wsadmin scripting

Use this topic to create new server, cluster, application, or authorization group objects for your configuration.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to create configuration objects in your environment.

Procedure

1. Start the wsadmin scripting tool.
2. Create a properties file template.

   Create a properties file template to use to create the new server, cluster, application, or authorization group object. Use the -configType parameter and the following guidelines to specify the type of template to create:

   - Specify `Server` to create a server type properties file template.
   - Specify `ServerCluster` to create a server cluster type properties file template.
   - Specify `Application` to create an application type properties file template.
   - Specify `AuthorizationGroup` to create an authorization group type properties file template.

   The following Jython example uses the createPropertiesFileTemplates command to create a new AuthorizationGroup object template:

   ```
   AdminTask.createPropertiesFileTemplates('[-propertiesFileName authorizationGroup.template -configType AuthorizationGroup]')
   ```

   The command generates a template file similar to the following sample template:

   ```
   #
   # Create parameters
   # Replace the line `SKIP=true` with 'SKIP=false' under each section that is needed
   # Set necessary parameters under each command or step sections
   # Invoke applyConfigProperties command using this properties file.
   #
   ResourceType=AuthorizationGroup
   ImplementingResourceType=AuthorizationGroup
   ResourceId=AuthorizationGroup=
   SKIP=true
   CreateDeleteCommandProperties=true
   #

   #
   #Properties
   #
   authorizationGroupName=authorizationGroupName #String,required
   commandName=createAuthorizationGroup
   ```

3. Modify the new template file.

   Modify the new AuthorizationGroup template file by setting the required parameters. You can also modify the optional parameters, but you must modify the required parameters. Change the `SKIP` required property value from `SKIP=true` to `SKIP=false` to indicate that the system should apply the properties in the specific section of the properties file to the configuration. To ignore a specific section of a properties file, set the `SKIP` property to `SKIP=true`.

   ```
   #
   # Create parameters
   # Replace the line `SKIP=true` with 'SKIP=false' under each section that is needed
   # Set necessary parameters under each command or step sections
   # Invoke applyConfigProperties command using this properties file.
   #
   ```

```
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=
SKIP=false
CreateDeleteCommandProperties=true
#

#
#Properties
#
authorizationGroupName=ag1 #String,required
commandName=createAuthorizationGroup
```

4. Apply the modified properties to your configuration.

   Use the applyConfigProperties command to apply the properties file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName authorizationGroup.template]')
```

   The command creates the `ag1` authorization group in your configuration.

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

Deleting server, cluster, application, or authorization group objects using properties files

Use this topic to delete server, cluster, application, or authorizationgroup objects from your configuration.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can use properties files to delete configuration objects from your environment.

Procedure

1. Start the wsadmin scripting tool.
2. Create a properties file template.

   Create a properties file template to use to delete the server, cluster, application, or authorization group object of interest. Use the -configType parameter and the following guidelines to specify the type of template to create:

   - Specify `Server` to create a server type properties file template.
   - Specify `ServerCluster` to create a server cluster type properties file template.
   - Specify `Application` to create an application type properties file template.
   - Specify `AuthorizationGroup` to create an authorization group type properties file template.

   The following Jython example uses the createPropertiesFileTemplates command to create a new AuthorizationGroup object template:

```
AdminTask.createPropertiesFileTemplates('[-propertiesFileName authorizationGroup.template -configType AuthorizationGroup]')
```

   The command generates a template file similar to the following sample template:

```
#
# Delete parameters
# Replace the line 'SKIP=true' with 'SKIP=false' under each section that is needed
# Set necessary parameters under each command or step sections
# Invoke deleteConfigProperties command using this properties file.
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=
SKIP=true
CreateDeleteCommandProperties=true
#

#
```

```
#Properties
#
authorizationGroupName=authorizationGroupName #String,required
commandName=deleteAuthorizationGroup
```

3. Modify the new template file.

   Modify the new AuthorizationGroup template file by setting the required parameters. You can also modify the optional parameters, but you must modify the required parameters.

   Change the `SKIP` required property value from `SKIP=true` to `SKIP=false` to indicate that the system should apply the properties in the specific section of the properties file to the configuration. To ignore a specific section of a properties file, set the `SKIP` property to `SKIP=true`.

```
#
# Delete parameters
# Replace the line `SKIP=true` with 'SKIP=false' under each section that is needed
# Set necessary parameters under each command or step sections
# Invoke deleteConfigProperties command using this properties file.
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=authorizationGroupName
SKIP=false
CreateDeleteCommandProperties=true
#

#
#Properties
#
authorizationGroupName=authorizationGroupName #String,required
commandName=deleteAuthorizationGroup
```

4. Remove the object from your configuration.

   Use the deleteConfigProperties command to remove the existing AuthorizationGroup object from the configuration, as the following Jython example demonstrates:

```
AdminTask.deleteConfigProperties('[-propertiesFileName authorizationGroup.template]')
```

   The command removes the `ag1` authorization group in your configuration.

5. Save the configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```

Extracting properties files using wsadmin scripting

Use this topic to extract properties files from your configuration. You can use the wsadmin tool to extract properties files for cell, server, server subtype, and node configurations.

About this task

Using the PropertiesBasedConfiguration command group for the AdminTask object, you can extract the configuration attributes and values from your environment to properties files.

Complete the following steps to run the extractConfigProperties command and extract a properties file for a cell, server, server subtype, or node configuration. Optionally, you can use interactive mode with the command:

```
AdminTask.extractConfigProperties('-interactive')
```

Procedure

- Extract a cell configuration.
  1. Start the wsadmin scripting tool.
  2. Extract the cell configuration.

     Use the extractConfigProperties command to extract the object configuration, as the following Jython example demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName
 ConfigProperties_cell.props]')
```

The system extracts the properties file, as the following example displays:

```
Cell.props  # # SubSection 1.0 # Cell level attributes # ResourceType=Cell
 ImplementingResourceType=Cell ResourceId=Cell=!{cellName} #  # #Properties # shortName=null
cellType=DISTRIBUTED #ENUM(UDP|TCP|MULTICAST|DISTRIBUTED|STANDALONE),readonly name=!{cellName}
 multicastDiscoveryAddressEndpointName=null  discoveryAddressEndpointName=null cellDiscoveryProtocol=TCP
 #ENUM(UDP|TCP|MULTICAST)  .... ...  Properties of nodes,servers,  clusters, applications, etc. ....
    EnvironmentVariablesSection # # #Environment Variables #Day Month 17 Time CDT Year  cellName=myCell
```

The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section contains each variable in the properties file.

- Extract a server configuration.
  1. Start the wsadmin scripting tool.
  2. Extract the application server configuration of interest.

     Use the extractConfigProperties command to extract the server configuration, as the following Jython example demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName ConfigProperties_server1.props -configData Server=server1]')
```

The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # Server Section # ResourceType=Server ImplementingResourceType=Server
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName} #  # #Properties
# shortName=null serverType=DEPLOYMENT_MANAGER  #readonly developmentMode=false
#boolean parallelStartEnabled=true #boolean name=!{serverName} clusterName=null uniqueId=null
modelId=null  ... ... Properties of other inner objects ( EJBContainer, WebContainer, ORB etc)
 and subtypes not shown. ...   EnvironmentVariablesSection # #Environment Variables
#Day Month 16 Time CDT Year cellName=myCell nodeName=myNode  hostName=myHost.com serverName=dmgr
```

The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section contains each variable in the properties file.

- Extract the a server subtype configuration for a specific server.
  1. Start the wsadmin scripting tool.
  2. Extract the EJB container and web container properties for a specific server.

     Use the extractConfigProperties command to extract the server configuration, as the following Jython examples demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName ejbcontainer.props -configData
 Server=server1 -filterMechanism SELECTED_SUBTYPES -selectedSubTypes [EJBContainer WebContainer]]')
```

The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # EJBContainer # ResourceType=EJBContainer
 ImplementingResourceType=EJBContainer
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
ID#ApplicationServer_1:EJBContainer=ID#EJBConntainer_1 AttributeInfo=components
#  # #Properties # EJBTimer={} #ObjectName*(null) name=null defaultDatasourceJNDIName=null
  inactivePoolCleanupInterval=30000 #long passivationDirectory="${USER_INSTALL_ROOT}/temp"
 enableSFSBFailover=false #boolean  server=null parentComponent=
WebSphere Application Server, Network Deployment Server  # # SubSection 1.0
# WebContainer  # ResourceType=WebContainer  ImplementingResourceType=WebContainer
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
ID#ApplicationServer_1:WebContainer=ID#WebConntainer_1 AttributeInfo=components #  # #Properties
# enableServletCaching=false #boolean name=null defaultVirtualHostName=null  server=null
maximumPercentageExpiredEntries=15 #integer asyncIncludeTimeout=60000 #integer parentComponent=WebSphere Application Server, Network Deployment
 Server disablePooling=false #boolean sessionAffinityFailoverServer=null
maximumResponseStoreSize=100 #integer  allowAsyncRequestDispatching=false #boolean
sessionAffinityTimeout=0 #integer   EnvironmentVariablesSection # #Environment
 Variables #Thu Apr 17 14:17:25 CDT 2008 cellName=myCell nodeName=myNode
hostName=myhost.com serverName=dmgr
```

The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section contains each variable in the properties file.

The EJBContainer=ID#EJBContainer_1 string represents the EJBContainer object within the server. Use this XML ID to uniquely identify the object in the configuration. You can modify this field to EJBContainer=myContainer if the name field is set to myContainer in the configuration before you apply the properties file to the configuration.

- Extract node properties without traversing the subtypes of the node.
  1. Start the wsadmin scripting tool.
  2. Extract the node properties, except for specific subtype properties of servers and resources.

     Use the extractConfigProperties command to extract the node configuration properties, as the following Jython examples demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName node.props -configData
 Node=myNode -filterMechanism NO_SUBTYPES]')
```

     The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # Node Section # ResourceType=Node ImplementingResourceType=Node
 ResourceId=Cell=!{cellName}:Node=!{nodeName} #  # #Properties # shortName=null name=!{nodeName}
 maxFilePermissionForApps=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755 " discoveryProtocol=TCP
#ENUM(UDP|TCP|MULTICAST)  hostName=!{hostName}  # ## Section 1.0_1#Cell=!{cellName}:Node=!{nodeName}
 # ResourceType=Node ImplementingResourceType=Node  ExtensionId=NodeMetadataExtension
ResourceId=Cell=!{cellName}:Node=!{nodeName}  # nodeOS=distributed nodeVersion=7.0.0.0   # #
 End of Section 1.0_1# Cell=!{cellName}:Node=!{nodeName} #  # # End of Section 1.0# Cell=!{cellName}
:Node=!{nodeName} #   EnvironmentVariablesSection # #Environment Variables #Day Month 17 Time
CDT Year cellName=myCell nodeName=myNode
```

     The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section of the properties file contains each variable in the file.

- Extract node properties without traversing the subtypes of the node or invoking extensions.
  1. Start the wsadmin scripting tool.
  2. Extract the node properties, except for specific subtype properties of servers and resources and without invoking extensions.

     Use the extractConfigProperties command to extract the node configuration properties, as the following Jython examples demonstrates:

```
AdminTask.extractConfigProperties('[-propertiesFileName node.props -configData
 Node=myNode -filterMechanism NO_SUBTYPES_AND_EXTENSIONS]')
```

     The system extracts the properties file, as the following example displays:

```
# # SubSection 1.0 # Node Section # ResourceType=Node ImplementingResourceType=Node
 ResourceId=Cell=!{cellName}:Node=!{nodeName} #  # #Properties # shortName=null name=!{nodeName}
 maxFilePermissionForApps=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755 " discoveryProtocol=TCP
#ENUM(UDP|TCP|MULTICAST)  hostName=!{hostName}  # ## Section 1.0_1#Cell=!{cellName}:Node=!{nodeName}
 # ResourceType=Node ImplementingResourceType=Node  ExtensionId=NodeMetadataExtension
ResourceId=Cell=!{cellName}:Node=!{nodeName}  # nodeOS=distributed nodeVersion=7.0.0.0   # #
 End of Section 1.0_1# Cell=!{cellName}:Node=!{nodeName} #  # # End of Section 1.0# Cell=!{cellName}:Node=!{nodeName}
 #  EnvironmentVariablesSection # #Environment Variables #Day Month 17 Time CDT Year cellName=myCell nodeName=myNode
```

     The command excludes the NodeMetadataExtension section from the extracted properties file, as that is an extension to a node resource. The properties file does not display the cell, node, server, cluster, application, core group, or node group names. Instead, the command creates variables, such as !{cellName}, and includes them in the EnvironmentVariables section at the bottom of the properties file. The Environment Variables section of the properties file contains each variable in the file.

What to do next

After extracting properties files, use this functionality for various purposes, including:

- To modify your existing configuration in one location, instead of configuring multiple administrative console panels or running many commands
- To improve the application development life cycle

You can use properties files to manage the following server subtypes in your environment:

- Application servers

- Nodes
- Profiles
- Virtual hosts
- Applications
- Authorization tables
- Data replication domains
- Variable maps
- Java™ Database Connectivity (JDBC) providers
- Uniform Resource Locator (URL) providers
- Mail providers
- Resource environment providers
- Java 2 Connector (J2C) resource adapters

Extracting or modifying WCCM object properties

Use the wsadmin tool to extract or modify the properties of an existing WCCM (WebSphere Common Configuration Model) object.

Before you begin

Determine the WCCM object whose properties you want to extract.

About this task

In previous releases of the product, extracted properties only contained the most important properties of each object type. You now can extract all the properties of any WCCM object, modify the properties, and then validate and apply the modified properties to a system configuration.

To extract WCCM object properties, use the extractConfigProperties command and specify the WCCM object identifier.

The extracted WCCM object properties file is not portable among environments. If you intend to apply the properties files to installations on different operating systems, you must modify the extracted properties file to make it portable. See the topic on applying portable properties files across multiple environments.

Also, the extracted properties file does not contain nested objects. You must extract the properties of each nested object independently.

For each of the commands in this topic, you can run in interactive mode by specifying the interactive parameter:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Start the wsadmin scripting tool.

   To start wsadmin using the Jython language, run the following command from the `bin` directory of the server profile:

```
wsadmin -lang Jython
```

2. Extract WCCM object properties using the createPropertiesFileTemplates command.

   For example, to extract properties for the WCCM SIBus object `mySib(cells/myCell04/buses/mySib|sib-bus.xml#SIBus_1250621844296)` to a file named `mySIBus.props`, run the following command:

```
AdminTask.extractConfigProperties('mySib(cells/myCell04/buses/mySib|sib-bus.xml#SIBus_1250621844296)',
   '[-propertiesFileName mySIBus.props ]')
```

   The resulting `mySIBus.props` file contains extracted properties such as the following:

```
#
ResourceType=SIBus
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:SIBus=!{sibus}
#

#
#Properties
#
secure=false #boolean,default(true)
uuid=1CAE88EF49150090
useServerIdForMediations=false #boolean,default(false)
name=mySib
interEngineAuthAlias=null
...

EnvironmentVariablesSection
#
#
#Environment Variables
sibus=mySib
cellName=myCell04
```

3. Open an editor on the extracted properties file and modify the extracted properties file as needed.
   Ensure that the extracted properties file provides suitable values for required parameters.
4. Apply the properties file using the applyConfigProperties command.
   For example, to apply the `mySIBus.props` properties file, run following wsadmin command:

```
AdminTask.applyConfigProperties('[-propertiesFileName mySIBus.props]')
```

Results

The administrative command runs and applies the properties file.

What to do next

Save the changes to your configuration.

Validating properties files using wsadmin scripting

Use this topic to validate configuration properties before applying properties files to your configuration.

Before you begin

Use the extractConfigProperties command in the PropertiesBasedConfiguration command group to extract a properties file from your configuration. Use a text editor to modify the properties in the properties file.

About this task

There are two steps to validate a properties file before applying it to the configuration. First, use the validateConfigProperties command to validate the properties file. Then, use the applyConfigProperties command and the -validate option to apply the properties and validate the file simultaneously.

Procedure

Use the validateConfigProperties command to validate a properties file.

1. Start the wsadmin scripting tool.
2. Validate the properties file of interest.
   For this example, validate the following EJBContainer properties file:

```
# # SubSection 1.0 # EJBContainer # ResourceType=EJBContainer
 ImplementingResourceType=EJBContainer
 ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}
 :ApplicationServer=ID#ApplicationServer_1:EJBContainer=ID#EJBCon
ntainer_1 AttributeInfo=components # # #Properties # EJBTimer={}
 #ObjectName*(null) name=null defaultDatasourceJNDIName=null
 inactivePoolCleanupInterval=30000 #long passivationDirectory="${USER_INSTALL_ROOT}/temp"
  enableSFSBFailover=true#boolean server=null parentComponent=Network Deployment Server
```

Always validate the entire properties file. Do not validate subsections of files. Use the validateConfigProperties command to validate the properties file, as the following Jython example demonstrates:

```
AdminTask.validateConfigProperties('[-propertiesFileName ejbcontainer.props
 -variablesMapFileName ejbprops.vars -reportFileName report.txt]')
```

The command returns a value of `true` if the system successfully validates the properties file. The command returns a value of `false` if the system does not validate the file.

The command also generates a report file and records configuration actions such as:

- changes to property values.
- no change to property values when the configuration value is the same as defined in the properties file.
- no change to read-only property values.
- exceptions.

The following example displays a sample report file:

```
ADMG0820I: Start applying properties from file ejbcontainer.props ADMG0818I: Processing
 section EJBContainer:ApplicationServer. ADMG0810I: Not changing value for this property EJBTimer. New value specified is same as
 current value {}. ADMG0810I: Not changing value for this property defaultDatasourceJNDIName. New value specified is same as
 current value null. ADMG0811I: Changing value for this property enableSFSBFailover. New value specified is true. Old value was
 false. ADMG0810I: Not changing value for this property inactivePoolCleanupInterval. New value specified is same as current value
 30000. ADMG0810I: Not changing value for this property name. New value specified is same as current value null. ADMG0807I:
 Property parentComponent is readonly. Will not be modified ADMG0810I: Not changing value for this property passivationDirectory.
 New value specified is same as current value ${USER_INSTALL_ROOT}/temp. ADMG0807I: Property server is readonly. Will not be
 modified ADMG0819I: End Processing section EJBContainer:ApplicationServer.
```

To make the reports more concise, specify the reportFilterMechanism parameter with the validateConfigProperties command to only report errors and changes to the configuration, as the following example demonstrates:

```
AdminTask.validateConfigProperties('[-propertiesFileName ejbcontainer.props
 -variablesMapFileName ejbprops.vars -reportFileName report.txt -reportFilterMechanism
 Errors_And_Changes]')
```

The filtered report file displays error and configuration changes only, as the following sample output demonstrates:

```
ADMG0820I: Start applying properties from file ejbcontainer.props ADMG0811I: Changing value
 for this property enableSFSBFailover. New value specified is true. Old value was false.  AADMG0831E: Value specified for property
 inactivePoolCleanupInterval is not a valid type. Specified value asdf, Required type long. ADMG0821I: End applying properties
 from file ejbcontainer.props.
```

What to do next

If validation of a properties file fails, the generated report file states the reason for failure. The report file lists any changes that are made to the configuration and any errors.

To identify the problem, you can set the report filter mechanism parameter to report only errors. Otherwise the generated report file might be too big to look for actual errors. In a report file with a combination of changes and errors, search for `ADMGXXXXE` messages or for one of the following `ADMGXXXX` messages:

- `ADMG0809I`
- `ADMG0815I`
- `ADMG0826I`
- `ADMG0829I`
- `ADMG0831I`
- `ADMG0832I`
- `ADMG0833I`
- `ADMG0834I`
- `ADMG0835I`

These ADMG messages are errors. If the report has any of these ADMG messages, correct the error condition in the properties file. The error caused the specified property or property value to be not valid.

Applying properties files using wsadmin scripting

Use this topic and the wsadmin tool to apply modified configuration properties to your environment using properties files.

Before you begin

Use the extractConfigProperties command in the PropertiesBasedConfiguration command group to extract the properties files of interest. Use a text editor to modify one or more values in the properties file.

Use the validateConfigProperties command in the PropertiesBasedConfiguration command group to validate the modified properties file before applying the file to your configuration.

About this task

You can also use interactive mode with these commands, as the following syntax demonstrates:

```
AdminTask.commandName('-interactive')
```

Procedure

- Modify one or more properties and apply the properties file to the configuration.
  1. Start the wsadmin scripting tool.
  2. Modify the properties of interest.

     In the following properties file, use a text editor to change the value of the `enableSFSB` property:

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
ID#ApplicationServer_1:EJBContainer=ID#EJBContainer_1AttributeInfo=components
#

#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long
passivationDirectory="${USER_INSTALL_ROOT}/temp"
enableSFSBFailover=true #boolean
server=null
parentComponent=WebSphere Application Server, Network Deployment Server

EnvironmentVariablesSection
#
#
#Environment Variables
#Thu Apr 17 14:10:31 CDT 2008
hostName2=*
hostName1=localhost
cellName=IBM-49F7FB781FECell07
nodeName=IBM-49F7FB781FECellManager07
hostName=IBM-49F7FB781FE.austin.ibm.com
serverName=dmgr
enableSSB=true
```

  3. Apply the modified properties to your configuration.

     Use the applyConfigProperties command to apply the properties file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName ejbcontainer.props]')
```

- Use additional user modified variables to modify the configuration.
  1. Start the wsadmin scripting tool.
  2. Use additional variables to modify the enableSFSBFailover property of the EJB container, changing the value from `true` to `false`.

In the following properties file, modify the enableSFSBFailover property by specifying the value as the !{enableSSB} variable. You can use the variable in the section header or in the properties part of the section. Also, one property value can contain multiple variables as shown for ResourceId.

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:
EJBContainer=ID#EJBContainer_1
AttributeInfo=components
#

#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long
passivationDirectory="${USER_INSTALL_ROOT}/temp"
enableSFSBFailover=!{enableSSB} #boolean
server=null
parentComponent=WebSphere Application Server, Network Deployment Server

EnvironmentVariablesSection
#
#
#Environment Variables
#Thu Apr 17 14:10:31 CDT 2008
hostName2=*
hostName1=localhost
cellName=IBM-49F7FB781FECell07
nodeName=IBM-49F7FB781FECellManager07
hostName=IBM-49F7FB781FE.austin.ibm.com
serverName=dmgr
enableSSB=true
```

3. Apply the modified properties to your configuration.

   Use the applyConfigProperties command to apply the properties file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName ejbcontainer.props]')
```

- Modify the configuration by applying a properties file and a variable map.
  1. Start the wsadmin scripting tool.
  2. Modify the enableSFSBFailover property of the EJB container, changing the value from `true` to `false`.

     Modify the enableSFSBFailover property by specifying the value as the !{enableSSB} variable in a separate variable map file. Instead of specifying the variable in the section header or in the properties part of the section, create a separate variable map file. The following code displays a sample variable map file:

```
ejbprops.vars:
#
#
#Environment Variables
#Day Month 11 Time CDT Year
hostName2=*
hostName1=localhost
cellName=myCell
nodeName=myNode
hostName=myhost.com
serverName=myServer
enableSSB=true
```

     The following code displays the corresponding properties file to apply to the configuration:

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=ID#ApplicationServer_1:
EJBContainer=ID#EJBContainer_1
AttributeInfo=components
#
#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
```

```
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long
passivationDirectory="${USER_INSTALL_ROOT}/temp"
enableSFSBFailover=true#boolean
server=null
parentComponent=WebSphere Application Server, Network Deployment Server
```

3. Apply the modified properties to your configuration.

   Use the applyConfigProperties command to apply the properties file and the variable map file to the configuration, as the following Jython example demonstrates:

```
AdminTask.applyConfigProperties('[-propertiesFileName ejbcontainer.props -variablesMapFileName
 ejbprops.vars]')
```

What to do next

To verify that the system made the changes to your configuration, extract the properties file from your configuration using the extractPropertiesFile command.

Applying portable properties files across multiple environments

Use the wsadmin tool to extract a properties file from one cell, modify environment-specific variables at the bottom of the extracted properties file, and then apply the modified properties file to another cell. Modifying environment-specific variables makes a properties file portable.

Before you begin

If the properties file that you want to edit was created before Version 7.0.0.7 of the product, examine the properties file and see whether it contains an XMI id such as the following:

```
#
# SubSection 1.0 # Virtual Hosts
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=ID#VirtualHost_1
#
#Properties
#
name=default_host

EnvironmentVariablesSection
#Environment Variables
cellName=myNode04Cell
```

An *XMI id* is a unique identifier that a product previous to Version 7.0.0.7 generates when creating a configuration object. An XMI id can be different in another environment for the same object. In this example, a VirtualHost resource for default host in one environment has an XMI id of `VirtualHost_1`. In another environment, the XMI id might be a different value, such as `VirtualHost_2`. Properties files that have XMI identifiers are not portable. You cannot apply extracted properties that have XMI identifiers to another environment without first modifying the resource identifiers.

The same virtual host section in a properties file that has portable resource identifiers resembles the following:

```
#
# SubSection 1.0 # Virtual Hosts
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=default_host
#
#Properties
#
name=default_host

EnvironmentVariablesSection
#Environment Variables
cellName=myNode04Cell
```

In this example, name is used as a key attribute to identify the VirtualHost object uniquely within an environment. An object can have more than one key attribute to uniquely identify it among different instances of the same type.

About this task

You can apply properties files that have portable resource identifiers to another environment.

To extract a properties file so that it has portable resource identifiers, use the extractConfigProperties command with the PortablePropertiesFile option set to true. Properties files extracted with this option are portable. The extracted properties files do not identify each resource uniquely. A resource identifier might have one or more attribute name and value pairs; for example, a nodeName attribute identifies a node and a serverName attribute identifies a server.

By default, an extracted properties file is not portable. But a properties file extracted with the PortablePropertiesFile option set to true is portable.

After you extract a properties file with the PortablePropertiesFile option set to true, change the EnvironmentVariablesSection at the bottom of the properties file, copy the properties files to the target environment, and then run the applyConfigProperties command to apply the properties file to another cell.

You can also use interactive mode with these commands, as the following syntax demonstrates:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Start the wsadmin scripting tool.

   To start wsadmin using the Jython language, run the following command from the bin directory of the server profile:

```
wsadmin -lang Jython
```

2. Extract a properties file using the extractConfigProperties command with the PortablePropertiesFile option set to true.

   For example, to extract properties of a server named server1 to the server.props file in a portable format, run following wsadmin command:

```
AdminTask.extractConfigProperties('[-propertiesFileName server.props -configData Server=server1
-options [[PortablePropertiesFile true]] ]')
```

   If a properties file refers to a resource of type Server, Node, Application, Cluster, or AuthorizationGroup that does not exist in the target environment, consider setting the GenerateTemplates option to true:

```
AdminTask.extractConfigProperties('[-propertiesFileName server.props -configData Server=
-options [[GenerateTemplates true][PortablePropertiesFile true]] ]')
```

   When the GenerateTemplates option is used, the extracted properties file has properties sections that support creation of another object of the same type. By default this option is disabled.

3. Optional: Open an editor on the extracted properties file and examine the resource identifiers to ensure that they are portable.

   Portable identifiers do not identify each resource uniquely. The following examples show portable identifiers in various subsections of properties files.

   Example 1: Using an attribute name and a value for a resource identifier

   In this example, jndiName is a portable resource identifier that identifies a DataSource:

```
#
# SubSection 1.0.1.0 # DataSource attributes
#
Resou:rceType=DataSource
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=Derby JDBC
 Provider(XA):DataSource=jndiName#jdbc/DefaultEJBTimerDataSource
```

   Example 2: Using multiple attribute name and value pairs for a resource identifier

In this example, the nodeName and serverName attributes together identify the coreGroupServer resource.

```
#
# SubSection 1.0.8.0 # CoreGroupServer Section
#
ResourceType=CoreGroupServer
ImplementingResourceType=CoreGroup
ResourceId=Cell=!{cellName}:CoreGroup=!{coreGroup}:CoreGroupServer=nodeName#myNode04,serverName#server1
AttributeInfo=coreGroupServers
```

Example 3: Singleton resource identifier

In this example, there is only one Security object in the cell. Because the Security object is considered a singleton object, no attribute is required to identify this resource uniquely.

```
#
# SubSection 1.0 # Security Section
#
ResourceType=Security
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=
```

Note: Some resources, such as JDBCProvider and ThreadPool, can be created with the same name multiple times. The name value is the key attribute for these objects. Do not create multiple instances of these objects with the same name under a given scope.

4. Modify the extracted properties file as needed.
   - If the extracted portable properties file refers to a resource of type Server, Node, Application, Cluster, or AuthorizationGroup that does not exist in the other environment and the extractConfigProperties command was run with the GenerateTemplates option set to true, edit the properties file to enable creation of the resource.

     The GenerateTemplates option enables you to create another server that is similar to an existing server. For example, when server properties are extracted using this option, the extracted properties file has a template at the top of the file to create another server. The template section is skipped by default. If you set SKIP=false and then set the required properties to create a new object, the product creates a new object when the applyConfigProperties command is run and supplies the edited properties file. Because the following sections contain configuration properties of an existing server and those sections are processed when the applyConfigProperties command is run, the newly created server has the same configuration as the old server from which the properties file was extracted.

     You must modify the environment section to reflect the new server that is created by changing the nodeName, cellName and serverName variables.

     This option generates a template properties section in front of each of server, cluster, application, and authorization group section. The sections are disabled by default.

     For each object that does not exist in target environment, edit the section corresponding to that object. Insert proper properties values and remove SKIP=true. Set environment specific variables at the end of the properties file that reflect the new target environment.
   - Do not modify or delete properties that are used as keys to identify an object.

     If you modify or delete a key attribute in an object, then subsequent sections of the properties file must not reference the object again. The resource specified in the subsequent sections will not be found.

     For example, examine the virtual host properties in the following sections:

```
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=admin_host
#
#Properties
#
name=admin_host #required
#
# SubSection 1.0.1 # MimeTypes section
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=admin_host
AttributeInfo=mimeTypes(type,extensions)
```

If `name=admin_host` is changed to `name=my_host`, the Mime Types section that has `ResourceId==!{cellName}:VirtualHost=admin_host` will not be able to find the resource specified by ResourceId because the name is changed in the previous section. Similarly, if `name=admin_host` is deleted, the Mime Types section will not be able to find the resource specified by ResourceId.

5. Validate the properties file.

   Run the validateConfigProperties command. See the topic on validating properties files.

6. Copy the extracted portable properties file to another environment.

7. If the extracted portable properties file refers to a resource of type Server, Node, Application, Cluster, or AuthorizationGroup that does not exist in the other environment and the extractConfigProperties command was not run with the GenerateTemplates option set to `true`, create the resource in the other environment.

   Because the properties file originally applied to a different environment, a resource identifier in the properties file might refer to a resource that does not exist in the target environment. If a resource of type Server, Node, Application, Cluster, or AuthorizationGroup does not exist in the target environment and the properties files does not enable creation of the resource, create the resource in the target environment before applying the properties for that resource. Also, an attribute might reference another resource. Ensure that all the referenced resources exist and, if necessary, create resources of type Server, Node, Application, Cluster, or AuthorizationGroup that are missing.

8. Apply the extracted portable properties file in the other environment using the applyConfigProperties command.

   For example, to apply the properties file `server.props` and generate a report named `rep.txt`, run following wsadmin command:

```
AdminTask.applyConfigProperties('[-propertiesFileName server.props -reportFileName rep.txt]')
```

   If a resource that is specified in the properties file exists in the target environment and the property value that is specified in the properties file is the same as what is already set in the target environment, that property is skipped or not set in target environment. The product only applies the properties that are different from what is specified in the properties file.

   If a resource that is specified in the properties file does not exist in the target environment, the product creates a new resource and sets all the properties for the newly created resource from the values for the resource in the properties file. The product does not create resources of type Server, Node, Application, Cluster, or AuthorizationGroup unless the extractConfigProperties command was run with the GenerateTemplates option set to `true` and the properties file specifies one or more new resources of those types.

Results

The target environment is updated by the applied properties file.

What to do next

Save the changes to your configuration.

Running administrative commands using properties files

Use the wsadmin tool to run an existing administrative command using properties file based configuration. The command must not contain a parameter that uses a complex data type. Supported parameter types are basic types such as String, Long, Integer, Float, Double, Boolean, Character, Short, Byte, URL, and ObjectName, and complex types such as Array of basic types, Properties, DownLoadFile, and UpLoadFile.

Before you begin

Determine the administrative command that you want to run. Ensure that all parameters in the command use only a supported data type.

About this task

You can extract the properties that are required to run a command using the createPropertiesFileTemplates command. Specify `GenericType` for the configType parameter and a `commandName` option.

After extracting a properties file for a command, edit the properties file as needed, and then validate and apply the properties file.

For each of the commands in this topic, you can run in interactive mode by specifying the interactive parameter:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Start the wsadmin scripting tool.

   To start wsadmin using the Jython language, run the following command from the `bin` directory of the server profile:

```
wsadmin -lang Jython
```

2. Extract the properties that are required to run the administrative command.

   To extract the properties that are required to run a command, use the createPropertiesFileTemplates command. Specify `GenericType` for the configType parameter and `commandName` *command_name* for the options parameter.

   For example, to extract properties for the `createSIBus` command to a file named `createSIBus.props`, run the following command:

```
AdminTask.createPropertiesFileTemplates('[-propertiesFileName createSIBus.props
  -configType GenericType -options [[commandName createSIBus]] ]')
```

   The resulting `createSIBus.props` file contains the following extracted properties:

```
#
CreateDeleteCommandProperties=true
#SKIP=true
commandName=createSIBus
#

#
#Properties
#
busSecurity=false #Boolean
highMessageThreshold=null #Long
bus=myBus #String,required
...
```

3. Open an editor on the extracted properties file and modify the extracted properties file as needed.

   Ensure that the extracted properties file provides suitable values for required parameters.

4. Apply the properties file using the applyConfigProperties command.

   For example, to apply the `createSIBus.props` properties file, run following wsadmin command:

```
AdminTask.applyConfigProperties('[-propertiesFileName createSIBus.props]')
```

Results

The administrative command runs and applies the properties file.

What to do next

Save the changes to your configuration.

Properties file syntax

To use the properties file based configuration tool, properties files must use supported syntax. This topic describes some important syntax.

- The default value of a property is shown in the format *propertyName=propertyValue* #default (*defaultValue*); for example:

```
enable=true #default(false)
```

- Properties that are required to create an object are marked as required in the format *propertyName=propertyValue* #required; for example:

```
jndiName=myJndi #required
```

- Properties that you cannot change are marked as read-only in the format *propertyName=propertyValue* #readonly; for example:

```
providerType=stdProviderType #readonly
```

- Valid values for a property are shown in the format *propertyName=propertyValue #type*(*range*). The range is a list of values using a vertical bar (|) delimiter; for example:

```
state=START #ENUM(START|STOP)
```

- A single property can specify more than one of the tags readonly, type, required, and default after the # character. Separate the tags by a comma (,); for example:

```
enable=true #boolean,required,default(false)
```

- For properties of basic types such as string, int (integer) or short, use the format name=value *#type*; for example:

```
port=9090 #int
```

  If the type is not specified, the product uses the string type.
- For a list or array type of properties, use the format name={val1, val2, val3} *#type*, where *type* is the type of each object in the list or array.
- Represent ConfigId or ObjectName in scope format. Scope format is Cell=cellName:Node=nodeName....

  You can also use ConfigId format, but scope format is more portable because it does not include an xmi id value.
- A property value of null is ignored during applyConfigProperties command processing.

PropertiesBasedConfiguration command group for the AdminTask object using wsadmin scripting

You can use the Jython scripting language to manage your system configuration using properties files. Use the commands in the PropertiesBasedConfiguration group to copy configuration properties from one environment to another, troubleshoot configuration issues, and to apply one set of configuration properties across multiple profiles, nodes, cells, servers, or applications.

Use the following commands to manage your system configuration:

applyConfigProperties

The **applyConfigProperties** command applies properties in a specific properties file to the configuration. The system adds attributes or configuration data to the configuration if a specific properties do not exist. If the properties exist in the configuration, the system sets the new values for the attributes.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file to apply. (String, required)

Optional parameters

-variablesMapFileName
Specifies the name of the variables map file. This file contains values for variables that the system uses from the properties file. (String, optional)

-variablesMap
Specifies the values of the variables to use with the properties file. (Properties, optional)

-reportFileName
Specifies the name of a report file that contains the output for the applyConfigProperties command. (String, optional)

-reportFilterMechanism
Specifies the type of report filter mechanism. Specify `All` to display all report information. Specify `Errors` to display error information. Specify `Errors_And_Changes` to display error and change information. (String, optional)

-validate
Specifies whether to validate the properties file before applying the changes. By default, the command validates the properties file. Specify `false` to disable validation. (Boolean, optional)

-zipFileName
Specifies the name of the .zip file that contains the policy sets that you want applied to the cell. (String, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.applyConfigProperties('-propertiesFileName myPropFile.props -zipFileName myZipFile.zip
-validate true')
```

- Using Jython list:

```
AdminTask.applyConfigProperties(['-propertiesFileName', 'myPropFile.props', '-zipFileName',
'myZipFile.zip', '-validate', 'true'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.applyConfigProperties('-interactive')
```

createPropertiesFileTemplates

The **createPropertiesFileTemplates** command creates template properties files to use to create or delete specific object types. The command stores the template properties file in the properties file specified by the propertiesFileName parameter.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file where the template is stored. (String, required)

-configType
Specifies the resource type for the template to create. (String, required)

- Specify `Server` to create a server type properties file template.
- Specify `ServerCluster` to create a server cluster type properties file template.
- Specify `Application` to create an application type properties file template.
- Specify `AuthorizationGroup` to create an authorization group type properties file template.

Optional parameters

None

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.createPropertiesFileTemplates('-propertiesFileName serverTemplate.props -configType Server')
```

- Using Jython list:

```
AdminTask.createPropertiesFileTemplates(['-propertiesFileName', 'serverTemplate.props',
'-configType', 'Server'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.createPropertiesFileTemplates('-interactive')
```

deleteConfigProperties

The **deleteConfigProperties** command deletes properties in your configuration as designated in a properties file. The system removes the attributes or configuration data that corresponds to each property in the properties file.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file to delete. (String, required)

Optional parameters

-variablesMapFileName
Specifies the name of the variables map file. This file contains values for variables that the system uses from the properties file. (String, optional)

-variablesMap
Specifies the values of the variables to use with the properties file. (Properties, optional)

-reportFileName
Specifies the name of a report file that contains the output for the the command. (String, optional)

-reportFilterMechanism
Specifies the type of report filter mechanism. Specify `All` to display all report information. Specify `Errors` to display error information. Specify `Errors_And_Changes` to display error and change information. (String, optional)

-validate
Specifies whether to validate the properties file before applying the changes. By default, the command validates the properties file. Specify `false` to disable validation. (Boolean, optional)

Return value

The command does not return output.

Batch mode example usage

- Using Jython string:

```
AdminTask.deleteConfigProperties('-propertiesFileName myPropFile.props')
```

- Using Jython list:

```
AdminTask.deleteConfigProperties(['-propertiesFileName', 'myPropFile.props'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.deleteConfigProperties('-interactive')
```

extractConfigProperties

The **extractConfigProperties** command extracts configuration data in the form of a properties file. The system exports the most commonly used configuration data and attributes, converts the attributes to properties, and saves the data to a file. You can specify the resource of interest with the target object or the configData parameter. Use the configData parameter to specify a server, node, cluster, policy set, or application instance. If no configuration object is specified, the command extracts the profile configuration data.

Target object

Specify the object name of the configuration object of interest in the format:
`Node=`*nodeName*`:Server=`*serverName*

Required parameters

-propertiesFileName
Specifies the name of the properties file to extract. (String, required)

Optional parameters

-configData
Specifies the configuration object instance in the format `Node=node1`. (String, optional)

-options
Specifies additional configuration options, such as `GENERATETEMPLATE=true`. (Properties, optional)

-filterMechanism
Specifies filter information for extracting configuration properties. (String, optional)

- Specify `All` to extract all configuration properties.
- Specify `NO_SUBTYPES` to extract the properties of the given object without including the subtypes.
- Specify `SELECTED_SUBTYPES_AND_EXTENSIONS` to extract only properties of the given object type without including the subtypes. This option also prevents the command from extracting properties using extensions, if extensions exist for the object type.
- Specify `SELECTED_SUBTYPES` to extract specific configuration object subtypes specified with the selectedSubTypes parameter. This can include any subtype for a configuration object or any WCCM type that exists under the object type hierarchy.

-selectedSubTypes

Specifies the configuration properties to include or exclude when the command extracts the properties. Specify this parameter if you set the filterMechanism parameter to NO_SUBTYPES or SELECTED_SUBTYPES. The following strings are examples of subtypes: ApplicationServer, EJBContainer. (String, optional)

Table 451. extractConfigProperties -selectedSubTypes subtypes and extensions. You can configure properties of the object types and subtypes.

Configuration object type	Subtypes	Extensions
AdminService	None	None
Application	JDBCProvider,VariableMap	None
ApplicationServer	TransactionService, DynamicCache, WebContainer, EJBContainer, PortletContainer, SIPContainer, WebserverPluginSettings	None
AuthorizationGroup	None	None
AuthorizationTableExt	None	None
Cell	VirtualHost, DataReplicationDomain, ServerCluster, CoreGroup, NodeGroup, AuthorizationGroup, AuthorizationTableExt, Security, J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, EventInfrastructureProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, Node, VariableMap	None
CoreGroup	None	None
CoreGroupBridgeService	None	None
DynamicCache	None	None
EJBContainer	None	None
EventInfrastructureProvider	None	None
EventInfrastructureService	None	None
HAManagerService	None	None
J2CResourceAdapter	None	None
JDBCProvider	None	None
JMSProvider	None	None
JavaVirtualMachine	None	None
Library	None	None
MailProvider	None	None
NameServer	None	None
Node	Server, J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, EventInfrastructureProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, VariableMap	The NodeMetadata Extension extracts node Metadata properties.
NodeGroup	None	None
ObjectPoolProvider	None	None
ObjectRequestBroker	None	None
PMEServerExtension	None	None
PMIModule	None	None
PMIService	None	None
PortletContainer	None	None
SIPContainer	None	None
SchedulerProvider	None	None
Security	None	None

Table 451. extractConfigProperties -selectedSubTypes subtypes and extensions (continued). You can configure properties of the object types and subtypes.

Configuration object type	Subtypes	Extensions
Server	PMIService, AdminService, CoreGroupBridgeService, TPVService, ObjectRequestBroker, ApplicationServer, NameServer, J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, EventInfrastructureProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, VariableMap, EventInfrastructureService, PMEServerExtension, Library, HAManagerService, PMIModule, Security	The extension lists deployed applications for a specific server.
ServerCluster	J2CResourceAdapter, JDBCProvider, JMSProvider, MailProvider, URLProvider, ObjectPoolProvider, WorkManagerProvider, TimerManagerProvider, SchedulerProvider, VariableMap	The extension lists deployed applications for a specific cluster.
TPVService	None	None
TimerManagerProvider	None	None
TransactionService	None	None
URLProvider	None	None
VariableMap	None	None
VirtualHost	None	None
WebContainer	None	None
WebserverPluginSettings	None	None
WorkManagerProvider	None	None

-zipFileName
Specifies the name of the .zip file into which you want to extract policy sets. (String, optional)

Return value

The command returns the name of the properties file that the system creates.

Batch mode example usage

- Using Jython string:

```
AdminTask.extractConfigProperties('-configData Node=myNode -propertiesFileName myNodeProperties.props
-zipFileName myZipFile.zip')
```

- Using Jython list:

```
AdminTask.extractConfigProperties(['-configData', 'Node=myNode', '-propertiesFileName',
'myNodeProperties.props', '-zipFileName', 'myZipFile.zip'])
```

Interactive mode example usage

- Using Jython string:

```
AdminTask.extractConfigProperties('-interactive')
```

validateConfigProperties

The **validateConfigProperties** command verifies that the properties in the properties file are valid and can be successfully applied to the new configuration.

Target object

None.

Required parameters

-propertiesFileName
Specifies the name of the properties file to validate. (String, required)

Optional parameters

-variablesMapFileName
Specifies the name of the variables map file. This file contains values for variables that the system uses from the properties file. (String, optional)

-variablesMap
Specifies the values of the variables to use with the properties file. (Properties, optional)

-reportFileName
Specifies the name of a report file that contains the output for the applyConfigProperties command. (String, optional)

-reportFilterMechanism
Specifies the type of report filter mechanism. Specify `All` to display all report information. Specify `Errors` to display error information. Specify `Errors_And_Changes` to display error and change information. (String, optional)

-zipFileName
Specifies the name of the .zip file that contains the policy sets that you want applied to the cell. (String, optional)

Return value

The command returns a value of `true` if the system validates the properties file or policy set .zip file.

Batch mode example usage

- Using Jython string:

```
AdminTask.validateConfigProperties('-propertiesFileName myNodeProperties.props -zipFileName
 myZipFile.zip -reportFileName report.txt')
```

- Using Jython list:

```
AdminTask.validateConfigProperties(['-propertiesFileName', 'myNodeProperties.props', '-zipFileName',
'myZipFile.zip', '-reportFileName', 'report.txt'])
```

Interactive mode example usage

- Using Jython:

```
AdminTask.validateConfigProperties('-interactive')
```

Managing specific configuration objects using properties files

Use the wsadmin tool to change configuration properties and manage configuration objects of your environment using properties files.

Before you begin

Determine the changes that you want to make to your server configuration or its configuration objects.

Read Properties file syntax so that changes you make to properties files use the supported syntax.

About this task

Using a properties file, you can create, modify, or delete a configuration object.

Extracted properties files identify all required properties and default values for an object of a given type. Required properties have the comment `#required` following a property setting. Properties that have default

values have the comment *#object_type*,default(*default_value*) following a property setting. When a property that has a default value is deleted, the value of that property is set automatically to the default value. If a property does not have default value, it is not shown. Because extracted properties files contain information about all required and default properties, you can verify that property values are set properly before applying the properties file to create, modify, or delete an object.

For example, an extracted properties file of type JDBCProvider identifies all required and default properties:

```
#
# SubSection 1.0 # JDBCProvider attributes
#
ResourceType=JDBCProvider
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:ServerCluster=!{clusterName}:JDBCProvider=Derby JDBC Provider (XA)
#

#
# Properties
#
classpath={${DERBY_JDBC_DRIVER_PATH}/derby.jar}
name=Derby JDBC Provider (XA) #required
implementationClassName=org.apache.derby.jdbc.EmbeddedXADataSource #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Built-in Derby JDBC Provider (XA)
providerType=Derby JDBC Provider (XA) #readonly
xa=true #boolean,default(false)
```

Procedure

1. Start the wsadmin scripting tool.

   To start wsadmin using the Jython language, run the following command from the `bin` directory of the server profile:

```
wsadmin -lang Jython
```

2. Create, edit, and apply the properties file of the configuration object.

   Instructions for creating and updating properties files of various configuration objects follow:

   - Activity session service
   - Application
     - Application deployment
       - Web module deployment
       - EJB module deployment
       - Application configuration
         - Session manager
         - Web module configuration
           - Session manager
         - EJB module configuration
           - Session manager
     - Servlet cache
     - Eviction policy
     - Data replication service (DRS)
   - Cache provider
     - Object cache
       - Object cache J2EE resource
     - Servlet cache
     - Eviction policy
     - Data replication service (DRS)
   - Data replication domain
     - Data replication
   - J2C resource adapter
   - Java 2 Platform, Enterprise Edition (J2EE) resource property
   - J2EE resource property set

- Java virtual machine (JVM)
- JDBC provider
  - Data source
    - Connection pool
    - Data source J2EE resource property set
  - CMP connection factory
    - Mapping module
- JMS provider
  - MQ topic
  - MQ topic connection factory
  - MQ queue
  - MQ queue connection factory
  - Mapping module
- Mail provider
  - Mail session
- Object pool
  - Object pool provider
    - Object pool provider J2EE resource
  - Object pool manager
    - Object pool manager J2EE resource
- Scheduler provider
  - Scheduler configuration
    - Scheduler configuration J2EE resource
- Security
  - Lightweight Directory Access Protocol (LDAP)
  - Lightweight Third Party Authentication (LTPA)
  - Java Authentication and Authorization Service (JAAS) configuration entry
  - JAAS authorization data
  - Secure Sockets Layer (SSL) configuration
    - Secure socket layer
  - Retrieve SSL signer certificate
  - Enable global security and configure a federated user registry
  - Map users and resources using authorization group properties files
- Server
  - Application server
    - Class loader
    - Library reference
  - Custom service
  - Dynamic cache
  - End point
  - Enterprise JavaBeans (EJB) container
  - HTTP transport
  - Listener port
  - Object Request Broker
  - Performance Monitoring Infrastructure (PMI) service
  - Process definition
  - SOAP connector
  - Stream redirect
  - Thread pool
  - Trace service
  - Transaction service
  - Web container
    - Session manager
- SIBus

  - SIBus member
  - SIB destination
  - SIB engine
- Timer manager provider
  - Timer manager information
    - Timer manager information J2EE resource
- Transport channel service
  - HTTP inbound channel
  - SSL inbound channel
  - TCP inbound channel
  - Web container inbound channel
- URL provider
  - URL
- Variable map
- Virtual host
  - Host alias
  - Mime entry
- Web server
  - Plug-in
  - Plug-in server cluster
  - Key store file
  - Administrative server authentication
  - Web server process definition
  - Web server Java virtual machine (JVM)
  - Web server JVM system
- Work area service
  - Work area partition service
- Work manager provider
  - Work manager information
    - Work manager information J2EE resource
- Web services endpoint URL fragment

The instructions contain examples to create, delete and modify WebSphere configuration objects using the properties file based configuration tool. Instructions are provided for many configuration objects but not all the supported configuration objects.

You can use interactive mode with a command to extract, edit, or apply a properties file. Run the command with the interactive option:

```
AdminTask.command_name('-interactive')
```

What to do next

Save the changes to your configuration.

Working with activity session service properties files

You can use properties files to change activity session service configuration objects and custom properties.

Before you begin

Determine the changes that you want to make to your activity session service configuration objects or custom properties.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete an activity session service object. You can also create, modify, or delete activity session service custom properties.

Run administrative commands using wsadmin to change a properties file for an activity session service, validate the properties, and apply them to your configuration.

Table 452. Actions for activity session service configuration objects. You can modify and delete activity session service objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Run the deleteConfigProperties command. Deleting a property sets the default value, if there is a default value for the property.
create Property	Not applicable
delete Property	Not applicable

Table 453. Actions for activity session service custom properties. You can create, modify, and delete activity session service custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Edit an activity session service properties file.
  1. Set ActivitySessionService object properties as needed.

     Open an editor on an ActivitySessionService properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ActivitySessionService properties file follows:

```
#
# Header
#
ResourceType=ActivitySessionService
ImplementingResourceType=PMEServerExtension
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:PMEServerExtension=:ActivitySessionService=
AttributeInfo=activitySessionService
#

#
#Properties
#
enable=false #boolean,default(false)
context=null
defaultTimeout=300 #integer,required,default(300)

#
EnvironmentVariablesSection
```

```
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

2. Set ActivitySessionService custom properties as needed.

   To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=ActivitySessionService
ImplementingResourceType=PMEServerExtension
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:PMEServerExtension=:ActivitySessionService=
AttributeInfo=properties(name,value)
#

#
#Properties
#
existingProp=value
newProp=newValue

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

3. Run the applyConfigProperties command to change an activity session service configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the activity session service or an existing custom property, you can delete the entire activity session service object or the custom property.
  - To delete the entire object, run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to manage the activity session service object and its properties.

What to do next

Save the changes to your configuration.

Using application properties files to install, update, and delete enterprise application files

You can use application properties files to install enterprise application files on a server, update deployed applications or modules, or uninstall deployed applications or modules. An enterprise application file must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Before you can install enterprise application files on an application server, you must assemble modules as needed. Also, configure the target application server. As part of configuring the server, determine whether your application files can be installed to your deployment targets.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the deployment target profile.

About this task

Using an application properties file, you can install, update, or uninstall an enterprise application or module.

Run administrative commands using wsadmin to deploy an application.

Table 454. Actions for application properties files. You can specify properties that install, update, or uninstall applications.

Action	Procedure
create (install)	1. Specify properties that identify the application and deployment target. 2. Run the applyConfigProperties command to install the application.
modify (update)	1. Edit application properties in the properties file. To update an application file, specify in the Properties section: • `Update=true` • `operationType=add`, `operationType=update` or `operationType=delete` • `contentType=file`, `contentType=moduleFile`, `contentType=partialapp`, or `contentType=app` 2. Run the applyConfigProperties command to update the deployed application.
delete (uninstall)	1. Remove properties that identify the deployment target from the properties file. 2. Run the deleteConfigProperties command to uninstall the deployed application.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:
`AdminTask.`*`command_name`*`('-interactive')`

Instead of running the wsadmin commands manually to apply an application properties file, you can add the properties file to a monitored directory. The product automatically runs the wsadmin commands. See Installing enterprise application files by adding properties files to a monitored directory.

If you are installing a stand-alone web application archive (WAR) or a Session Initiation Protocol (SIP) archive (SAR), specify the context root of the WAR or SAR file. The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is `/gettingstarted` and the servlet mapping is `MySession`, then the URL is `http://host:port/gettingstarted/MySession`.

This topic describes how to complete the following procedures:

- Install an enterprise application on a deployment target.
- Extract the properties for a deployed enterprise application.
- Update a single file in a deployed enterprise application.
- Remove a single file from a deployed enterprise application.
- Update a single module in a deployed enterprise application.
- Remove a single module from a deployed enterprise application.
- Replace, add, or delete multiple files of a deployed enterprise application.
- Replace the entire deployed enterprise application.
- Uninstall an application from a deployment target.
- Edit the deployment options of a deployed application.
- Deploy an application again.

Procedure

- Install an enterprise application on a deployment target.
  1. Create a properties file that identifies the application and deployment target.

     Open an editor and create a properties file such as the following to install an Application configuration object:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=

# Properties
Name=hello
TargetServer=!{serverName}
TargetNode=!{nodeName}
EarFileLocation=/temp/Hello.ear
#TargetCluster=cluster1
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
nodeName=myNode
serverName=myServer
```

  2. Run the applyConfigProperties command to install the application.

     Running the applyConfigProperties command applies the properties file. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Extract the properties for a deployed enterprise application.

  Run the extractConfigProperties command to extract the configuration attributes and values of a deployed enterprise application to a properties file:

```
AdminTask.extractConfigProperties('[-propertiesFileName myApp.props -configData Application=MyApplication]')
```

  Running this Jython example produces a file named `myApp.props` that lists the properties of an Application configuration object named `MyApplication`. You can use the extracted properties file to view and edit the properties of the application. The MapModulesToServers section of the properties file resembles the following:

```
#
# SubSection 1.0.2
# MapModulesToServers Section. taskName and row0 should not be edited. row0 contains column names for the task.
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
#Properties
#
taskName=MapModulesToServers
row2={"My Web Application" MyWebApplication.war,WEB-INF/web.xml WebSphere:cell=!{cellName},
node=!{nodeName},server=!{serverName}
23 moduletype.web "Web Module"}
row1={"My Enterprise Java Bean" My.jar,META-INF/ejb-jar.xml WebSphere:cell=!{cellName},node=!{nodeName},
server=!{serverName} 20 moduletype.ejb "EJB Module"}
mutables={false false true false false false} #readonly
row0={module uri server ModuleVersion moduletype moduletypeDisplay} #readonly
```

  By default, the extractConfigProperties command produces output that displays all columns, including hidden and non-hidden columns, of install task and task data values in separate rows. The `mutables` row shows which columns you can edit (`true`) and which you cannot edit (`false`).

  Note: To enhance the output of application properties, run the AdminTask extractConfigProperties command with the `SimpleOutputFormat` option. When the option is set to `true`, the output displays non-hidden columns of application properties in *`columnName=value`* pairs. Hidden

columns of application properties are not included in the output. The enhanced output makes it easier for you to find and edit application property values. You can use an edited properties file to install or update an application. The following example specifies the `SimpleOutputFormat` option in the extractConfigProperties command:

```
AdminTask.extractConfigProperties('[-propertiesFileName myApp.props -configData Application=MyApplication
   -option [[SimpleOutputFormat true]]]')
```

With the `SimpleOutputFormat` option, the MapModulesToServers section of the extracted application properties file resembles the following:

```
#
# SubSection 1.0.2
# MapModulesToServers Section. taskName and lines marked as "#readonly" should not be edited.
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
#Properties
#
taskName=MapModulesToServers
row0={ module="My Enterprise Java Bean"  #readonly
             uri=My.jar,META-INF/ejb-jar.xml  #readonly
             server=WebSphere:cell=!{cellName},node=!{nodeName},server=!{serverName} }
row1={ module="My Web Application"  #readonly
             uri=MyWebApplication.war,WEB-INF/web.xml  #readonly
             server=WebSphere:cell=!{cellName},node=!{nodeName},server=!{serverName} }
```

- Update a single file in a deployed enterprise application.
  1. Edit the application properties file so that it specifies the file to add or change.

     Edit the properties of an Application configuration object. Specify `Update=true`, an operation type such as `operationType=add`, and `contentType=file` in the Properties section. The following example adds the `addMe.jsp` file to a deployed application named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
Update=true
operationType=add
contentType=file
contentURI=test.war/com/ibm/addMe.jsp
contentFile=c:/temp/addMe.jsp
```

  2. Run the applyConfigProperties command to update the application.

     Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Remove a single file from a deployed enterprise application.
  1. Edit the application properties file so that it specifies the file to remove.

     Edit the properties of an Application configuration object. Specify `Update=true`, `operationType=delete`, and `contentType=file` in the Properties section. The following example removes the `addMe.jsp` file from a deployed application named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
```

```
Update=true
operationType=delete
contentType=file
contentURI=test.war/com/ibm/addMe.jsp
```

2. Run the applyConfigProperties command to update the application.

   Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Update a single module in a deployed enterprise application.
  1. Edit the application properties file so that it specifies the Java EE module to add or change.

     Edit the properties of an Application configuration object. Specify `Update=true`, an operation type such as `operationType=add`, and `contentType=moduleFile` in the Properties section. The following example adds the `Increment.jar` file to a deployed application named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
Update=true
operationType=add
contentType=modulefile
#contextRoot="/mywebapp" # required for web module only
contentURI=Increment.jar
contentFile=c:/apps/app1/Increment.jar
deployEJB=false
```

  2. Run the applyConfigProperties command to update the application.

     Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Remove a single module from a deployed enterprise application.
  1. Edit the application properties file so that it specifies the Java EE module to remove.

     Edit the properties of an Application configuration object. Specify `Update=true`, `operationType=delete`, and `contentType=moduleFile` in the Properties section. The following example removes the `test.war` file from a deployed application named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
Update=true
operationType=delete
contentType=moduleFile
contentURI=test.war
```

  2. Run the applyConfigProperties command to update the application.

     Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Replace, add, or delete multiple files of a deployed enterprise application.

  This option specifies to update multiple files of an installed application by uploading a compressed file. Depending on the contents of the compressed file, a single use of this option can replace files in, add new files to, and delete files from the installed application. Each entry in the compressed file is treated as a single file and the path of the file from the root of the compressed file is treated as the relative path of the file in the installed application.

To replace a file, a file in the compressed file must have the same relative path as the file to be updated in the installed application.

To add a new file to the installed application, a file in the compressed file must have a different relative path than the files in the installed application.

The relative path of a file in the installed application is formed by concatenation of the relative path of the module (if the file is inside a module) and the relative path of the file from the root of the module separated by /.

To remove a file from the installed application, specify metadata in the compressed file using a file named `META-INF/ibm-partialapp-delete.props` at any archive scope. The `ibm-partialapp-delete.props` file must be an ASCII file that lists files to be deleted in that archive with one entry for each line. The entry can contain a string pattern such as a regular expression that identifies multiple files. The file paths for the files to be deleted must be relative to the archive path that has the `META-INF/ibm-partialapp-delete.props` file.

For more information on the metadata `.props` file to include in compressed files, see the "Replace, add, or delete multiple files" section in Preparing for application update settings.

1. Edit the application properties file so that it specifies the compressed file.

   Edit the properties of an Application configuration object. Specify `Update=true`, `operationType=update`, and `contentType=partialapp` in the Properties section. The following example uses the `myAppPartial.zip` compressed file to update a deployed application named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
Update=true
operationType=update
contentType=partialapp
contentFile= c:/temp/MyApp/myAppPartial.zip
```

2. Run the applyConfigProperties command to update the application.

   Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Replace the entire deployed enterprise application.

  This option specifies to replace the application already installed on a deployment target with a new (updated) enterprise application `.ear` file.

  1. Edit the application properties file so that it specifies the application file.

     Edit the properties of an Application configuration object. Specify `Update=true`, `operationType=update`, and `contentType=app` in the Properties section. The following example replaces the `newApp1.ear` file of a deployed application named `hello`. The `useDefaultBindings=true` property instructs the product to generate default bindings for the application.

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
Update=true
operationType=update
contentType=app
contentFile=c:/apps/app1/newApp1.ear
useDefaultBindings=true
```

When the full application is updated, the old application is uninstalled and the new application is installed. When the configuration changes are saved and subsequently synchronized, the application files are expanded on the node where application will run. If the application is running on the node while it is updated, then the application is stopped, application files are updated, and application is started.

2. Run the applyConfigProperties command to update the application.

   Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Uninstall an application from a deployment target.

  You can uninstall an application in either of two ways:

  – Specify `CreateDeleteCommandProperties=true` and run the deleteConfigProperties command.

    1. Edit the properties file so that it identifies the application but no longer identifies the deployment target. For example, specify properties such as the following to uninstall an Application configuration object named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
CreateDeleteCommandProperties=true
ResourceId=Deployment=hello
#

#
# Properties
#
Name=hello
#
```

    2. Run the deleteConfigProperties command to uninstall the application.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myApplication.props
-reportFileName report.txt]')
```

  – Specify `DELETE=true` and run the applyConfigProperties command.

    1. Edit the properties file so that it identifies the application but no longer identifies the deployment target. For example, specify properties such as the following to uninstall an Application configuration object named `hello`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
DELETE=true
ResourceId=Deployment=hello

# Properties
Name=hello
```

    2. Run the applyConfigProperties command.

       For an example, see the install step.

- Edit the deployment options of a deployed application.

  1. Edit the properties file so that it specifies new or changed deployment options.

     For example, specify properties such as the following for an Application configuration object named `appl`:

```
#
# Header
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Deployment=appl
#

#
# Properties
#
taskName=AppDeploymentOptions
row1={$(APP_INSTALL_ROOT)/$(CELL)
AppDeploymentOption.Yes
AppDeploymentOption.No
AppDeploymentOption.No
```

```
AppDeploymentOption.No
""
off .*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755
"WASX.SERV1 [x0617.27]"
AppDeploymentOption.No
AppDeploymentOption.No}
mutables={true true true true true true true false true true}
row0={installed.ear.destination
distributeApp
useMetaDataFromBinary
createMBeansForResources
reloadEnabled
reloadInterval
validateinstall
filepermission
buildVersion
allowDispatchRemoteInclude
allowServiceRemoteInclude} #readonly
```

`row1` contains current values for each property. To change a property, modify values in `row1`.

`mutables` specifies whether a given property can be changed.

`row0` specifies deployment property names.

2. Run the applyConfigProperties command to update the application.

   Running the applyConfigProperties command applies the properties file. For an example, see the install step.

- Redeploy an application.
  1. Edit the properties file so that it specifies deployment options as needed.

     For example, specify properties such as the following for an Application configuration object named `myApp`.

     `mutables` specifies whether a given property can be changed.

     `row0` specifies deployment property names.

     `row1` contains current values for each property. To change a property, modify values in `row1`.

```
#
# Header MapModulesToServers
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Deployment=!{applicationName}
#

#
# Properties
#
taskName=MapModulesToServers
mutables={false false true false false false} #readonly
row0={module uri server ModuleVersion moduletype moduletypeDisplay} # readonly
row1={"My Web Module" myWebModule.war,WEB-INF/web.xml WebSphere:cell=!{cellName},node=!{nodeName},
server=!{serverName} 14 moduletype.web "Web Module"}
row2={"My EJB module" MyEjbModule.jar,META-INF/ejb-jar.xml WebSphere:cell=!{cellName},node=!{nodeName},
server=!{serverName} 13 moduletype.ejb "EJB Module"}
#

#
# Header MapRolesToUsers
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
#
taskName=MapRolesToUsers
row0={role role.everyone role.all.auth.user role.user role.group role.all.auth.realms role.user.access.ids role.group.access.ids} #readonly
mutables={false true true true true true true true} #readonly

row1={administrator AppDeploymentOption.No AppDeploymentOption.No "adminuser" "admingroup"
 AppDeploymentOption.No "" ""}

#
# Header BindJndiForEJBNonMessageBinding
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#
```

```
#
#Properties
#
taskName=BindJndiForEJBNonMessageBinding
row0={EJBModule EJB uri JNDI ModuleVersion localHomeJndi remoteHomeJndi} #readonly
mutables={false false false true false true true} #readonly
row1={"My EJB module" myEjb myEjbModule.jar,META-INF/ejb-jar.xml myEjb 20 "" ""}
#

#
# Header MapEJBRefToEJB
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
#Properties
#
taskName=MapEJBRefToEJB
row0={module EJB uri referenceBinding class JNDI ModuleVersion} #readonly
mutables={false false false false false true false} #readonly
row1={"My EJB module" myEJB MyEjbModule.jar,META-INF/ejb-jar.xml myEJB
com.ibm.defaultapplication.Increment Increment 23}
#

#
# Header DataSourceFor20EJBModules
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
#Properties
#
taskName=DataSourceFor20EJBModules
row0={AppVersion EJBModule uri JNDI resAuth login.config.name auth.props dataSourceProps} #readonly
mutables={false false false true true true true true} #readonly
row1={13 "My EJB module" MyEjbModule.jar,META-INF/ejb-jar.xml MyDataSource
cmpBinding.perConnectionFactory "" "" ""}
#

#
# Header DataSourceFor20CMPBeans
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
#
taskName=DataSourceFor20CMPBeans
row0={AppVersion EJBVersion EJBModule EJB uri JNDI resAuth login.config.name auth.props} #readonly
mutables={false false false false false true true true true} #readonly
row1={13 13 "My EJB module" MyEjb MyEjbModule.jar,META-INF/ejb-jar.xml myDataSource
 cmpBinding.perConnectionFactory "" ""}
#

#
# Header MapWebModToVH
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
#
taskName=MapWebModToVH
row0={webModule uri virtualHost} #readonly
mutables={false false true} #readonly
row1={"My Web Application" MyWebModule.war,WEB-INF/web.xml default_host}
#

#
# Header CtxRootForWebMod
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
```

```
#
taskName=CtxRootForWebMod
row0={webModule uri web.contextroot} #readonly
mutables={false false true} #readonly
row1={"My Web Application" MyWebModule.war,WEB-INF/web.xml /}
#

#
# Header MapSharedLibForMod
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
#
taskName=MapSharedLibForMod
row0={module uri sharedLibName} #readonly
mutables={false false true} #readonly
row2={"My Web Application" MyWebModule.war,WEB-INF/web.xml ""}
row1={myApp META-INF/application.xml ""}
#

#
# Header JSPReloadForWebMod
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
#
taskName=JSPReloadForWebMod
row0={webModule uri jspReloadEnabled jspReloadInterval} #readonly
mutables={false false true true} #readonly
row1={"My Web Application" MyWebModule.war,WEB-INF/ibm-web-ext.xmi AppDeploymentOption.Yes 10}
#

#
# Header SharedLibRelationship
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
#

#
# Properties
#
taskName=SharedLibRelationship
row0={module uri relationship compUnitName matchTarget origRelationship}
#readonly
mutables={false false true true true false} #readonly
row2={"My Web Application" MyWebModule.war,WEB-INF/web.xml "" "" AppDeploymentOption.Yes ""}
row1={myApp META-INF/application.xml "" "" AppDeploymentOption.Yes ""}
#

EnvironmentVariablesSection
#
# Environment Variables
#
cellName=myCell04
applicationName=myApp
nodeName=myNode
serverName=myServer
```

In this example, only the most common tasks such as MapModulesToServer and CtxRootForWebMod are shown. You can obtain properties for tasks that are not in the example by extracting the properties of an existing application and modifying the contents of the extracted properties file to match the environment of your application.

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

– For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.

– For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

2. Run the applyConfigProperties command to update the application.

   Running the applyConfigProperties command applies the properties file. For an example, see the install step.

Results

You can use the properties file to configure and manage the application object and its properties.

What to do next

Save the changes to your configuration.

Working with application deployment properties files

You can use properties files to modify enterprise application deployment properties. An enterprise application must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Determine the changes that you want to make to your application deployment configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify an application deployment object. This task resembles editing the deployment options of an application, except it uses an ApplicationDeployment resource type instead of an Application resource type and it specifies more properties.

Run administrative commands using wsadmin to change a properties file for an application deployment, validate the properties, and apply them to your configuration.

Table 455. Actions for application deployment properties files. You can modify application deployment properties.

Action	Procedure
create	Not applicable
modify	Edit the properties file and then run the applyConfigProperties command.
delete	Not applicable
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

1. Edit an application deployment configuration.

   For example, specify properties such as the following for an ApplicationDeployment instance:

```
ResourceType=ApplicationDeployment
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=
#

#
#Properties
#
recycleOnUpdate=PARTIAL #ENUM(PARTIAL|NONE|FULL),default(PARTIAL)
allowDispatchRemoteInclude=false #boolean,default(false)
autoLink=false #boolean,default(false)
expandSynchronously=false #boolean,default(false)
zeroBinaryCopy=false #boolean,default(false)
allowServiceRemoteInclude=false #boolean,default(false)
warClassLoaderPolicy=MULTIPLE #ENUM(MULTIPLE|SINGLE),required,default(MULTIPLE)
asyncRequestDispatchType=null #ENUM(CLIENT_SIDE|DISABLED|SERVER_SIDE)
filePermission=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755"
enableDistribution=true #boolean,default(true)
deploymentId=0 #required
startingWeight=1 #integer,required,default(1)
zeroEarCopy=false #boolean,default(false)
backgroundApplication=false #boolean,default(false)
reloadInterval=3 #long,default(3)
useMetadataFromBinaries=false #boolean,default(false)
reloadEnabled=false #boolean,default(true)
appContextIDForSecurity="href:IBM-49F7FB781FENode01Cell/DefaultApplication"
createMBeansForResources=false #boolean,default(false)
binariesURL="$(APP_INSTALL_ROOT)/$(CELL)/DefaultApplication.ear" #required
startOnDistribute=false #boolean,default(false)

#
# Header ( ApplicationDeployment ClassLoader)
#
ResourceType=Classloader
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:Classloader=
#

#
#Properties
#
mode=PARENT_FIRST #ENUM(PARENT_FIRST|PARENT_LAST),default(PARENT_FIRST)


#
# Header (ApplicationDeployment ClassLoader's LibraryRef )
#
ResourceType=LibraryRef
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:Classloader=:LibraryRef=libraryName#myLibName
#

#
#Properties
#
libraryName=myLibName
sharedClassLoader=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

2. Run the applyConfigProperties command to change an application deployment configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to configure and manage the application deployment object and its properties.

What to do next

Save the changes to your configuration.

Working with web module deployment properties files:

You can use properties files to modify web module deployment properties. A web module must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Determine the changes that you want to make to your web module deployment configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify a web module deployment object.

Run administrative commands using wsadmin to change a properties file for a web module deployment, validate the properties, and apply them to your configuration.

Table 456. Actions for web module deployment properties files. You can modify web module deployment properties.

Action	Procedure
create	Not applicable
modify	Edit the properties file and then run the applyConfigProperties command.
delete	Not applicable
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Edit a web module deployment configuration.

   For example, specify properties such as the following for a WebModuleDeployment instance:

```
#
# Header
#
ResourceType=WebModuleDeployment
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:WebModuleDeployment=
uri#myWebModule.war
#

#
#Properties
#
startingWeight=10000 #integer,required,default(1)
deploymentId=1 #required
classloaderMode=PARENT_FIRST #ENUM(PARENT_FIRST|PARENT_LAST),default(PARENT_FIRST)
altDD=null
uri=myWebModule.war #required
applicationDeployment=Cell=!{cellName}:Deployment=!{applicationName2}:ApplicationDeployment=ID#
ApplicationDeployment_1183122148265 #ObjectName(ApplicationDeployment)

#
# Header ( WebDeployment ClassLoader)
#
ResourceType=Classloader
ImplementingResourceType=Application
```

```
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:WebModuleDeployment=uri
#myWebModule.war:Classloader=
#


#
#Properties
#
mode=PARENT_FIRST #ENUM(PARENT_FIRST|PARENT_LAST),default(PARENT_FIRST)


#
# Header ( WebDeployment ClassLoader's LibraryRef )
#
ResourceType=LibraryRef
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:WebModuleDeployment=uri
#myWebModule.war:Classloader=:LibraryRef=libraryName#myLibName
#

#
#Properties
#
libraryName=myLibName
sharedClassLoader=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

2. Run the applyConfigProperties command to change a web module deployment configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to configure and manage the web module deployment object and its properties.

What to do next

Save the changes to your configuration.

Working with EJB module deployment properties files:

You can use properties files to modify Enterprise JavaBeans (EJB) module deployment properties. An EJB module must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Determine the changes that you want to make to your EJB module deployment configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify an EJB module deployment object.

Run administrative commands using wsadmin to change a properties file for an EJB module deployment, validate the properties, and apply them to your configuration.

Table 457. Actions for EJB module deployment properties files. You can modify EJB module deployment properties.

Action	Procedure
create	Not applicable
modify	Edit the properties file and then run the applyConfigProperties command.
delete	Not applicable
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Edit an EJB module deployment configuration.

   For example, specify properties such as the following for an EJBModuleDeployment instance:

```
#
# Header
#
ResourceType=EJBModuleDeployment
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:
EJBModuleDeployment=uri#myEjb.jar
#

#
#Properties
#
startingWeight=5000 #integer,required,default(1)
deploymentId=1 #required
altDD=null
uri=myEJB.jar #required
applicationDeployment=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=
#ObjectName(ApplicationDeployment)

#
# Header ( EJBDeployment ClassLoader)
#
ResourceType=Classloader
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:EJBModuleDeployment=uri
#myEJB.jar:Classloader=
#

#
#Properties
#
mode=PARENT_FIRST #ENUM(PARENT_FIRST|PARENT_LAST),default(PARENT_FIRST)


#
# Header ( EJBDeployment ClassLoader's LibraryRef )
#
ResourceType=LibraryRef
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:EJBModuleDeployment=uri
#myEJB.jar:Classloader=:LibraryRef=libraryName#myLibName
#

#
#Properties
#
libraryName=myLibName
sharedClassLoader=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

2. Run the applyConfigProperties command to change an EJB module deployment configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to configure and manage the EJB module deployment object and its properties.

What to do next

Save the changes to your configuration.

Working with application configuration properties files:

You can use properties files to create, modify, or delete application configuration objects.

Before you begin

Determine the changes that you want to make to an application configuration object.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an application configuration.

Run administrative commands using wsadmin to create or change a properties file for an application configuration, validate the properties, and apply them to your configuration.

Table 458. Actions for application configuration properties files. You can create, modify, and delete application configuration properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Make required changes to properties and then run the applyConfigProperties command.
delete	To delete the entire ApplicationConfig object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for an ApplicationConfig object.
  1. Set ApplicationConfig properties as needed.

     Open an editor on an ApplicationConfig properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ApplicationConfig properties file follows:

```
#
# Header
#
ResourceType=ApplicationConfig
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:ApplicationConfig=
#DELETE=true
#
```

```
#
#Properties
#
enableSFSBFailover=false #boolean,default(false)
overrideDefaultDRSSettings=false #boolean,default(false)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

2. Run the applyConfigProperties command to create or change an application configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the application configuration that you want to change.

     You can extract a properties file for an ApplicationConfig object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the application configuration, you can delete the entire application configuration object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the application configuration object and its properties.

What to do next

Save the changes to your configuration.

Working with application configuration session manager properties files:

You can use properties files to create, modify, or delete session manager objects of an application configuration.

Before you begin

Determine the changes that you want to make to a session manager of an application configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a session manager.

Run administrative commands using wsadmin to create or change a properties file for a session manager, validate the properties, and apply them to your configuration.

Table 459. Actions for application configuration session manager properties files. You can create, modify, and delete session manager properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Make required changes to properties and then run the applyConfigProperties command.
delete	To delete the entire ApplicationConfig SessionManager object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for an ApplicationConfig SessionManager object.
  1. Set ApplicationConfig SessionManager properties as needed.

     Open an editor on an ApplicationConfig SessionManager properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example ApplicationConfig SessionManager properties file follows:

```
#
# Header (ApplicationConfig's Session Manager)
#
ResourceType=SessionManager
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:ApplicationConfig=:SessionManager=
AttributeInfo=sessionManagement
#DELETE=true
#

#
#Properties
#
enableSecurityIntegration=false #boolean,default(false)
maxWaitTime=5 #integer,default(0)
context=null
allowSerializedSessionAccess=false #boolean,default(false)
enableProtocolSwitchRewriting=false #boolean,default(false)
enableUrlRewriting=false #boolean,default(false)
enable=true #boolean,default(false)
accessSessionOnTimeout=true #boolean,default(true)
enableSSLTracking=false #boolean,default(false)
sessionPersistenceMode=NONE #ENUM(DATABASE|DATA_REPLICATION|NONE),default(NONE)
enableCookies=true #boolean,default(true)
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

  2. Run the applyConfigProperties command to create or change a session manager.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the session manager that you want to change.

     You can extract a properties file for an ApplicationConfig SessionManager object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.

- If you no longer need the session manager, you can delete the entire session manager object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the session manager object and its properties.

What to do next

Save the changes to your configuration.

Working with application configuration web module properties files:

You can use properties files to create, modify, or delete web module configuration objects of an application configuration. A web module must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Determine the changes that you want to make to a web module of an application configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a web module.

Run administrative commands using wsadmin to create or change a properties file for a web module, validate the properties, and apply them to your configuration.

Table 460. Actions for application configuration web module configuration properties files. You can create, modify, and delete web module configuration properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Make required changes to properties and then run the applyConfigProperties command.
delete	To delete the entire ApplicationConfig WebModuleConfig object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for an ApplicationConfig WebModuleConfig object.
  1. Set ApplicationConfig WebModuleConfig properties as needed.

     Open an editor on an ApplicationConfig WebModuleConfig properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ApplicationConfig WebModuleConfig properties file follows:

```
#
# Header (ApplicationConfig Session Manager)
#
ResourceType=WebModuleConfig
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:ApplicationConfig=: WebModuleConfig =myWebModule
#DELETE=true
#

#
#Properties
#
name=myWebModule
description=null
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

2. Run the applyConfigProperties command to create or change a web module.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the web module that you want to change.

     You can extract a properties file for an ApplicationConfig WebModuleConfig object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the web module, you can delete the entire web module object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the web module object and its properties.

What to do next

Save the changes to your configuration.

Working with web module configuration session manager properties files:

You can use properties files to create, modify, or delete session manager objects of a web module configuration.

Before you begin

Determine the changes that you want to make to a session manager of a web module configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a session manager.

Run administrative commands using wsadmin to create or change a properties file for a session manager, validate the properties, and apply them to your configuration.

Table 461. Actions for web module configuration session manager properties files. You can create, modify, and delete session manager properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Make required changes to properties and then run the applyConfigProperties command.
delete	To delete the entire WebModuleConfig SessionManager object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for a WebModuleConfig SessionManager object.
  1. Set WebModuleConfig SessionManager properties as needed.

     Open an editor on a WebModuleConfig SessionManager properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example WebModuleConfig SessionManager properties file follows:

```
#
# Header (WebModuleConfig's Session Manager)
#
ResourceType=SessionManager
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:ApplicationConfig=:WebModuleConfig=myWebModule:SessionManager=
AttributeInfo=sessionManagement
#DELETE=true
#

#
#Properties
#
enableSecurityIntegration=false #boolean,default(false)
maxWaitTime=5 #integer,default(0)
context=null
allowSerializedSessionAccess=false #boolean,default(false)
enableProtocolSwitchRewriting=false #boolean,default(false)
enableUrlRewriting=false #boolean,default(false)
enable=true #boolean,default(false)
accessSessionOnTimeout=true #boolean,default(true)
enableSSLTracking=false #boolean,default(false)
sessionPersistenceMode=NONE #ENUM(DATABASE|DATA_REPLICATION|NONE),default(NONE)
enableCookies=true #boolean,default(true)
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

  2. Run the applyConfigProperties command to create or change a session manager.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the session manager that you want to change.

     You can extract a properties file for a WebModuleConfig SessionManager object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

Ensure that the environment variables in the properties file match your system.

3. Run the applyConfigProperties command.

- If you no longer need the session manager, you can delete the entire session manager object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the session manager object and its properties.

What to do next

Save the changes to your configuration.

Working with application configuration EJB module properties files:

You can use properties files to create, modify, or delete Enterprise JavaBeans (EJB) module configuration objects of an application configuration. An EJB module must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Determine the changes that you want to make to an EJB module configuration of an application configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an EJB module.

Run administrative commands using wsadmin to create or change a properties file for an EJB module, validate the properties, and apply them to your configuration.

Table 462. Actions for application configuration EJB module configuration properties files. You can create, modify, and delete EJB module configuration properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Make required changes to properties and then run the applyConfigProperties command.
delete	To delete the entire ApplicationConfig EJBModuleConfig object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for an ApplicationConfig EJBModuleConfig object.
  1. Set ApplicationConfig EJBModuleConfig properties as needed.

Open an editor on an ApplicationConfig EJBModuleConfig properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ApplicationConfig EJBModuleConfig properties file follows:

```
#
# Header (ApplicationConfig EJBModuleConfig)
#
ResourceType=EJBModuleConfig
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:ApplicationConfig=: EJBModuleConfig =myEJBModule
#DELETE=true
#

#
#Properties
#
name=myEJBModule
description=null
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

2. Run the applyConfigProperties command to create or change an EJB module.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the EJB module configuration that you want to change.

     You can extract a properties file for an ApplicationConfig EJBModuleConfig object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the EJB module configuration, you can delete the entire EJB module configuration object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the EJB module configuration and its properties.

What to do next

Save the changes to your configuration.

Working with EJB module configuration session manager properties files:

You can use properties files to create, modify, or delete session manager objects of an Enterprise JavaBeans (EJB) module configuration. An EJB module must conform to Java Platform, Enterprise Edition (Java EE) specifications.

Before you begin

Determine the changes that you want to make to a session manager of an EJB module configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a session manager.

Run administrative commands using wsadmin to create or change a properties file for a session manager, validate the properties, and apply them to your configuration.

Table 463. Actions for EJB module configuration session manager properties files. You can create, modify, and delete session manager properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Make required changes to properties and then run the applyConfigProperties command.
delete	To delete the entire EJBModuleConfig SessionManager object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for an EJBModuleConfig SessionManager object.
  1. Set EJBModuleConfig SessionManager properties as needed.

     Open an editor on an EJBModuleConfig SessionManager properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example EJBModuleConfig SessionManager properties file follows:

```
#
# Header (EJBModuleConfig Session Manager)
#
ResourceType=SessionManager
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}:ApplicationDeployment=:ApplicationConfig=:EJBModuleConfig=myEJBModule:SessionManager=
AttributeInfo=sessionManagement
#DELETE=true
#

#
#Properties
#
enableSecurityIntegration=false #boolean,default(false)
maxWaitTime=5 #integer,default(0)
context=null
allowSerializedSessionAccess=false #boolean,default(false)
enableProtocolSwitchRewriting=false #boolean,default(false)
enableUrlRewriting=false #boolean,default(false)
enable=true #boolean,default(false)
accessSessionOnTimeout=true #boolean,default(true)
enableSSLTracking=false #boolean,default(false)
sessionPersistenceMode=NONE #ENUM(DATABASE|DATA_REPLICATION|NONE),default(NONE)
enableCookies=true #boolean,default(true)
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
applicationName=myApp
```

  2. Run the applyConfigProperties command to create or change a session manager.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.

1. Obtain a properties file for the session manager that you want to change.
   You can extract a properties file for an EJBModuleConfig SessionManager object using the extractConfigProperties command.
2. Open the properties file in an editor and change the properties as needed.
   Ensure that the environment variables in the properties file match your system.
3. Run the applyConfigProperties command.

- If you no longer need the session manager, you can delete the entire session manager object.
  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the session manager object and its properties.

What to do next

Save the changes to your configuration.

Working with cache provider properties files

You can use properties files to create, modify, or delete cache provider properties and custom properties.

Before you begin

Determine the changes that you want to make to your cache provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a cache provider object. You can also create, modify, or delete cache provider custom properties.

Run administrative commands using wsadmin to create or change a properties file for a cache provider, validate the properties, and apply them to your configuration.

Table 464. Actions for cache provider properties files. You can create, modify, and delete cache provider properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire CacheProvider object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit a cache provider properties file.
  1. Set CacheProvider properties as needed.

     Open an editor on a CacheProvider properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example CacheProvider properties file follows:

```
#
# Header
#
#
ResourceType=CacheProvider
ImplementingResourceType=CacheProvider
ResourceId=Cell=!{cellName}:CacheProvider=myCacheProvider
#DELETE=true
#

#
#Properties
#
classpath={}
name=myCacheProvider #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Default Cache Provider
providerType=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to change a cache provider configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the cache provider or an existing custom property, you can delete the entire cache provider object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the cache provider object and its properties.

What to do next

Save the changes to your configuration.

Working with object cache properties files

You can use properties files to create, modify, or delete object cache properties and custom properties.

Before you begin

Determine the changes that you want to make to your object cache configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an object cache instance. You can also create, modify, or delete object cache custom properties.

Run administrative commands using wsadmin to create or change a properties file for an object cache, validate the properties, and apply them to your configuration.

Table 465. Actions for object cache properties files. You can create, modify, and delete object cache properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire ObjectCacheInstance object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit an ObjectCacheInstance properties file.
  1. Set ObjectCacheInstance properties as needed.

     Open an editor on an ObjectCacheInstance properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ObjectCacheInstance properties file follows:

```
#
# Header
#
ResourceType=ObjectCacheInstance
ImplementingResourceType=ObjectCacheInstance
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:CacheProvider=myCacheProvider:
ObjectCacheInstance=jndiName#myObjectCacheJndiName
#DELETE=true
#

#
#Properties
#
diskCacheEntrySizeInMB=0 #integer,default(0)
defaultPriority=1 #integer,required,default(1)
useListenerContext=false #boolean,default(false)
pushFrequency=1 #integer,default(1)
memoryCacheSizeInMB=0 #integer,default(0)
hashSize=1024 #integer,default(1024)
providerType=null
diskCacheSizeInEntries=0 #integer,default(0)
diskOffloadLocation=null
diskCacheSizeInGB=0 #integer,default(0)
enableCacheReplication=false #boolean,default(false)
cacheSize=2000 #integer,required,default(2000)
jndiName=myObjectCacheJndiName #required
enableDiskOffload=false #boolean,required,default(false)
replicationType=NONE #ENUM(PULL|PUSH|PUSH_PULL|NONE),default(NONE)
category=null
ENUM=null
description=null
disableDependencyId=false #boolean,default(false)
#provider=CacheProvider#ObjectName(CacheProvider),readonly
diskCacheCleanupFrequency=0 #integer,default(0)
referenceable=null
flushToDiskOnStop=false #boolean,default(false)
diskCachePerformanceLevel=BALANCED #ENUM(LOW|BALANCED|HIGH|CUSTOM),default(BALANCED)
name=myObjectCache #required
```

```
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=odr
nodeName=myNode03
```

2. Run the applyConfigProperties command to change an object cache configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the object cache or an existing custom property, you can delete the entire object cache instance or the custom property.
  - To delete the entire instance, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the object cache instance and its properties.

What to do next

Save the changes to your configuration.

Working with object cache J2EE resource properties files:

You can use properties files to create, modify, or delete object cache Java 2 Platform, Enterprise Edition (J2EE) resource custom properties.

Before you begin

Determine the changes that you want to make to your object cache J2EE resource configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete object cache J2EE resource custom properties.

Run administrative commands using wsadmin to change a properties file for an object cache J2EE resource, validate the properties, and apply them to your configuration.

Table 466. Actions for object cache J2EE resource properties. You can create, modify, and delete object cache J2EE resource custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.

Table 466. Actions for object cache J2EE resource properties (continued). You can create, modify, and delete object cache J2EE resource custom properties.

Action	Procedure
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create object cache J2EE resource properties.
  1. Specify ObjectCacheInstance J2EEResourcePropertySet custom properties in a properties file.

     Open an editor and specify object cache J2EE resource properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=ObjectCacheInstance
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:CacheProvider=myCacheProvider:
ObjectCacheInstance=jndiName#myObjectCacheJndiName:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)

#
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=odr
nodeName=myNode03
```

  2. Run the applyConfigProperties command to create or change an object cache J2EE resource configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing object cache J2EE resource properties.
  1. Obtain a properties file for the object cache J2EE resource that you want to change.

     You can extract a properties file for an ObjectCacheInstance J2EEResourcePropertySet using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need an object cache J2EE resource custom property, you can delete the custom property.

  To delete one or more custom properties, specify only the properties to delete in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the object cache J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with servlet cache properties files

You can use properties files to create, modify, or delete servlet cache properties and custom properties.

Before you begin

Determine the changes that you want to make to your servlet cache configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a servlet cache instance. You can also create, modify, or delete servlet cache custom properties.

Run administrative commands using wsadmin to create or change a properties file for a servlet cache, validate the properties, and apply them to your configuration.

Table 467. Actions for servlet cache properties files. You can create, modify, and delete servlet cache properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire ServletCacheInstance object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create or edit a servlet cache properties file.
   a. Set ServletCacheInstance properties as needed.

      Open an editor on a ServletCacheInstance properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ServletCacheInstance properties file follows:

```
#
# Header
#
ResourceType=ServletCacheInstance
ImplementingResourceType=ServletCacheInstance
ResourceId=Cell=!{cellName}:CacheProvider=myCacheProvider:ServletCacheInstance=jndiName#myServletCacheJndiName
#DELETE=true
#
```

```
#
#Properties
#
diskCacheEntrySizeInMB=0 #integer,default(0)
defaultPriority=1 #integer,required,default(1)
useListenerContext=false #boolean,default(false)
pushFrequency=1 #integer,default(1)
memoryCacheSizeInMB=0 #integer,default(0)
hashSize=1024 #integer,default(1024)
providerType=null
diskCacheSizeInEntries=0 #integer,default(0)
diskOffloadLocation=null
diskCacheSizeInGB=0 #integer,default(0)
enableCacheReplication=false #boolean,default(false)
cacheSize=2000 #integer,required,default(2000)
jndiName=myServletCacheJndiName #required
enableDiskOffload=false #boolean,required,default(false)
replicationType=NONE #ENUM(PULL|PUSH|PUSH_PULL|NONE),default(NONE)
category=null
description=null
#provider=CacheProvider#ObjectName(CacheProvider),readonly
diskCacheCleanupFrequency=0 #integer,default(0)
referenceable=null
flushToDiskOnStop=false #boolean,default(false)
diskCachePerformanceLevel=BALANCED #ENUM(LOW|BALANCED|HIGH|CUSTOM),default(BALANCED)
name=myServletCache #required
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

b. Run the applyConfigProperties command to create or change a servlet cache configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. If you no longer need the servlet cache or an existing custom property, you can delete the entire servlet cache object or the custom property.
   - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

   - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the servlet cache object and its properties.

What to do next

Save the changes to your configuration.

Working with eviction policy properties files

You can use properties files to create, modify, or delete eviction policy properties and custom properties.

Before you begin

Determine the changes that you want to make to your eviction policy configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an eviction policy instance. You can also create, modify, or delete eviction policy custom properties.

Run administrative commands using wsadmin to create or change a properties file for an eviction policy, validate the properties, and apply them to your configuration.

Table 468. Actions for eviction policy properties files. You can create, modify, and delete eviction policy properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire eviction policy, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit an eviction policy properties file.
  1. Set eviction policy properties as needed.

     Open an editor on a MemoryCacheEvictionPolicy or DiskCacheEvictionPolicy properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example eviction policy properties file follows:

```
#
#  MemoryCacheEvictionPolicy
#
ResourceType=MemoryCacheEvictionPolicy
ImplementingResourceType=ServletCacheInstance
ResourceId=Cell=!{cellName}:CacheProvider=myCacheProvider:ServletCacheInstance=jndiName#myServletCacheJndiName:MemoryCacheEvictionPolicy=
AttributeInfo=memoryCacheEvictionPolicy
#DELETE=true
#

#
#Properties
#
lowThreshold=80 #integer,default(80)
highThreshold=95 #integer,default(95)


#
# DiskCacheEvictionPolicy
#
ResourceType=DiskCacheEvictionPolicy
ImplementingResourceType=ServletCacheInstance
ResourceId=Cell=!{cellName}:CacheProvider=myCacheProvider:ServletCacheInstance=jndiName#myServletCacheJndiName:DiskCacheEvictionPolicy=
AttributeInfo=diskCacheEvictionPolicy
#DELETE=true
#

#
#Properties
#
algorithm=NONE #ENUM(RANDOM|SIZE|NONE),default(NONE)
lowThreshold=90 #integer,default(90)
highThreshold=95 #integer,default(95)

#
EnvironmentVariablesSection
```

```
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create or change an eviction policy configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the eviction policy or an existing custom property, you can delete the entire eviction policy object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage an eviction policy object and its properties.

What to do next

Save the changes to your configuration.

Working with data replication service properties files

You can use properties files to create, modify, or delete data replication service (DRS) properties and custom properties.

Before you begin

Determine the changes that you want to make to your DRS configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a DRS instance. You can also create, modify, or delete servlet cache custom properties.

Run administrative commands using wsadmin to create or change a properties file for a DRS instance, validate the properties, and apply them to your configuration.

Table 469. Actions for DRS properties files. You can create, modify, and delete DRS properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire DRSSettings object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 470. Actions for DRS custom properties. You can create, modify, and delete DRS custom properties.

Action	Procedure
create	Not applicable
modify	Run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Run the applyConfigProperties command to create a custom property.
delete Property	Run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for a DRSSettings object.
  1. Set DRSSettings properties as needed.

     Open an editor on a DRSSettings properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example DRSSettings properties file follows:

```
#
# Header
#
ResourceType=DRSSettings
ImplementingResourceType=ServletCacheInstance
ResourceId=Cell=!{cellName}:CacheProvider=myCacheProvider:ServletCacheInstance=jndiName#myServletCacheJndiName:DRSSettings=
AttributeInfo=cacheReplication
#DELETE=true
#

#
#Properties
#
overrideHostConnectionPoints={}
ids={} #integer*
messageBrokerDomainName=null
dataReplicationMode=BOTH #ENUM(SERVER|CLIENT|BOTH),default(BOTH)
preferredLocalDRSBrokerName=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to create or change a DRS configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the DRSSettings that you want to change.

     You can extract a properties file for a DRSSettings object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.

     To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=DRSSettings
ImplementingResourceType=ServletCacheInstance
ResourceId=Cell=!{cellName}:CacheProvider=myCacheProvider:ServletCacheInstance=jndiName#myServletCacheJndiName:DRSSettings=
AttributeInfo=properties(name,value)
#
```

```
#Properties
#
existingProp=newValue
newProp=value

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
```

3. Run the applyConfigProperties command.

- If you no longer need the servlet cache or an existing custom property, you can delete the entire servlet cache object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the DRS object and its properties.

What to do next

Save the changes to your configuration.

Working with data replication domain properties files

You can use properties files to create, modify, or delete data replication domain properties and custom properties.

Before you begin

Determine the changes that you want to make to your data replication domain configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a data replication domain object. You can also create, modify, or delete data replication domain custom properties.

Run administrative commands using wsadmin to create or change a properties file for a data replication domain, validate the properties, and apply them to your configuration.

Table 471. Actions for data replication domain properties files. You can create, modify, and delete data replication domain properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire DataReplicationDomain object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.

Table 471. Actions for data replication domain properties files (continued). You can create, modify, and delete data replication domain properties.

Action	Procedure
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create or edit a data replication domain properties file.
   a. Set DataReplicationDomain properties as needed.

      Open an editor on an DataReplicationDomain properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example DataReplicationDomain properties file follows:

```
#
# Header
#
ResourceType=DataReplicationDomain
ImplementingResourceType=DataReplicationDomain
ResourceId=Cell=!{cellName}:DataReplicationDomain=myDRDomain
#DELETE=true
#

#
#Properties
#
name=myDRDomain #required

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=IBM-49F7FB781FECell04
```

   b. Run the applyConfigProperties command to create or change a data replication domain configuration.

      Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. If you no longer need the data replication domain or an existing custom property, you can delete the entire data replication domain object or the custom property.
   - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

   - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the data replication domain object and its properties.

What to do next

Save the changes to your configuration.

Working with data replication properties files

You can use properties files to create, modify, or delete data replication properties and custom properties.

Before you begin

Determine the changes that you want to make to your data replication configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a data replication object. You can also create, modify, or delete data replication custom properties.

Run administrative commands using wsadmin to create or change a properties file for a data replication, validate the properties, and apply them to your configuration.

Table 472. Actions for data replication properties files. You can create, modify, and delete data replication properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire DataReplication object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

1. Create a data replication properties file.
   a. Set DataReplication properties as needed.

      Open an editor on a DataReplication properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example DataReplication properties file follows:

```
#
# Header
#
ResourceType=DataReplication
ImplementingResourceType=DataReplicationDomain
ResourceId=Cell=!{cellName}:DataReplicationDomain=myDRDomain:DataReplication=messageBrokerName#myMessageBroker
AttributeInfo=defaultDataReplicationSettings
#DELETE=true
#

#
#Properties
#
password=null
userId=null
requestTimeout=5 #integer,default(5)
encryptionKeyValue=null
useSSL=false #boolean,default(false)
numberOfReplicas=1 #integer,default(1)
messageBrokerName=myMessageBroker
encryptionType=NONE #ENUM(DES|TRIPLE_DES|NONE),default(NONE)
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

b. Run the applyConfigProperties command to create or change a data replication configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. Modify an existing properties file.
   a. Obtain a properties file for the data replication object that you want to change.

      You can extract a properties file for a DataReplication object using the extractConfigProperties command.

   b. Open the properties file in an editor and change the properties as needed.

      Ensure that the environment variables in the properties file match your system.

   c. Run the applyConfigProperties command.
3. If you no longer need the data replication or an existing custom property, you can delete the entire data replication object or the custom property.
   - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

   - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the data replication object and its properties.

What to do next

Save the changes to your configuration.

Working with J2C resource adapter properties files

You can use properties files to install or remove Java 2 Platform, Enterprise Edition (J2EE) Connector Architecture (J2C) resource adapters.

Before you begin

Determine the changes that you want to make to your J2C resource adapter configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can install or remove a J2C resource adapter.

Run administrative commands using wsadmin to apply or delete a properties file for a J2C resource adapter.

Table 473. Actions for J2C resource adapter properties files. You can create and delete J2C resource adapter properties.

Action	Procedure
create (install)	1. Create a properties file that sets required properties for a J2C resource adapter and that contains the following lines in the header: `CreateDeleteCommandProperties=true` `commandName=installResourceAdapter` 2. Run the applyConfigProperties command to install the resource adapter.
modify	Not applicable
delete (remove)	To remove the entire resource adapter, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Install a J2CResourceAdapter instance.
  1. Create a properties file for a J2CResourceAdapter instance.

     Open an editor and create a file such as the following for a J2CResourceAdapter instance:

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=installResourceAdapter
#

#
#Properties
#
rarPath=myRA.rar #String,required
rar.nativePath=null #String
rar.archivePath=null #String
rar.propertiesSet=null #java.util.Properties
rar.name=null #String
rar.DeleteSourceRar=null #Boolean
rar.classpath=null #String
rar.HACapability=null #String
nodeName=!{nodeName} #String,required
rar.desc=null #String
rar.enableHASupport=null #Boolean
rar.threadPoolAlias=null #String
rar.isolatedClassLoader=null #Boolean#

EnvironmentVariablesSection
#
#Environment Variables
nodeName=myNode
```

  2. Run the applyConfigProperties command to install a J2C resource adapter configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Remove the entire J2C resource adapter instance.
  1. Specify `DELETE=true` in the header section of the properties file.

     A J2C resource adapter properties file with `DELETE=true` resembles the following:

```
#
# Header
#
ResourceType=J2CResourceAdapter
ImplementingResourceType=J2CResourceAdapter
ResourceId=Cell=!{cellName}: J2CResourceAdapter=myJ2CResourceAdapter
DELETE=true
```

```
#

#
#Properties
#
name=myJ2CResourceAdapter
#
EnvironmentVariablesSection
#
#Environment Variables
cellName=myCell
```

2. Run the deleteConfigProperties command.

   The following Jython example uses the optional `-reportFileName` parameter and produces a report named `report.txt`:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the J2C resource adapter.

What to do next

Save the changes to your configuration.

Working with J2EEResourceProperty properties files

You can use properties files to create or change J2EEResourceProperty properties.

Before you begin

Determine the changes that you want to make to your Java 2 Platform, Enterprise Edition (J2EE) resource property configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a J2EEResourceProperty object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a J2EE resource property, validate the properties, and apply them to your configuration.

Table 474. Actions for J2EEResourceProperty properties files. You can create, modify, and delete J2EEResourceProperty configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a J2EE resource property.
  1. Create a properties file for a J2EEResourceProperty object.

Open an editor and create a J2EEResourceProperty properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

The following example properties file defines a J2EEResourceProperty named `myPropName` with a value of `myVal` inside a J2CResourceAdapter named `WebSphere Relational Resource Adapter` at the cell scope:

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:J2CResourceAdapter=WebSphere Relational Resource Adapter:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#DELETE=true
#key = name

#
#Properties
myPropName=myVal

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create a J2EEResourceProperty configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing J2EE resource property.
  1. Obtain a properties file for the J2EE resource property that you want to change.

     You can extract a properties file for a J2EEResourceProperty object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete a J2EE resource property.

  To delete the entire J2EEResourceProperty object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the J2EEResourceProperty object.

What to do next

Save the changes to your configuration.

Working with J2EEResourcePropertySet properties files

You can use properties files to create or change J2EEResourcePropertySet properties.

Before you begin

Determine the changes that you want to make to your Java 2 Platform, Enterprise Edition (J2EE) resource property set configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a J2EEResourcePropertySet object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a J2EE resource property set, validate the properties, and apply them to your configuration.

Table 475. Actions for J2EEResourcePropertySet properties files. You can create, modify, and delete J2EEResourcePropertySet configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a J2EE resource property set.
  1. Create a properties file for a J2EEResourcePropertySet object.

     Open an editor and create a properties file for a J2EE resource property set. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for a J2EEResourcePropertySet object under J2CResourceAdapter named `WebSphere Relational Resource Adapter` at cell scope follows:

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:J2CResourceAdapter=WebSphere Relational Resource Adapter:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#DELETE=true

#
#Properties
#

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

  2. Run the applyConfigProperties command to create a J2EE resource property set.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify a J2EE resource property set.
  1. Obtain a properties file for the J2EE resource property set that you want to change.

     You can extract a properties file for a J2EEResourcePropertySet object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete a J2EE resource property set.

To delete the entire J2EEResourcePropertySet object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the J2EEResourcePropertySet object.

What to do next

Save the changes to your configuration.

Working with JDBC provider properties files

You can use properties files to create, modify, or delete JDBC provider properties.

Before you begin

Determine the changes that you want to make to your JDBC provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a JDBC provider object.

Run administrative commands using wsadmin to apply a properties file for a JDBC provider to your configuration, validate the properties, or delete them.

Table 476. Actions for JDBC provider properties files. You can create, modify, and delete JDBC provider properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command.
delete	To delete the entire JDBCProvider object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a JDBCProvider properties file.
  1. Set JDBCProvider properties as needed.

     Open an editor on a JDBCProvider properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example JDBCProvider properties file follows:

```
#
# Header
#
ResourceType=JDBCProvider
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=myJDBCProvider
#DELETE=true
```

```
#

#
#Properties
#
classpath={${DERBY_JDBC_DRIVER_PATH}/derby.jar}
name=myJDBCProvider
implementationClassName=org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
nativepath={}
description=Derby embedded non-XA  JDBC Provider
#providerType=Derby JDBC Provider #readonly
xa=false #boolean

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

2. Run the applyConfigProperties command to create or change a JDBC provider configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the JDBC provider that you want to change.

     You can extract a properties file for a JDBCProvider object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the JDBC provider, you can delete the entire JDBC provider object.

  Specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the JDBC provider object and its properties.

What to do next

Save the changes to your configuration.

Working with data source properties files

You can use properties files to create, modify, or delete data source properties.

Before you begin

Determine the changes that you want to make to your data source configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a data source object.

Run administrative commands using wsadmin to apply a properties file for a data source to your configuration, validate the properties, or delete them.

Table 477. Actions for data source properties files. You can create, modify, and delete data source properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command.
delete	To delete the entire DataSource object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for a DataSource instance.
  1. Set DataSource properties as needed.

     Open an editor on a DataSource properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example DataSource properties file follows:

```
#
# Header
#
ResourceType=DataSource
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JDBCProvider=myJDBCProvider:DataSource=jndiName#myDataSourceJNDI
#DELETE=true
#

#
#Properties
#
name=myDataSource
category=null
datasourceHelperClassname=com.ibm.websphere.rsadapter.DerbyDataStoreHelper
authMechanismPreference=BASIC_PASSWORD #ENUM(BASIC_PASSWORD|KERBEROS)
statementCacheSize=10 #integer
#providerType=Derby JDBC Provider #readonly
jndiName=myDataSourceJNDI
relationalResourceAdapter=WebSphere Relational Resource Adapter #ObjectName(J2CResourceAdapter)
xaRecoveryAuthAlias=null
diagnoseConnectionUsage=false #boolean
authDataAlias=null
manageCachedHandles=false #boolean
#provider=Derby JDBC Provider #ObjectName(JDBCProvider),readonly
description=Datasource for the WebSphere Default Application
logMissingTransactionContext=true #boolean

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

  2. Run the applyConfigProperties command to create or change a data source configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the data source that you want to change.

     You can extract a properties file for a DataSource object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.

- If you no longer need the data source, you can delete the entire data source object.

  Specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the data source object and its properties.

What to do next

Save the changes to your configuration.

Working with connection pool properties files:

You can use properties files to create, modify, or delete connection pool properties of a data source.

Before you begin

Determine the changes that you want to make to your data source configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a connection pool object.

Run administrative commands using wsadmin to apply a properties file for a connection pool to your configuration, validate the properties, or delete them.

Table 478. Actions for connection pool properties files. You can create, modify, and delete connection pool properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command.
delete	To delete the entire ConnectionPool object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for a ConnectionPool instance.
  1. Set ConnectionPool properties as needed.

     Open an editor on a ConnectionPool properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example ConnectionPool properties file follows:

```
#
# Header
#
ResourceType=ConnectionPool
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:Node=!{nodeName2}:Server=!{serverName2}:JDBCProvider=myJDBCProvider:
DataSource=jndiName#myDataSourceJNDI:ConnectionPool=
AttributeInfo=connectionPool
#

#
#Properties
#
stuckThreshold=0 #integer
unusedTimeout=1800 #long
maxConnections=10 #integer
stuckTimerTime=0 #integer
testConnectionInterval=0 #integer
minConnections=1 #integer
surgeThreshold=-1 #integer
connectionTimeout=180 #long
purgePolicy=EntirePool #ENUM(EntirePool|FailingConnectionOnly)
surgeCreationInterval=0 #integer
numberOfUnsharedPoolPartitions=0 #integer
stuckTime=0 #integer
agedTimeout=0 #long
reapTime=180 #long
testConnection=false #boolean
numberOfSharedPoolPartitions=0 #integer
freePoolDistributionTableSize=0 #integer
numberOfFreePoolPartitions=0 #integer

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

2. Run the applyConfigProperties command to create or change a connection pool configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the connection pool that you want to change.

     You can extract a properties file for a ConnectionPool object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the connection pool, you can delete the entire connection pool object.

  Specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the connection pool object and its properties.

What to do next

Save the changes to your configuration.

Working with data source J2EE resource properties files:

You can use properties files to create or delete properties of data source Java 2 Platform, Enterprise Edition (J2EE) resource property sets.

Before you begin

Determine the changes that you want to make to your data source J2EE resource property set configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create or delete data source J2EE resource properties.

Run administrative commands using wsadmin to change a properties file for a data source J2EE resource, validate the properties, and apply them to your configuration.

Table 479. Actions for properties of data source J2EE resource property sets. You can create and delete data source J2EE resource properties.

Action	Procedure
create	Not applicable
modify	Not applicable
delete	Not applicable
create Property	Add a new property to the properties file and then run the applyConfigProperties command.
delete Property	Edit the properties file so that it lists only the properties to be deleted and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create data source J2EE resource properties.
  1. Specify DataSource J2EEResourcePropertySet properties in a properties file.

     You can extract a properties file for a DataSource J2EEResourcePropertySet object using the extractConfigProperties command. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system. To specify a property, edit the `AttributeInfo` value and properties values.

     The following example of a DataSource J2EEResourcePropertySet properties file shows new properties in bold:

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=JDBCProvider
ResourceId=Cell=!{cellName}:Node=!{nodeName2}:Server=!{serverName2}:JDBCProvider=myJDBCProvider:
DataSource=jndiName#myDataSourceJNDI:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#

#
#Properties
#
connectionAttributes=upgrade=true #String
shutdownDatabase= #String
description= #String
dataSourceName= #String
databaseName=${APP_INSTALL_ROOT}/${CELL}/DefaultApplication.ear/DefaultDB #String
createDatabase= #String
newProperty=newValue #String
newIntProp=10 #integer
```

```
EnvironmentVariablesSection
#Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

2. Run the applyConfigProperties command to create or change a data source J2EE resource configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need a data source J2EE resource property, you can delete the property.

  Specify only the property to be deleted in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the data source J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with CMP connection factory properties files

You can use properties files to create, modify, or delete container managed persistence (CMP) connection factory properties.

Before you begin

Determine the changes that you want to make to your CMP connection factor configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a CMP connection factory object.

Run administrative commands using wsadmin to apply a properties file for a CMP connection factory to your configuration, validate the properties, or delete them.

Table 480. Actions for CMP connection factory properties files. You can create, modify, and delete CMP connection factory properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command.
delete	To delete the entire CMPConnectorFactory object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for a CMPConnectorFactory instance.
  1. Set CMPConnectorFactory properties as needed.

     Open an editor on a CMPConnectorFactory properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example CMPConnectorFactory properties file follows:

```
#
# Header
#
ResourceType=CMPConnectorFactory
ImplementingResourceType=J2CResourceAdapter
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:J2CResourceAdapter=myJ2CResourceAdapter:
CMPConnectorFactory=jndiName#myCFJNDI
#DELETE=true
#

#
#Properties
#
name=myCF #required
connectionDefinition=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:J2CResourceAdapter=
myJ2CResourceAdapter:Connector=:ResourceAdapter=:OutboundResourceAdapter=:ConnectionDefinition=
connectionFactoryImplClass#com.ibm.ws.rsadapter.cci.WSRdbConnectionFactoryImpl #ObjectName
(ConnectionDefinition),required
category=null
authMechanismPreference=BASIC_PASSWORD #ENUM(BASIC_PASSWORD|KERBEROS),default(BASIC_PASSWORD)
providerType=null
jndiName=myCFJNDI #required
diagnoseConnectionUsage=false #boolean,default(false)
xaRecoveryAuthAlias=null
authDataAlias=null
manageCachedHandles=false #boolean,default(false)
description=null
logMissingTransactionContext=true #boolean,default(true)
#

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

  2. Run the applyConfigProperties command to create or change a CMP connection factory configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the CMP connection factory that you want to change.

     You can extract a properties file for a CMPConnectorFactory object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the CMP connection factory, you can delete the entire CMP connection factory object.

  Specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the CMP connection factory object and its properties.

What to do next

Save the changes to your configuration.

Working with CMP connection factory mapping module properties files:

You can use properties files to create, modify, or delete mapping module properties of container managed persistence (CMP) connection factories.

Before you begin

Determine the changes that you want to make to your CMP connection factory mapping module configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a CMP connection factory mapping module object.

Run administrative commands using wsadmin to apply a properties file for a mapping module to your configuration, validate the properties, or delete them.

Table 481. Actions for CMP connection factory mapping module properties files. You can create, modify, and delete mapping module properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command.
delete	To delete the entire CMPConnectorFactory MappingModule object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create a properties file for a CMPConnectorFactory MappingModule instance.
  1. Set CMPConnectorFactory MappingModule properties as needed.

     Open an editor on a CMPConnectorFactory MappingModule properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example CMPConnectorFactory MappingModule properties file follows:

```
#
# Header
#
ResourceType=MappingModule
ImplementingResourceType=J2CResourceAdapter
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:J2CResourceAdapter=myJ2CResourceAdapter:
CMPConnectorFactory=jndiName#myCFJNDI:MappingModule=
AttributeInfo=mapping
#DELETE=true
#
#Properties
#
```

```
authDataAlias=myADA
mappingConfigAlias=myMCA

#

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

2. Run the applyConfigProperties command to create or change a mapping module configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the mapping module that you want to change.

     You can extract a properties file for a CMPConnectorFactory MappingModule object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the mapping module, you can delete the entire mapping module object.

  Specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the mapping module object and its properties.

What to do next

Save the changes to your configuration.

Working with JVM properties files

You can use properties files to modify or delete Java virtual machine (JVM) properties.

Before you begin

Determine the changes that you want to make to your JVM configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete JVM properties.

Run administrative commands using wsadmin to change a properties file for a JVM, validate the properties, and apply them to your configuration.

Table 482. Actions for JVM properties. You can modify or delete JVM properties.

Action	Procedure
create	Not applicable

Table 482. Actions for JVM properties (continued). You can modify or delete JVM properties.

Action	Procedure
modify	Edit JVM properties in the properties file and then run the applyConfigProperties command. The product ignores null or empty list {} property values, and applies only those values that are not null and non-empty.
delete	Edit the JVM properties file so that it contains only those properties to delete and then run the deleteConfigProperties command. Deleting a property sets the property value to a default value, if it exists. If a default value does not exist for the property, the product removes the property.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify an existing properties file.
  1. Obtain a properties file for the JVM that you want to change.

     You can extract a properties file for a JavaVirtualMachine object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.

     An example JavaVirtualMachine properties file follows:

```
#
# Header
#
ResourceType=JavaVirtualMachine
ImplementingResourceType=Server
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:JavaVirtualMachine=
AttributeInfo=jvmEntries
#

#
#Properties
#
internalClassAccessMode=ALLOW #ENUM(ALLOW|RESTRICT),default(ALLOW)
JavaHome="C:\cf50922.30\test/java" #readonly
debugArgs="-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777"
classpath={}
initialHeapSize=0 #integer,default(0)
runHProf=false #boolean,default(false)
genericJvmArguments=
hprofArguments=
osName=null
bootClasspath={}
verboseModeJNI=false #boolean,default(false)
maximumHeapSize=0 #integer,default(0)
disableJIT=false #boolean,default(false)
verboseModeGarbageCollection=false #boolean,default(false)
executableJarFileName=null
verboseModeClass=false #boolean,default(false)
debugMode=false #boolean,default(false)


#
# Header JVM System properties
#
ResourceType=JavaVirtualMachine
ImplementingResourceType=Server
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:JavaVirtualMachine=
AttributeInfo=systemProperties(name,value)
#

#
#Properties
#
com.ibm.security.krb5.Krb5Debug=off
com.ibm.security.jgss.debug=off
```

```
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
nodeName=myNode
serverName=myServer
```

3. Run the applyConfigProperties command to create or change a JVM configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. The product ignores null or empty list {} property values, and applies only those values that are not null and non-empty. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need a JVM property, you can delete the property, provided the property does not have a default value.

  To delete one or more properties, specify only the properties to be deleted in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  If a property has a default value, the property is not deleted but rather is set to the default value.

Results

You can use the properties file to configure and manage the JVM properties.

What to do next

Save the changes to your configuration.

Working with JMS provider properties files

You can use properties files to create or change Java Message Service (JMS) provider properties.

Before you begin

Determine the changes that you want to make to your JMS provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a JMS provider object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a JMS provider, validate the properties, and apply them to your configuration.

Table 483. Actions for JMS provider properties files. You can create, modify, and delete JMS provider configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a JMS provider and its properties.
  1. Create a properties file for a JMSProvider object.

     Open an editor and create a JMS provider properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for a JMSProvider object named `myProvider` at cell scope follows:

```
#
# Header
#
ResourceType=JMSProvider
ImplementingResourceType=JMSProvider
ResourceId=Cell=!{cellName}:JMSProvider=myProvider
#DELETE=true

#
#Properties
#
supportsASF=true #boolean
classpath={}
name=myProvider
externalProviderURL=myEPURL
nativepath={}
description=WebSphere MQ Messaging Provider
providerType=null #readonly
externalInitialContextFactory=myEICFactory

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

  2. Run the applyConfigProperties command to create a JMS provider configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing JMS provider.
  1. Obtain a properties file for the JMS provider that you want to change.

     You can extract a properties file for a JMSProvider object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a JMS provider configuration.
- If you no longer need a JMS provider, you can delete the entire JMS provider object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the JMS provider object.

What to do next

Save the changes to your configuration.

Working with WebSphere MQ topic properties files

You can use properties files to create or change WebSphere MQ topic properties.

Before you begin

Determine the changes that you want to make to your WebSphere MQ topic configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a WebSphere MQ topic object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a WebSphere MQ topic, validate the properties, and apply them to your configuration.

Table 484. Actions for WebSphere MQ topic properties files. You can create, modify, and delete WebSphere MQ topic configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create a WebSphere MQ topic and its properties.
  1. Create a properties file for an MQTopic object.

     Open an editor and create an MQ topic properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for an MQTopic object with a JNDI name of `topicJndiName` under a JMSProvider named `WebSphere MQ JMS Provider` at the cell scope follows:

```
#
# Header
#
ResourceType=MQTopic
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:JMSProvider=WebSphere MQ JMS Provider:MQTopic=jndiName#topicJndiName
#DELETE=true
#
#Properties
#
wildcardFormat=topicWildcards #ENUM(characterWildcards|allWildcards|topicWildcards),default(topicWildcards)
readAhead=YES #ENUM(QUEUE_DEFINED|YES|NO),default(YES)
specifiedExpiry=0 #long,default(0)
baseTopicName=topicBaseTopicName #required
brokerPubQmgr=null
providerType=null #readonly
decimalEncoding=Normal #ENUM(Normal|Reversed),default(Normal)
expiry=APPLICATION_DEFINED #ENUM(APPLICATION_DEFINED|SPECIFIED|UNLIMITED),default(APPLICATION_DEFINED)
sendAsync=YES #ENUM(QUEUE_DEFINED|YES|NO),default(YES)
integerEncoding=Normal #ENUM(Normal|Reversed),default(Normal)
brokerPubQueue=null
specifiedPriority=0 #integer,default(0)
jndiName=topicJndiName #required
CCSID=0 #integer,default(0)
category=null
description=null
brokerVersion=V1 #ENUM(V2|V1),default(V1)
brokerCCDurSubQueue=null
brokerDurSubQueue=null
boolean=null
targetClient=JMS #ENUM(JMS|MQ),default(JMS)
priority=APPLICATION_DEFINED #ENUM(APPLICATION_DEFINED|SPECIFIED|QUEUE_DEFINED),default(APPLICATION_DEFINED)
provider=WebSphere MQ JMS Provider #ObjectName(JMSProvider),readonly
persistence=APPLICATION_DEFINED #ENUM(APPLICATION_DEFINED|HIGH|NONPERSISTENT|PERSISTENT|QUEUE_DEFINED),default(APPLICATION_DEFINED)
floatingPointEncoding=IEEENormal #ENUM(IEEENormal|IEEEReversed|S390),default(IEEENormal)
```

```
readAheadClose=DELIVERALL #ENUM(DELIVERCURRENT|DELIVERALL),default(DELIVERALL)
multicast=AS_CF #ENUM(AS_CF|DISABLED|NOT_RELIABLE|RELIABLE|ENABLED),default(AS_CF)
name=topicName #required

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create an MQTopic configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing WebSphere MQ topic.
  1. Obtain a properties file for the WebSphere MQ topic that you want to change.

     You can extract a properties file for an MQTopic using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a WebSphere MQ topic configuration.
- If you no longer need a WebSphere MQ topic, you can delete the entire WebSphere MQ topic object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the WebSphere MQ topic object.

What to do next

Save the changes to your configuration.

Working with WebSphere MQ topic connection factory properties files

You can use properties files to create or change WebSphere MQ topic connection factory properties.

Before you begin

Determine the changes that you want to make to your WebSphere MQ topic connection factory configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a WebSphere MQ topic connection factory object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a WebSphere MQ topic connection factory, validate the properties, and apply them to your configuration.

Table 485. Actions for WebSphere MQ topic connection factory properties files. You can create, modify, and delete WebSphere MQ topic connection factory configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.

Table 485. Actions for WebSphere MQ topic connection factory properties files (continued). You can create, modify, and delete WebSphere MQ topic connection factory configuration properties.

Action	Procedure
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create an WebSphere MQ topic connection factory and its properties.
  1. Create a properties file for an MQTopicConnectionFactory object.

     Open an editor and create a WebSphere MQ topic connection factory properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for an MQTopicConnectionFactory object with a name of `myName` and a JNDI name of `tcfJndiName` under a JMSProvider named `WebSphere MQ JMS Provider` at cell scope follows. The example includes an associated ConnectionPool (connectionPool attribute of MQTopicConnectionFactory).

```
#
# Header
#
ResourceType=MQTopicConnectionFactory
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:JMSProvider=WebSphere MQ JMS Provider:MQTopicConnectionFactory=jndiName#tcfJndiName
#DELETE=true
#
#Properties
#
wmqServerSvrconnChannel=null
diagnoseConnectionUsage=false #boolean,default(false)
CCSID=null
rcvExit=null
wmqServerName=null
sslConfiguration=null
pubSubCleanup=SAFE #ENUM(STRONG|SAFE|ASPROP|NONE),default(SAFE)
host=null
category=null
brokerControlQueue=null
tempModel=null
directAuth=BASIC #ENUM(BASIC|CERTIFICATE),default(BASIC)
secExit=null
proxyPort=0 #integer,default(0)
name=tcfName #required
xaRecoveryAuthAlias=null
description=null
brokerCCSubQ=null
wildcardFormat=topicWildcards #ENUM(characterWildcards|allWildcards|topicWildcards),default(topicWildcards)
useConnectionPooling=true #boolean,default(true)
pubSubCleanupInterval=3600000 #long,default(3600000)
manageCachedHandles=false #boolean,default(false)
wmqServerEndpoint=null
provider=WebSphere MQ JMS Provider #ObjectName(JMSProvider),readonly
ccdtUrl=null
providerVersion=null
sslResetCount=0 #integer,default(0)
secExitInitData=null
pollingInterval=5000 #integer,default(5000)
tempTopicPrefix=null
brokerPubQueue=null
proxyHostName=null
brokerVersion=MQSI #ENUM(MA0C|MQSI),default(MQSI)
queueManager=null
jndiName=tcfJndiName #required
sendExit=null
authMechanismPreference=BASIC_PASSWORD #ENUM(BASIC_PASSWORD|KERBEROS),default(BASIC_PASSWORD)
sendExitInitData=null
multicast=DISABLED #ENUM(DISABLED|NOT_RELIABLE|RELIABLE|ENABLED),default(DISABLED)
maxBatchSize=10 #integer,default(10)
statRefreshInterval=60000 #integer,default(60000)
XAEnabled=true #boolean,default(true)
providerType=null #readonly
rcvExitInitData=null
channel=null
port=0 #integer,default(0)
authDataAlias=null
```

```
sslPeerName=null
rescanInterval=5000 #integer,default(5000)
compressHeaders=NONE #ENUM(SYSTEM|NONE),default(NONE)
failIfQuiesce=true #boolean,default(true)
brokerSubQueue=null
clientID=null
localAddress=null
publishAckInterval=25 #integer,default(25)
brokerQueueManager=null
sslCRL=null
sparseSubscriptions=false #boolean,default(false)
substore=MIGRATE #ENUM(QUEUE|MIGRATE|BROKER),default(MIGRATE)
logMissingTransactionContext=true #boolean,default(true)
transportType=BINDINGS #ENUM(BINDINGS_THEN_CLIENT|DIRECTHTTP|BINDINGS|DIRECT|CLIENT),default(BINDINGS)
sslCipherSuite=null
compressPayload=NONE #ENUM(ZLIBHIGH|ZLIBFAST|RLE|NONE),default(NONE)
qmgrType=QMGR #ENUM(QSG|QMGR),default(QMGR)
sslType=NONE #ENUM(CENTRAL|SPECIFIC|NONE),default(NONE)
cloneSupport=false #boolean,default(false)
msgSelection=BROKER #ENUM(CLIENT|BROKER),default(BROKER)
#
# SubSection 1.0.0.2 # ConnectionPool attributes
#
ResourceType=ConnectionPool
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:JMSProvider=WebSphere MQ JMS Provider:MQTopicConnectionFactory=jndiName#tcfJndiName:ConnectionPool=
AttributeInfo=connectionPool
#

#
#Properties
#
stuckThreshold=0 #integer,default(0)
unusedTimeout=1800 #long,default(1800)
maxConnections=10 #integer,default(10)
stuckTimerTime=0 #integer,default(0)
testConnectionInterval=0 #integer,default(0)
minConnections=1 #integer,default(1)
surgeThreshold=-1 #integer,default(-1)
connectionTimeout=180 #long,default(180)
purgePolicy=FailingConnectionOnly #ENUM(EntirePool|FailingConnectionOnly),default(FailingConnectionOnly)
surgeCreationInterval=0 #integer,default(0)
numberOfUnsharedPoolPartitions=0 #integer,default(0)
stuckTime=0 #integer,default(0)
agedTimeout=0 #long,default(0)
reapTime=180 #long,default(180)
testConnection=false #boolean,default(false)
numberOfSharedPoolPartitions=0 #integer,default(0)
freePoolDistributionTableSize=0 #integer,default(0)
numberOfFreePoolPartitions=0 #integer,default(0)


EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create an MQTopicConnectionFactory configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing WebSphere MQ topic connection factory.
  1. Obtain a properties file for the WebSphere MQ topic connection factory that you want to change.

     You can extract a properties file for an MQTopicConnectionFactory using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need a WebSphere MQ topic connection factory, you can delete the entire WebSphere MQ topic connection factory object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the WebSphere MQ topic connection factory.

What to do next

Save the changes to your configuration.

Working with WebSphere MQ queue properties files

You can use properties files to create or change WebSphere MQ queue properties.

Before you begin

Determine the changes that you want to make to your WebSphere MQ queue configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a WebSphere MQ queue object and its configuration properties.

Run administrative commands using wsadmin to extract a properties file for a WebSphere MQ queue, validate the properties, and apply them to your configuration.

Table 486. Actions for WebSphere MQ queue properties files. You can create, modify, and delete WebSphere MQ queue configuration properties.

Action	Procedure
create	Set at least required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.
delete Property	To delete one or more properties, run deleteConfigProperties with only those properties to delete in the properties file.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a WebSphere MQ queue and its properties.
  1. Create a properties file for an MQQueue object.

     Open an editor and create a WebSphere MQ queue properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for an MQQueue object with a name of `myName` and a JNDI name of `myJndiName` under a JMSProvider at the cell scope follows:

```
#
# Header
#
ResourceType=MQQueue
ImplementingResourceType=JMSProvider
ResourceId=Cell=!{cellName}:JMSProvider=WebSphere MQ JMS Provider:MQQueue=jndiName#myJndiName
#

#
#Properties
#
queueManagerPort=0 #integer,default(0)
```

```
password=null
readAhead=YES #ENUM(QUEUE_DEFINED|YES|NO),default(YES)
specifiedExpiry=0 #long,default(0)
queueManagerHost=null
baseQueueName=queueName #required
baseQueueManagerName=null
providerType=null #readonly
decimalEncoding=Normal #ENUM(Normal|Reversed),default(Normal)
serverConnectionChannelName=null
expiry=APPLICATION_DEFINED #ENUM(APPLICATION_DEFINED|SPECIFIED|UNLIMITED),defaul
t(APPLICATION_DEFINED)
sendAsync=YES #ENUM(QUEUE_DEFINED|YES|NO),default(YES)
userName=null
integerEncoding=Normal #ENUM(Normal|Reversed),default(Normal)
specifiedPriority=0 #integer,default(0)
jndiName=myJndiName#required
CCSID=0 #integer,default(0)
category=null
description=null
useNativeEncoding=false #boolean,default(false)
boolean=null
targetClient=JMS #ENUM(JMS|MQ),default(JMS)
priority=APPLICATION_DEFINED #ENUM(APPLICATION_DEFINED|SPECIFIED|QUEUE_DEFINED),
default(APPLICATION_DEFINED)
persistence=APPLICATION_DEFINED #ENUM(APPLICATION_DEFINED|HIGH|NONPERSISTENT|PERSISTENT|QUEUE_DEFINED),default(APPLICATION_DEFINED)
provider=WebSphere MQ JMS Provider #ObjectName(JMSProvider),readonly
floatingPointEncoding=IEEENormal #ENUM(IEEENormal|IEEEReversed|S390),default(IEEENormal)
readAheadClose=DELIVERALL #ENUM(DELIVERCURRENT|DELIVERALL),default(DELIVERALL)
name=myName #required

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create an MQQueue configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing WebSphere MQ queue.
  1. Obtain a properties file for the WebSphere MQ queue that you want to change.

     You can extract a properties file for an MQQueue using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a WebSphere MQ queue configuration.
- Delete the entire WebSphere MQ queue object or its properties.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

Results

You can use the properties file to configure and manage the WebSphere MQ queue object.

What to do next

Save the changes to your configuration.

Working with WebSphere MQ queue connection factory properties files

You can use properties files to create or change WebSphere MQ queue connection factory properties.

Before you begin

Determine the changes that you want to make to your WebSphere MQ queue connection factory configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a WebSphere MQ queue connection factory object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a WebSphere MQ queue connection factory, validate the properties, and apply them to your configuration.

Table 487. Actions for WebSphere MQ queue connection factory properties files. You can create, modify, and delete MQ queue connection factory configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a WebSphere MQ queue connection factory and its properties.
  1. Create a properties file for an MQQueueConnectionFactory object.

     Open an editor and create a WebSphere MQ queue connection factory properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation. The example defines a WebSphere MQ queue connection factory named `myJndiName` under a JMSProvider at cell scope:

```
#
# Header
#
ResourceType=MQQueueConnectionFactory
ImplementingResourceType=JMSProvider
ResourceId=Cell=!{cellName}:JMSProvider=WebSphere MQ JMS Provider:MQQueueConnectionFactory=jndiName#myJndiName
#DELETE=true
#
#Properties
#
wmqServerSvrconnChannel=null
diagnoseConnectionUsage=false #boolean,default(false)
CCSID=null
rcvExit=null
wmqServerName=null
sslConfiguration=null
host=null
category=null
tempModel=null
replyWithRFH2=AS_REPLY_DEST #ENUM(AS_REPLY_DEST|ALWAYS),default(AS_REPLY_DEST)
secExit=null
name=myName #required
xaRecoveryAuthAlias=null
description=null
useConnectionPooling=true #boolean,default(true)
manageCachedHandles=false #boolean,default(false)
wmqServerEndpoint=null
provider=WebSphere MQ JMS Provider #ObjectName(JMSProvider),readonly
ccdtUrl=null
providerVersion=null
sslResetCount=0 #integer,default(0)
secExitInitData=null
pollingInterval=5000 #integer,default(5000)
msgRetention=true #boolean,default(true)
```

```
jndiName= myJndiName #required
queueManager=null
sendExit=null
authMechanismPreference=BASIC_PASSWORD #ENUM(BASIC_PASSWORD|KERBEROS),default(BASIC_PASSWORD)
sendExitInitData=null
maxBatchSize=10 #integer,default(10)
tempQueuePrefix=null
XAEnabled=true #boolean,default(true)
providerType=null #readonly
channel=null
rcvExitInitData=null
port=0 #integer,default(0)
authDataAlias=null
sslPeerName=null
rescanInterval=6000 #integer,default(5000)
compressHeaders=NONE #ENUM(SYSTEM|NONE),default(NONE)
failIfQuiesce=true #boolean,default(true)
clientID=null
localAddress=null
sslCRL=null
logMissingTransactionContext=true #boolean,default(true)
transportType=BINDINGS #ENUM(BINDINGS_THEN_CLIENT|DIRECTHTTP|BINDINGS|DIRECT|CLIENT),default(BINDINGS)
sslCipherSuite=null
compressPayload=NONE #ENUM(ZLIBHIGH|ZLIBFAST|RLE|NONE),default(NONE)
qmgrType=QMGR #ENUM(QSG|QMGR),default(QMGR)
sslType=NONE #ENUM(CENTRAL|SPECIFIC|NONE),default(NONE)


EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create a WebSphere MQ queue connection factory configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing WebSphere MQ queue connection factory.
  1. Obtain a properties file for the WebSphere MQ queue connection factory that you want to change.

     You can extract a properties file for an MQQueueConnectionFactory object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a WebSphere MQ queue connection factory configuration.
- If you no longer need a WebSphere MQ queue connection factory, you can delete the entire WebSphere MQ queue connection factory object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the WebSphere MQ queue connection factory object.

What to do next

Save the changes to your configuration.

Working with mapping module properties files

You can use properties files to create or change mapping module properties.

Before you begin

Determine the changes that you want to make to your mapping module configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a mapping module object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a mapping module, validate the properties, and apply them to your configuration.

Table 488. Actions for mapping module properties files. You can create, modify, and delete mapping module configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a mapping module and its properties.
  1. Create a properties file for a MappingModule object.

     Open an editor and create a mapping module properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation. The example defines a mapping module under a MQQueueConnectionFactory with jndiName `myJName`. The factory is defined under a JMSProvider at cell scope:

```
#
# Header
#
ResourceType=MappingModule
ImplementingResourceType=JMSProvider
ResourceId=Cell=!{cellName}:JMSProvider=WebSphere MQ JMS Provider:MQQueueConnectionFactory=jndiName#myJName:MappingModule=
AttributeInfo=mapping
#DELETE=true
#
#Properties
#
authDataAlias=myADA
mappingConfigAlias=myMCA

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

  2. Run the applyConfigProperties command to create a mapping module configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing mapping module.
  1. Obtain a properties file for the mapping module that you want to change.

You can extract a properties file for a MappingModule object using the extractConfigProperties command.

2. Open the properties file in an editor and change the properties as needed.

   Ensure that the environment variables in the properties file match your system.

3. Run the applyConfigProperties command.

- If you no longer need a mapping module, you can delete the entire mapping module object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the mapping module object.

What to do next

Save the changes to your configuration.

Working with mail provider properties files

You can use properties files to create or change mail provider properties.

Before you begin

Determine the changes that you want to make to your mail provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a mail provider object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a mail provider, validate the properties, and apply them to your configuration.

Table 489. Actions for mail provider properties files. You can create, modify, and delete mail provider configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a mail provider and its properties.

  1. Create a properties file for a MailProvider object.

Open an editor and create a mail provider properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation. The example defines a mail provider named `myProvider` at cell scope:

```
#
# Header
#
ResourceType=MailProvider
ImplementingResourceType=MailProvider
ResourceId=Cell=!{cellName}:MailProvider=myProvider
#DELETE=true
#

#
#Properties
#
classpath={}
name=myProvider
isolatedClassLoader=false #boolean
nativepath={}
description=new mail provider
providerType=null #readonly


EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create a mail provider configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing mail provider.
  1. Obtain a properties file for the mail provider that you want to change.

     You can extract a properties file for a MailProvider object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a mail provider configuration.
- If you no longer need a mail provider, you can delete the entire mail provider object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the mail provider object.

What to do next

Save the changes to your configuration.

Working with mail session properties files

You can use properties files to create or change mail session properties.

Before you begin

Determine the changes that you want to make to your mail session configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a mail session object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a mail session, validate the properties, and apply them to your configuration.

Table 490. Actions for mail session properties files. You can create, modify, and delete mail session configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a mail session and its properties.
  1. Create a properties file for a MailSession object.

     Open an editor and create a mail session properties file. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     The following example defines a mail session with jndiName `myJndiName` under a mail provider named `Built-in Mail Provider` at cell scope:

```
#
# Header
#
ResourceType=MailSession
ImplementingResourceType=MailSession
ResourceId=Cell=!{cellName}:MailProvider=Built-in Mail Provider:MailSession=jndiName#myJndiName
#DELETE=true
#

#
#Properties
#
mailStorePort=0 #integer,default(0)
mailFrom=null
mailTransportPort=0 #integer,default(0)
mailStoreHost=null
providerType=null
mailTransportHost=null
mailStorePassword=null
debug=false #boolean,default(false)
mailStoreUser=null
category=null
description=myJndiName #required
mailTransportUser=null
ObjectName=null
mailStoreProtocol=null
provider=Built-in Mail Provider #ObjectName(MailProvider),readonly
String=null
mailTransportPassword=null
strict=true #boolean,default(true)
name=myName #required


EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
nodeName=WASNode04
```

  2. Run the applyConfigProperties command to create a mail session configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing mail session.
  1. Obtain a properties file for the mail session that you want to change.

     You can extract a properties file for a MailSession object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a mail session configuration.
- If you no longer need a mail session, you can delete the entire mail session object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the mail session object.

What to do next

Save the changes to your configuration.

Working with object pool properties files

You can use properties files to create, modify, or delete object pool properties and custom properties.

Before you begin

Determine the changes that you want to make to your object pool configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an object pool instance. You can also create, modify, or delete object pool custom properties.

Run administrative commands using wsadmin to create or change a properties file for an object pool, validate the properties, and apply them to your configuration.

Table 491. Actions for object pool properties files. You can create, modify, and delete object pool instances.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire ObjectPool instance, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 492. Actions for object pool custom properties. You can create, modify, and delete object pool custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit an object pool properties file.
  1. Set ObjectPool object properties as needed.

     Open an editor on an ObjectPool properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ObjectPool properties file follows:

```
#
# Header
#
ResourceType=ObjectPool
ImplementingResourceType=ObjectPoolManagerInfo
ResourceId=Cell=!{cellName}:ObjectPoolProvider= myObjectPoolProvider:ObjectPoolManagerInfo=
jndiName#myObjPoolJndiName:ObjectPool=poolClassName#abc,poolImplClassName#abc
AttributeInfo=objectPools
#DELETE=true

#
#Properties
#
#
#Properties
#
poolClassName=abc #required
poolImplClassName=abc #required
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to change an object pool configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit object pool custom properties.
  1. Set ObjectPool custom properties as needed.

     Open an editor on an ObjectPool properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=ObjectPool
ImplementingResourceType=ObjectPoolManagerInfo
ResourceId=Cell=!{cellName}:ObjectPoolProvider=myObjectPoolProvider:ObjectPoolManagerInfo=
jndiName#myObjPoolJndiName:ObjectPool=poolClassName#abc,poolImplClassName#abc
AttributeInfo=properties(name,value)
#
```

```
#
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command.

- If you no longer need the object pool or an existing custom property, you can delete the entire object pool object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the object pool instance and its properties.

What to do next

Save the changes to your configuration.

Working with object pool provider properties files

You can use properties files to create, modify, or delete object pool provider properties and custom properties.

Before you begin

Determine the changes that you want to make to your object pool provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an object pool provider instance. You can also create, modify, or delete object pool provider custom properties.

Run administrative commands using wsadmin to create or change a properties file for an object pool provider, validate the properties, and apply them to your configuration.

Table 493. Actions for object pool provider properties files. You can create, modify, and delete object pool provider instances.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.

Table 493. Actions for object pool provider properties files (continued). You can create, modify, and delete object pool provider instances.

Action	Procedure
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire ObjectPoolProvider instance, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit an object pool provider properties file.
  1. Set ObjectPoolProvider properties as needed.

     Open an editor on an ObjectPoolProvider properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example ObjectPoolProvider properties file follows:

```
#
# Header
#
ResourceType=ObjectPoolProvider
ImplementingResourceType=ObjectPoolProvider
ResourceId=Cell=!{cellName}:ObjectPoolProvider=myObjectPoolProvider
#DELETE=true
#

#
#Properties
#
classpath={}
name=myObjectPoolProvider #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Default Object Pool Provider
providerType=null #readonly

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to change an object pool provider configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the object pool provider or an existing custom property, you can delete the entire object pool provider or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the object pool provider instance and its properties.

What to do next

Save the changes to your configuration.

Working with object pool provider J2EE resource properties files:

You can use properties files to create, modify, or delete object pool provider Java 2 Platform, Enterprise Edition (J2EE) resource custom properties.

Before you begin

Determine the changes that you want to make to your object pool provider J2EE resource configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete object pool provider J2EE resource custom properties.

Run administrative commands using wsadmin to change a properties file for an object pool provider J2EE resource, validate the properties, and apply them to your configuration.

Table 494. Actions for object pool provider J2EE resource properties. You can create, modify, and delete object pool provider J2EE resource custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create object pool provider J2EE resource properties.
  1. Specify ObjectPoolProvider J2EEResourcePropertySet custom properties in a properties file.

     Open an editor and specify object pool provider J2EE resource properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=ObjectPoolProvider
ResourceId=Cell=!{cellName}:ObjectPoolProvider=myObjectPoolProvider:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#

#
#Properties
#
existingProp=newValue
```

```
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create an ObjectPoolProvider J2EEResourcePropertySet configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing object pool provider J2EE resource properties.
  1. Obtain a properties file for the object pool provider J2EE resource that you want to change.

     You can extract a properties file for an ObjectPoolProvider J2EEResourcePropertySet using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the object pool provider J2EE resource properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the object pool provider J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with object pool manager properties files

You can use properties files to create, modify, or delete object pool manager information properties and custom properties.

Before you begin

Determine the changes that you want to make to your object pool manager configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an object pool manager instance. You can also create, modify, or delete object pool manager custom properties.

Run administrative commands using wsadmin to create or change a properties file for an object pool manager, validate the properties, and apply them to your configuration.

Table 495. Actions for object pool manager information properties files. You can create, modify, and delete object pool manager information instances.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire ObjectPoolManagerInfo instance, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit an object pool manager properties file.
  1. Set ObjectPoolManagerInfo properties as needed.

     Open an editor on an ObjectPoolManagerInfo properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values. An example ObjectPoolManagerInfo properties file follows:

```
#
# Header
#
ResourceType=ObjectPoolManagerInfo
ImplementingResourceType= ObjectPoolManagerInfo
ResourceId=Cell=!{cellName}:ObjectPoolProvider=myObjectPoolProvider:ObjectPoolManagerInfo=jndiName#myObjPoolJndiName
AttributeInfo=factories
#DELETE=true
#

#
#Properties
#
name=myObjPoolManagerInfo #required
category=null
description=null
providerType=null
jndiName=myObjPoolJndiName #required

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to change an object pool manager configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the object pool manager or an existing custom property, you can delete the entire object pool manager object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the object pool manager instance and its properties.

What to do next

Save the changes to your configuration.

Working with object pool manager J2EE resource properties files:

You can use properties files to create, modify, or delete object pool manager information Java 2 Platform, Enterprise Edition (J2EE) resource custom properties.

Before you begin

Determine the changes that you want to make to your object pool manager information J2EE resource configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete object pool manager information J2EE resource custom properties.

Run administrative commands using wsadmin to change a properties file for an object pool manager information J2EE resource, validate the properties, and apply them to your configuration.

Table 496. Actions for object pool manager information J2EE resource properties. You can create, modify, and delete object pool manager information J2EE resource custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create object pool manager J2EE resource properties.
  1. Specify ObjectPoolManagerInfo J2EEResourcePropertySet custom properties in a properties file.

     Open an editor and specify object pool manager J2EE resource properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
```

```
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=ObjectPoolManagerInfo
ResourceId=Cell=!{cellName}:ObjectPoolProvider=myObjectPoolProvider:ObjectPoolManagerInfo=jndiName#myObjPoolJndiName:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create an ObjectPoolManagerInfo J2EEResourcePropertySet configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing object pool manager J2EE resource properties.
  1. Obtain a properties file for the object pool manager J2EE resource that you want to change.

     You can extract a properties file for an ObjectPoolManagerInfo J2EEResourcePropertySet using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the object pool manager J2EE resource properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure the object pool manager information J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with scheduler provider properties files

You can use properties files to create, modify, or delete scheduler provider properties and custom properties.

Before you begin

Determine the changes that you want to make to your scheduler provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a scheduler provider object. You can also create, modify, or delete scheduler provider custom properties.

Run administrative commands using wsadmin to create or change a properties file for a scheduler provider, validate the properties, and apply them to your configuration.

Table 497. Actions for scheduler provider properties files. You can create, modify, and delete scheduler provider properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire SchedulerProvider object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create a scheduler provider properties file.
   a. Set SchedulerProvider properties as needed.

      Open an editor on a SchedulerProvider properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example SchedulerProvider properties file follows:

```
#
# Header
#
ResourceType=SchedulerProvider
ImplementingResourceType=SchedulerProvider
ResourceId=Cell=!{cellName}:SchedulerProvider=mySchedulerProvider
#DELETE=true
#

#
#Properties
#
classpath={}
name=mySchedulerProvider #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Default Scheduler Provider
providerType=null #readonly
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

   b. Run the applyConfigProperties command to create or change a scheduler provider configuration.

      Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. Modify an existing properties file.
   a. Obtain a properties file for the scheduler provider that you want to change.

      You can extract a properties file for a SchedulerProvider object using the extractConfigProperties command.

   b. Open the properties file in an editor and change the properties as needed.

      Ensure that the environment variables in the properties file match your system.

   c. Run the applyConfigProperties command.

3. If you no longer need the scheduler provider or an existing custom property, you can delete the entire scheduler provider object or the custom property.
   - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

   - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the scheduler provider object and its properties.

What to do next

Save the changes to your configuration.

Working with scheduler configuration properties files

You can use properties files to create, modify, or delete scheduler configuration properties and custom properties.

Before you begin

Determine the changes that you want to make to your scheduler configuration or its objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a scheduler configuration object. You can also create, modify, or delete scheduler configuration custom properties.

Run administrative commands using wsadmin to create or change a properties file for a scheduler configuration, validate the properties, and apply them to your configuration.

Table 498. Actions for scheduler configuration properties files. You can create, modify, and delete scheduler configuration properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire SchedulerConfiguration object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create a scheduler configuration properties file.
   a. Set SchedulerConfiguration properties as needed.

      Open an editor on a SchedulerConfiguration properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example SchedulerConfiguration properties file follows:

```
#
# Header
#
ResourceType=SchedulerConfiguration
ImplementingResourceType=SchedulerConfiguration
ResourceId=Cell=!{cellName}:SchedulerProvider=mySchedulerProvider:SchedulerConfiguration=jndiName#mySchedulerJndiName
#DELETE=true
#

#
#Properties
#
securityRole=null
tablePrefix=null #required
referenceable=null
name=myScheduler #required
pollInterval=30 #integer,required,default(30)
category=null
datasourceJNDIName=null #required
workManagerInfo=null
loginConfigName=null
providerType=null
workManagerInfoJNDIName=null
useAdminRoles=false #boolean,default(false)
jndiName=mySchedulerJndiName #required
datasourceAlias=null
#provider=SchedulerProvider#ObjectName(SchedulerProvider),readonly
description=null
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

   b. Run the applyConfigProperties command to create or change a scheduler configuration.

      Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. Modify an existing properties file.
   a. Obtain a properties file for the scheduler configuration that you want to change.

      You can extract a properties file for a SchedulerConfiguration object using the extractConfigProperties command.

   b. Open the properties file in an editor and change the properties as needed.

      Ensure that the environment variables in the properties file match your system.

   c. Run the applyConfigProperties command.
3. If you no longer need the scheduler configuration or an existing custom property, you can delete the entire scheduler configuration object or the custom property.
   - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

   - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the scheduler configuration object and its properties.

What to do next

Save the changes to your configuration.

Working with scheduler configuration J2EE resource properties files:

You can use properties files to create, modify, or delete scheduler configuration Java 2 Platform, Enterprise Edition (J2EE) resource custom properties.

Before you begin

Determine the changes that you want to make to your scheduler configuration J2EE resource configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete scheduler configuration J2EE resource custom properties.

Run administrative commands using wsadmin to change a properties file for a scheduler configuration J2EE resource, validate the properties, and apply them to your configuration.

Table 499. Actions for scheduler configuration J2EE resource properties. You can create, modify, and delete scheduler configuration J2EE resource custom properties.

Action	Procedure
create	Not applicable
modify	Run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Run the applyConfigProperties command to create a custom property.
delete Property	Run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create scheduler configuration J2EE resource properties.
  1. Specify SchedulerConfiguration J2EEResourcePropertySet custom properties in a properties file.

     Open an editor and specify scheduler configuration J2EE resource properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType= SchedulerConfiguration
ResourceId=Cell=!{cellName}:SchedulerProvider=mySchedulerProvider:SchedulerConfiguration=jndiName#mySchedulerJndiName:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue
```

```
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create or change a scheduler configuration J2EE resource configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing scheduler configuration J2EE resource properties.
  1. Obtain a properties file for the scheduler configuration J2EE resource that you want to change.

     You can extract a properties file for a SchedulerConfiguration J2EEResourcePropertySet using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need a scheduler configuration J2EE resource custom property, you can delete the custom property.

  To delete one or more custom properties, specify only the properties to delete in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the scheduler configuration J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with security properties files

You can use properties files to modify or delete security properties.

Before you begin

Determine the changes that you want to make to your security configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a security object.

Run administrative commands using wsadmin to change a properties file for a security object, validate the properties, and apply them to your configuration.

Table 500. Actions for security properties. You can modify or delete security properties.

Action	Procedure
create	Not applicable

Table 500. Actions for security properties (continued). You can modify or delete security properties.

Action	Procedure
modify	Edit property values in the security properties file and then run the applyConfigProperties command.
delete	Run the deleteConfigProperties command to delete one or more properties. If a deleted property has a default value, the property is set to the default value. Otherwise, the deleted property is removed.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify an existing properties file.
  1. Obtain a properties file for the Security object that you want to change.

     You can extract a properties file for a Security object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system. An example Security properties file follows:

```
#
# Header
#
ResourceType=Security
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=
#

#
#Properties
#
useLocalSecurityServer=true #boolean,default(false)
cacheTimeout=600 #integer,required,default(0)
allowBasicAuth=true #boolean,default(false)
enforceJava2Security=false #boolean,default(false)
activeAuthMechanism=Cell=!{cellName}:Security=:LTPA= #ObjectName(LTPA)
enabled=true #boolean,default(false)
adminPreferredAuthMech=null
enableJava2SecRuntimeFiltering=false #boolean,default(false)
allowAllPermissionForApplication=false #boolean,default(false)
useDomainQualifiedUserNames=false #boolean,default(false)
internalServerId=null
activeUserRegistry= Cell=!{cellName}:Security=:LDAPUserRegistry=type#IBM_DIRECTORY_SERVER #ObjectName(LDAPUserRegistry)
defaultSSLSettings=Cell=!{cellName}:Security=:SSLConfig=alias#CellDefaultSSLSett
ings,managementScope#"Cell=!{cellName}:Security=:ManagementScope=scopeName#"(cell):!{cellName}"" #ObjectName(SSLConfig)
enforceFineGrainedJCASecurity=false #boolean,default(false)
dynamicallyUpdateSSLConfig=true #boolean,default(false)
activeProtocol=BOTH #ENUM(CSI|IBM|BOTH),required,default(IBM)
issuePermissionWarning=true #boolean,default(false)
appEnabled=false #boolean,default(false)

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

  3. Run the applyConfigProperties command to create or change a security object.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need a property, you can delete the security property.

  To delete one or more properties, specify only the properties to be deleted in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the security properties.

What to do next

Save the changes to your configuration.

Working with LDAP properties files

You can use properties files to create, modify, or delete Lightweight Directory Access Protocol (LDAP) user registry properties.

Before you begin

Determine the changes that you want to make to your LDAP configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a LDAP user registry object.

Run administrative commands using wsadmin to create or change a properties file for a LDAP user registry, validate the properties, and apply them to your configuration.

Table 501. Actions for LDAP user registry properties files. You can create, modify, and delete LDAP properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command..
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire LDAPUserRegistry object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create a properties file for a LDAPUserRegistry object.
   a. Set LDAPUserRegistry properties as needed.

      Open an editor on an LDAPUserRegistry properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed.

      An example LDAPUserRegistry properties file follows. The example file creates an IBM_DIRECTORY_SERVER type LDAP registry. The properties differ for different types of LDAP registry. The LDAP registry type is used as a key to identify various configured LDAP registries. Ensure that there is only one LDAP registry configuration for each type of LDAP registry.

```
#
# Header
#
ResourceType=LDAPUserRegistry
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:LDAPUserRegistry=type#IBM_DIRECTORY_SERVER
#DELETE=true
```

```
#

#
#Properties
#
useRegistryRealm=false #boolean,default(false)
serverPassword="{xor}"
sslConfig=
primaryAdminId=
useRegistryServerId=false #boolean,default(false)
limit=0 #integer,default(0)
searchTimeout=120 #long,default(0)
bindPassword=
serverId=
realm=
baseDN=
ignoreCase=true #boolean,default(false)
type=IBM_DIRECTORY_SERVER #ENUM(NETSCAPE|DOMINO502|CUSTOM|ACTIVE_DIRECTORY|NDS|IBM_DIRECTORY_SERVER|
IPLANET|SECUREWAY),default(IBM_DIRECTORY_SERVER)
reuseConnection=true #boolean,default(false)
sslEnabled=false #boolean,default(false)
monitorInterval=0 #long,default(0)
bindDN=


#
# Header LDAPSearchFilter Section
#
ResourceType=LDAPSearchFilter
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:LDAPUserRegistry=type#IBM_DIRECTORY_SERVER:LDAPSearchFilter=
AttributeInfo=searchFilter
#

#
#Properties
#
krbUserFilter="(&(krbPrincipalName=%v)(objectclass=ePerson))"
groupMemberIdMap="ibm-allGroups:member;ibm-allGroups:uniqueMember"
certificateFilter=
userIdMap="*:uid"
userFilter="(&(uid=%v)(objectclass=ePerson))"
groupIdMap="*:cn"
groupFilter="(&(cn=%v)(|(objectclass=groupOfNames)(objectclass=groupOfUniqueNames)))"
certificateMapMode=EXACT_DN #ENUM(CERTIFICATE_FILTER|EXACT_DN),default(EXACT_DN)

#
# Header EndPoint
#
ResourceType=EndPoint
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:LDAPUserRegistry=type#IBM_DIRECTORY_SERVER:EndPoint=
AttributeInfo=hosts
#

#
#Properties
#
port=389 #integer,required,default(0)
host= #required


EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

b. Run the applyConfigProperties command to create or change a LDAP user registry configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. Modify an existing properties file.
   a. Obtain a properties file for the LDAPUserRegistry object that you want to change.

      You can extract a properties file for a LDAPUserRegistry object using the extractConfigProperties command.

   b. Open the properties file in an editor and change the properties as needed.

      Ensure that the environment variables in the properties file match your system.

   c. Run the applyConfigProperties command.
3. If you no longer need the LDAP user registry object or an existing property, you can delete the entire LDAP object or one or more properties.

- To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

- To delete one or more properties, specify only the properties to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the LDAP object and its properties.

What to do next

Save the changes to your configuration.

Working with LTPA properties files

You can use properties files to modify or delete Lightweight Third Party Authentication (LTPA) properties.

Before you begin

Determine the changes that you want to make to your LTPA configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a LTPA object.

Run administrative commands using wsadmin to change a properties file for a LTPA object, validate the properties, and apply them to your configuration.

Table 502. Actions for LTPA properties. You can modify or delete LTPA properties.

Action	Procedure
create	Not applicable
modify	Edit property values in the LTPA properties file and then run the applyConfigProperties command.
delete	Run the deleteConfigProperties command to delete one or more properties. If a deleted property has a default value, the property is set to the default value. Otherwise, the deleted property is removed.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify an existing properties file.
  1. Obtain a properties file for the LTPA object that you want to change.
     You can extract a properties file for an LTPA object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

Ensure that the environment variables in the properties file match your system. To specify a custom property, edit the `AttributeInfo` value and properties values. An example LTPA properties file follows:

```
#
# Header
#
ResourceType=LTPA
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:LTPA=
#

#
#Properties
#
simpleAuthConfig=system.LTPA
keySetGroup=CellLTPAKeySetGroup #ObjectName(KeySetGroup)
authContextImplClass=com.ibm.ISecurityLocalObjectTokenBaseImpl.WSSecurityContextLTPAImpl
authConfig=system.LTPA
isCredentialForwardable=false #boolean,default(false)
timeout=120 #long,required,default(0)
OID="oid:1.3.18.0.2.30.2"
password=null #required
authValidationConfig=system.LTPA


#
# Header SingleSignon Section
#
ResourceType=SingleSignon
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:LTPA=:SingleSignon=
AttributeInfo=singleSignon
#

#
#Properties
#
enabled=true #boolean,default(false)
domainName=
requiresSSL=false #boolean,default(false)


EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

3. Run the applyConfigProperties command to create or change an LTPA object.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need a property, you can delete the LTPA property.

  To delete one or more properties, specify only the properties to be deleted in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the LTPA properties.

What to do next

Save the changes to your configuration.

Working with JAAS configuration entry properties files

You can use properties files to create, modify, or delete Java Authentication and Authorization Service (JAAS) configuration entry properties.

Before you begin

Determine the changes that you want to make to your JAAS configuration entry object or its properties.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a JAAS configuration entry object.

Run administrative commands using wsadmin to create or change a properties file for a JAAS configuration entry, validate the properties, and apply them to your configuration.

Table 503. Actions for JAAS configuration entry properties files. You can create, modify, and delete JAAS properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command..
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire JAASConfigurationEntry object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create a JAASConfigurationEntry properties file.
   a. Set JAASConfigurationEntry properties as needed.

      You can add a new JAAS configuration entry under either `systemLoginConfiguration` or `applicationLoginConfiguration`.

      Open an editor and create a properties file for a JAASConfigurationEntry object. The following example uses `systemLoginConfiguration` to add a new JAAS configuration entry:

```
#
# Header
#
ResourceType=JAASConfigurationEntry
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:JAASConfiguration=systemLoginConfig#:JAASConfigurationEntry=
alias#myJAAS
#DELETE=true
#

#
#Properties
#
alias=myJAAS #required


#
# Header JAASLoginModule
#
ResourceType=JAASLoginModule
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:JAASConfiguration=systemLoginConfig#:JAASConfigurationEntry=
alias#myJAAS:JAASLoginModule=moduleClassName#com.acme.myLoginModule
AttributeInfo=loginModules
#DELETE=true
#

#
#Properties
#
callbackHandlerClassName=null
moduleClassName=com.acme.myLoginModule #required
authenticationStrategy=REQUIRED #ENUM(OPTIONAL|REQUISITE|REQUIRED|SUFFICIENT),de
fault(REQUIRED)
```

```
#
# Header JAASLoginModule options
#
ResourceType=JAASLoginModule
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:JAASConfiguration=systemLoginConfig#:JAASConfigurationEntry=
alias#myJAAS:JAASLoginModule=moduleClassName#com.acme.myLoginModule
AttributeInfo=options(name,value)
#

#
#Properties
#
myProp=myValue

#
# Header JAASLoginModule Another module
#
ResourceType=JAASLoginModule
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:JAASConfiguration=systemLoginConfig#:JAASConfigurationEntry=
alias#myJAAS:JAASLoginModule=moduleClassName#com.acme.myAnotherLoginModule
AttributeInfo=loginModules
#DELETE=true
#

#
#Properties
#
callbackHandlerClassName=null
moduleClassName=com.acme.myAnotherLoginModule #required
authenticationStrategy=REQUIRED #ENUM(OPTIONAL|REQUISITE|REQUIRED|SUFFICIENT),de
fault(REQUIRED)


#
# Header JAASLoginModule options
#
ResourceType=JAASLoginModule
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:JAASConfiguration=systemLoginConfig#:JAASConfigurationEntry=
alias#myJAAS:JAASLoginModule=moduleClassName#com.acme.myAnotherLoginModule
AttributeInfo=options(name,value)
#

#
#Properties
#
myProp=myValue

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

   b. Run the applyConfigProperties command to create or change a JAAS configuration entry.

      Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

2. Modify an existing properties file.
   a. Obtain a properties file for the JAASConfigurationEntry that you want to change.

      You can extract a properties file for a JAASConfigurationEntry object using the extractConfigProperties command.

   b. Open the properties file in an editor and change the properties as needed.

      Ensure that the environment variables in the properties file match your system.

   c. Run the applyConfigProperties command.
3. If you no longer need the JAAS configuration entry object or an existing property, you can delete the entire JAAS object or one or more properties.
   - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

   - To delete one or more properties, specify only the properties to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the JAAS configuration entry object and its properties.

What to do next

Save the changes to your configuration.

Working with JAAS authorization data properties files

You can use properties files to create, modify, or delete Java Authentication and Authorization Service (JAAS) authorization data properties.

Before you begin

Determine the changes that you want to make to your JAAS authorization data object or its properties.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a JAAS authorization data object.

Run administrative commands using wsadmin to create or change a properties file for a JAAS authorization data, validate the properties, and apply them to your configuration.

Table 504. Actions for JAAS authorization data properties files. You can create, modify, and delete JAAS properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit any properties and then run the applyConfigProperties command..
delete	To delete the entire JAASAuthData object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create a properties file for a JAASAuthData object.
  1. Set JAASAuthData properties as needed.

     Open an editor on a JAASAuthData properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example JAASAuthData properties file follows:

```
#
# Header
#
ResourceType=JAASAuthData
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:JAASAuthData=alias#myAlias
AttributeInfo=authDataEntries
#DELETE=true
#

#
#Properties
```

```
#
password=myPassword #required
userId=cp_web #required
alias=myAlias #required
description=my new alias

EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

2. Run the applyConfigProperties command to create or change a JAAS authorization data object.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the JAASAuthData object that you want to change.

     You can extract a properties file for a JAASAuthData object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the JAAS authorization data object, you can delete the entire JAAS object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the JAAS authorization data object and its properties.

What to do next

Save the changes to your configuration.

Working with SSL configuration properties files

You can use properties files to create, modify, or delete Secure Sockets Layer (SSL) configuration properties.

Before you begin

Determine the changes that you want to make to your SSL configuration object or its properties.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a SSL configuration object.

Run administrative commands using wsadmin to create or change a properties file for a SSL configuration, validate the properties, and apply them.

Table 505. Actions for SSL configuration properties files. You can create, modify, and delete SSL configuration properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.

Table 505. Actions for SSL configuration properties files (continued). You can create, modify, and delete SSL configuration properties.

Action	Procedure
modify	Edit any properties and then run the applyConfigProperties command..
delete	To delete the entire SSLConfig object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for an SSL configuration object.
  1. Set SSLConfig properties as needed.

     Open an editor on an SSLConfig properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example SSLConfig properties file follows:

```
#
# Header
#
ResourceType=SSLConfig
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:SSLConfig=alias#CellDefaultSSLSettings,managementScope#
"Cell=!{cellName}:Security=:ManagementScope=scopeName#"(cell):!{cellName}""
#DELETE=true
#

#
#Properties
#
managementScope=Cell=!{cellName}:Security=:ManagementScope=scopeName#"(cell):!{cellName}"
#ObjectName(ManagementScope)
alias=CellDefaultSSLSettings #required
type=JSSE #ENUM(SSSL|JSSE),default(JSSE)

#
EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

  2. Run the applyConfigProperties command to create or change an SSL configuration object.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the SSL configuration object that you want to change.

     You can extract a properties file for an SSLConfig object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the SSL configuration object, you can delete the entire SSL object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the SSL configuration object and its properties.

What to do next

Save the changes to your configuration.

Working with secure socket layer properties files:

You can use properties files to create, modify, or delete secure socket layer properties.

Before you begin

Determine the changes that you want to make to your secure socket layer object or its properties.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a secure socket layer object.

Run administrative commands using wsadmin to create or change a properties file for a secure socket layer, validate the properties, and apply them.

Table 506. Actions for secure socket layer properties files. You can create, modify, and delete secure socket layer properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit any properties and then run the applyConfigProperties command..
delete	To delete the entire SecureSocketLayer object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a properties file for a secure socket layer.
  1. Set SecureSocketLayer properties as needed.

     Open an editor on a SecureSocketLayer properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the AttributeInfo value and properties. An example SecureSocketLayer properties file follows:

```
#
# Header
#
ResourceType=SecureSocketLayer
ImplementingResourceType=Security
ResourceId=Cell=!{cellName}:Security=:SSLConfig=alias#CellDefaultSSLSettings,managementScope#
"Cell=!{cellName}:Security=:ManagementScope=scopeName#"(cell):!{cellName}"":SecureSocketLayer=
AttributeInfo=setting
#

#
```

```
#Properties
#
keyFileName=null
enableCryptoHardwareSupport=false #boolean,default(false)
serverKeyAlias=null
sslProtocol=SSL_TLS
clientAuthentication=false #boolean,default(false)
securityLevel=HIGH #ENUM(MEDIUM|HIGH|CUSTOM|LOW),default(HIGH)
keyFileFormat=JKS #ENUM(JCEK|JKS|JCERACFKS|JCE4758RACFKS|PKCS12),default(JKS)
CryptoHardwareToken"=null
keyStore=CellDefaultKeyStore #ObjectName(KeyStore)
enabledCiphers=
keyManager=IbmX509 #ObjectName(KeyManager)
trustFileFormat=JKS #ENUM(JCEK|JKS|JCERACFKS|JCE4758RACFKS|PKCS12),default(JKS)
clientAuthenticationSupported=false #boolean,default(false)
trustStore=CellDefaultTrustStore #ObjectName(KeyStore)
keyFilePassword=null
jsseProvider=IBMJSSE2
clientKeyAlias=null
trustFileName=null
trustFilePassword=null
trustManager={IbmPKIX} #ObjectName*(TrustManager)

#
EnvironmentVariablesSection
#Environment Variables
cellName=myCell
```

2. Run the applyConfigProperties command to create or change a secure socket layer object.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the SecureSocketLayer object that you want to change.

     You can extract a properties file for a SecureSocketLayer object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the secure socket layer object, you can delete the entire SSL object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the secure socket layer object and its properties.

What to do next

Save the changes to your configuration.

Retrieving signer certificates using SSL properties files

You can use properties files to retrieve Secure Sockets Layer (SSL) signer certificates.

Before you begin

Determine whether you want to change your SSL configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can set SSL properties and retrieve SSL signer certificates from a port.

Run the applyConfigProperties command using wsadmin to apply SSL properties and run the retrieveSignerFromPort command.

Table 507. Actions for SSL properties. You can run the retrieveSignerFromPort command .

Action	Procedure
create	Not applicable
modify	Not applicable
delete	Not applicable
create Property	Not applicable
delete Property	Not applicable
retrieve signer	1. Create a properties file that specifies `CreateDeleteCommandProperties=true`, `commandName=retrieveSignerFromPort`, and SSL property values such as port number, certificate alias, and key store name. 2. Run the applyConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Create a properties file that specifies the retrieveSignerFromPort command and SSL property values.

   The following properties file specifies `CreateDeleteCommandProperties=true`, `commandName=retrieveSignerFromPort`, and SSL property values such as port number, certificate alias, and key store name:

```
#
# Header
#
CreateDeleteCommandProperties=true
#SKIP=true
commandName=retrieveSignerFromPort
#

#
#Properties
#
port=1234 #Integer,required
keyStoreScope=null #String
sslConfigName=null #String
host=myHost #String,required
certificateAlias=certificateAlias #String,required
keyStoreName=CellDefaultTrustStore #String,required
sslConfigScopeName=null
#
```

   If needed, modify the environment section to match your system and make any required changes to properties.

2. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to retrieve signer certificates and manage the SSL properties.

What to do next

If you changed SSL properties, save the changes to your configuration.

Enabling global security and configuring federated user registries using properties files

You can use properties files to enable global security and configure federated Lightweight Directory Access Protocol (LDAP) user registries. This topic provides an example properties file that you can modify for your environment.

Before you begin

Determine whether you want to use Secure Sockets Layer (SSL) to access a LDAP server. If you use SSL to access a LDAP server, you must extract the LDAP server signer certificate, store it in the default trust store, and then restart the server. The example properties file completes these steps.

About this task

The example properties file in this topic assumes that the administrative user already exists in the LDAP user registry.

Further, the example specifies several commands:

- retrieveSignerFromPort to retrieve an SSL signer certificate from a port
- applyWizardSettings
- createIdMgrLDAPRepository
- addIdMgrLDAPServer
- addIdMgrRepositoryBaseEntry
- updateIdMgrSupportedEntityType
- addIdMgrRealmBaseEntry
- deleteIdMgrRealmBaseEntry
-

Procedure

1. Start the wsadmin scripting tool.

   To start wsadmin using the Jython language, run the following command from the `bin` directory of the server profile:

   ```
   wsadmin -lang Jython
   ```

2. Create a properties file that retrieves an SSL signer certificate from a port, enables global security, and configures a federated LDAP user registry.

```
#
# Extract LDAP server signer certificate and store it in default trust store.
# Save configuration and restart server after retrieving signer certificate.
#
CreateDeleteCommandProperties=true
SKIP=true
commandName=retrieveSignerFromPort
#

#
#Properties
#
port=636 #Integer,required
keyStoreScope=(cell):!{cellName}:(node):!{nodeName} #String
sslConfigName=NodeDefaultSSLSettings #String
host=!{ldapHostName} #String,required
certificateAlias=ldapalias #String,required
keyStoreName=NodeDefaultTrustStore #String,required
sslConfigScopeName=(cell):!{cellName}:(node):!{nodeName} #String
#

#
# Enable global security with adminuser and adminpasswd
#

#
CreateDeleteCommandProperties=true
commandName=applyWizardSettings
#

#
# Properties
```

```
#
adminPassword=!{adminPasswd} #String
userRegistryType=WIMUserRegistry #String,required
secureApps=true #Boolean,required
ldapServerType=null #String
customProps=null #String
adminName=!{adminUser} #String,required
ldapPort=null #String
secureLocalResources=false #Boolean,required
ldapBindPassword=null #String
ldapBaseDN=null #String
customRegistryClass=null #String
ignoreCase=null #Boolean
ldapHostName=null #String
ldapBindDN=null #String
#

#
# create IdMgr for LDAP
#

#
CreateDeleteCommandProperties=true
commandName=createIdMgrLDAPRepository
#

#
#Properties
#
certificateFilter=null #String
searchTimeLimit=null #Integer
translateRDN=null #Boolean
supportSorting=null #Boolean
ldapServerType=IDS #String,required
supportTransactions=null #Boolean
supportAsyncMode=null #Boolean
primaryServerQueryTimeInterval=null #Integer
adapterClassName=null #String
supportExternalName=null #Boolean
isExtIdUnique=null #Boolean
sslConfiguration=NodeDefaultSSLSettings #String
searchCountLimit=null #Integer
id=!{ldapRegId} #String,required
searchPageSize=null #Integer
loginProperties=uid #String
supportPaging=null #Boolean
default=true #Boolean
returnToPrimaryServer=null #Boolean
certificateMapMode=exactdn #String
#

#
# add IdMgr to ldap server
#

CreateDeleteCommandProperties=true
commandName=addIdMgrLDAPServer
#

#
# Properties
#
sslConfiguration=NodeDefaultSSLSettings #String
id=!{ldapRegId} #String,required
port=636 #Integer
derefAliases=null #String
ldapServerType=IDS #String
bindPassword=!{bindPasswd} #String
certificateFilter=null #String
authentication=simple #String
sslEnabled=true #Boolean
connectTimeout=null #Integer
referal=ignore #String
host=!{ldapHostName} #String,required
bindDN=cn=root #String
certificateMapMode=exactdn #String
connectionPool=null #Boolean
#

#
# configure other LDAP attrs
#

#
CreateDeleteCommandProperties=true
commandName=addIdMgrRepositoryBaseEntry
#

#
# Properties
```

```
#
name=c=us #String,required
nameInRepository=c=us #String
id=!{ldapRegId} #String,required
#

#
CreateDeleteCommandProperties=true
commandName=updateIdMgrSupportedEntityType
#

#
# Properties
#
defaultParent=c=us #String
name=Group #String,required
rdnProperties=cn #String
#

#
CreateDeleteCommandProperties=true
commandName=updateIdMgrSupportedEntityType
#

#
# Properties
#
defaultParent=c=us #String
name=OrgContainer #String,required
rdnProperties=o;ou;dc;cn #String
#

#
CreateDeleteCommandProperties=true
commandName=updateIdMgrSupportedEntityType
#

#
#Properties
#
defaultParent=c=us #String
name=PersonAccount #String,required
rdnProperties=uid
#

#
# add this IdMgr as base entry to default realm
#

#
CreateDeleteCommandProperties=true
commandName=addIdMgrRealmBaseEntry
#

#
# Properties
#
name=defaultWIMFileBasedRealm #String,required
baseEntry=c=us #String,required
#

#
# delete old WIM IdMgr as base entry from default realm.
#

#
CreateDeleteCommandProperties=true
commandName=deleteIdMgrRealmBaseEntry
#

#
# Properties
#
name=defaultWIMFileBasedRealm #String,required
baseEntry=o=defaultWIMFileBasedRealm #String,required
#


EnvironmentVariablesSection
#
# Environment Variables
#
cellName=myCell04
nodeName=myNode03
ldapHostName=myLdapHost
adminUser=myAdminId
adminPasswd=myAdminPasswd
ldapRegId=ldapRegId
bindPasswd=myBindPw
```

3. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName mySecurityConfig.props -reportFileName report.txt '])
```

Results

The properties file uses administrative command to enable global security and configure a federated LDAP user registry.

What to do next

If you want to apply this properties file, modify the for your environment.

Mapping users and resources using authorization group properties files

You can use authorization group properties files to map users to administrative roles and resources to authorization groups.

Before you begin

Determine the property values that you want to set for an authorization group configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create or modify an authorization group object and map users to administrative roles or resources to groups.

This topic provides sample properties files that you can modify for your environment and apply:

- Create an authorization group.
- Map users to administrative roles.
- Map resources to administrative groups.

Procedure

- Create an authorization group.
  1. Create a properties file that uses the createAuthorizationGroup command and names the group.

     The following example creates an authorization group named `ag1`:

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=createAuthorizationGroup
#

#
# Properties
#
authorizationGroupName=ag1 #String,required
```

  2. Run the applyConfigProperties command.

     Running the applyConfigProperties command applies the properties file. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Map users to administrative roles.
  1. Modify an AuthorizationGroup properties file so that it lists users for administrative roles.

To add a new user to a role, add the user to the role list. For example, to add user5 to the administrators role list, change `administrators={}` to `administrators={user:user5,group:group1}`.

To remove a user from a role, remove the user from the role list; for example, `adminsecuritymanagers={user:user4}`. To remove all users for a role, make the list empty.

```
#
# Header
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=CellAuthorizationGroup
#

#
# Properties
#
deployers={}
name=CellAuthorizationGroup
resources={}
configurators={}
monitors={}
operators={}
adminsecuritymanagers={user:user4}
auditors={special:SERVERID,special:PRIMARYADMINID}
administrators={user:user5,group:group1}
```

2. Run the applyConfigProperties command.

- Map resources to administrative groups.
  1. Modify an AuthorizationGroup properties file so that it lists resources.

     To add a new resource to an authorization group, add the resource to the resources list. To remove a resource from an authorization group, remove the resource from the list.

     The following example maps users to administrative roles of an authorization group and maps resources to an authorization group. An authorization group is used to enable fine-grained administrative security.

```
#
# Header
#
ResourceType=AuthorizationGroup
ImplementingResourceType=AuthorizationGroup
ResourceId=AuthorizationGroup=ag1
#

#
# Properties
#
deployers={}
name=ag1
resources={Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName},Cell=!{cellName}:Deployment=myApp}
configurators={}
monitors={}
operators={}
adminsecuritymanagers={}
auditors={}
administrators={user:user5,group:group1}
#


EnvironmentVariablesSection
#
# Environment Variables
cellName=myCell
nodeName=myNode
serverName=myServer
```

  2. Run the applyConfigProperties command.

Results

You can use the properties file to configure and manage authorization groups.

What to do next

Save the changes to your configuration.

Working with server properties files

You can use properties files to change server properties.

Before you begin

Determine the changes that you want to make to your server configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a server object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a server, validate the properties, and apply them to your configuration.

Table 508. Actions for server properties files. You can create, modify, and delete server properties.

Action	Procedure
create	1. Run the createPropertiesFileTemplates command to create a server template. 2. Edit the template `create` sections and set `SKIP=false`. 3. Run the applyConfigProperties command.
modify	Follow create and delete steps.
delete	1. Run the createPropertiesFileTemplates command to create a server template. 2. Edit the template `delete` sections and set `SKIP=false`. 3. Run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Run the createPropertiesFileTemplates command to create a server template.
2. Open the properties file in an editor.
3. To create server properties, edit the template `create` sections, set `SKIP=false`, and run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

4. To delete server properties, edit the template `delete` sections, set `SKIP=false`, and run the deleteConfigProperties command.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

What to do next

Save the changes to your configuration.

Working with application server properties files

You can use properties files to create or change application server properties and the associated StateManageable instance under a server.

Before you begin

Determine the changes that you want to make to your application server configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an application server properties.

Run administrative commands using wsadmin to create or change a properties file for an application server, validate the properties, and apply them to your configuration.

Table 509. Actions for application server properties files. You can create, modify, and delete application server properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify a properties file for an ApplicationServer object.
  1. Obtain a properties file for the application server that you want to change.

     You can extract a properties file for an ApplicationServer using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     For example, modify the ApplicationServer and associated StateManageable under a server. The following properties file shows a property under ApplicationServer with name `myName` and value `myVal`:

```
#
# Header
#
ResourceType=ApplicationServer
ImplementingResourceType=ApplicationServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
AttributeInfo=components
#

#
#Properties
#
applicationClassLoaderPolicy=MULTIPLE #ENUM(MULTIPLE|SINGLE),default(MULTIPLE)
name=null
applicationClassLoadingMode=PARENT_FIRST #ENUM(PARENT_FIRST|PARENT_LAST),default(PARENT_FIRST)
server=!{serverName}
parentComponent=null
id=-1 #long,default(-1)


#
# Header
#
ResourceType=StateManageable
ImplementingResourceType=ApplicationServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:StateManageable=
AttributeInfo=stateManagement
#

#
#Properties
#
initialState=START #ENUM(STOP|START),default(START)
managedObject=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer= #ObjectName(ApplicationServer),readonly
```

```
#
# Header
#
ResourceType=ApplicationServer
ImplementingResourceType=ApplicationServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
AttributeInfo=properties(name,value)
#

#
#Properties
#
myName=myVal
#

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

3. Run the applyConfigProperties command to change an application server configuration and create any new properties.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Delete application server properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the application server object.

What to do next

Save the changes to your configuration.

Working with class loader properties files:

You can use properties files to create or change PARENT_LAST class loader properties under the ApplicationServer object of a server.

Before you begin

Determine the changes that you want to make to your class loader configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a class loader object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a class loader, validate the properties, and apply them to your configuration.

Table 510. Actions for PARENT_LAST class loader properties files. You can create, modify, and delete class loader configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a class loader and its properties.
  1. Create a properties file for a Classloader object.

     Open an editor and create a class loader properties file. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     The following example defines a class loader with mode `PARENT_LAST` under an ApplicationServer object of a server:

```
#
# SubSection 1.0.0 # ApplicationServer Classloader
#
ResourceType=Classloader
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:Classloader=mode#PARENT_LAST
AttributeInfo=classloaders
#DELETE=true

#
#Properties
#
mode=PARENT_LAST #ENUM(PARENT_FIRST|PARENT_LAST),default(PARENT_FIRST)
```

  2. Run the applyConfigProperties command to create a class loader configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing class loader.
  1. Obtain a properties file for the class loader that you want to change.

     You can extract a properties file for a Classloader object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.
  3. Run the applyConfigProperties command.
- If you no longer need a class loader, you can delete the entire class loader object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the class loader object.

What to do next

Save the changes to your configuration.

Working with library reference properties files:

You can use properties files to create or change library reference properties under a PARENT_LAST class loader that is under the ApplicationServer object of a server.

Before you begin

Determine the changes that you want to make to your library reference configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a library reference object and its configuration properties.

Run administrative commands using wsadmin to create or change library reference properties, validate the properties, and apply them to your configuration.

Table 511. Actions for library reference properties files. You can create, modify, and delete library reference configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a library reference and its properties.
  1. Create LibraryRef properties.

     Open an editor and specify library reference properties. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example defines a library reference under a PARENT_LAST class loader that is under the ApplicationServer object of a server.

```
#
# SubSection 1.0.0.0.0 # ApplicationServer Classloader
#
ResourceType=LibraryRef
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:
Classloader=mode#PARENT_LAST:LibraryRef=libraryName#mylibName
AttributeInfo=libraries
#DELETE=true


#
#Properties
#
libraryName=mylibName
sharedClassloader=false #boolean,default(true)
```

  2. Run the applyConfigProperties command to create a library reference configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing library reference.
  1. Obtain a properties file for the library reference that you want to change.
     You can extract a properties file for a LibraryRef object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.
     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need a library reference, you can delete the entire library reference object.
  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the library reference properties.

What to do next

Save the changes to your configuration.

Working with custom service properties files

You can use properties files to create or change custom service properties under a server.

Before you begin

Determine the changes that you want to make to your custom service configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a custom service object and its configuration properties.

Run administrative commands using wsadmin to create or modify a properties file for a custom service, validate the properties, and apply them to your configuration.

Table 512. Actions for custom service properties files. You can create, modify, and delete custom service properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.
create Property	Set properties and then run the applyConfigProperties command.
delete Property	To delete one or more properties, run deleteConfigProperties with only those properties to delete in the properties file.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a custom service and its properties.
  1. Create a properties file for a CustomService object.

     Open an editor and create a custom service properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for a CustomService object under a server follows. A property under CustomService with a name of `myName` and a value of `myVal` is shown in the example:

```
#
# SubSection 1.0 # CustomService
#
ResourceType=CustomService
ImplementingResourceType=CustomService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:CustomService=displayName#mydisplayName
AttributeInfo=customServices
#DELETE=true

#
#Properties
#
displayName=mydisplayName #required
classpath=myclasspath #required
enable=false #boolean,default(false)
externalConfigURL=null
context=null
description=null
classname=mclass.name #required
prerequisiteServices={} #ObjectName*(null)


#
# SubSection 1.0.1 # CustomService properties
#
ResourceType=CustomService
ImplementingResourceType=CustomService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:CustomService=displayName#mydisplayName
AttributeInfo=properties(name,value)
#

#
#Properties
#
myName=myVal

EnvironmentVariablesSection

#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

  2. Run the applyConfigProperties command to create a CustomService configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing custom service.
  1. Obtain a properties file for the custom service that you want to change.

     You can extract a properties file for a CustomService using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a custom service configuration.
- Delete the entire custom service object or its properties.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

Results

You can use the properties file to configure and manage the custom service object.

What to do next

Save the changes to your configuration.

Working with dynamic cache properties files

You can use properties files to change dynamic cache properties under a server.

Before you begin

Determine the changes that you want to make to your dynamic cache configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify a dynamic cache object and its configuration properties.

Run administrative commands using wsadmin to change a properties file for a dynamic cache, validate the properties, and apply them to your configuration.

Table 513. Actions for dynamic cache properties files. You can create, modify and delete dynamic cache properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available
create Property	Set properties and then run the applyConfigProperties command.
delete Property	To delete one or more properties, run deleteConfigProperties with only those properties to delete in the properties file.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit dynamic cache properties under a server.
  1. Specify DynamicCache properties in a properties file.

     Open an editor on a properties file. Example properties for a DynamicCache under a server follow. A property under DynamicCache has a name of `myName` and a value of `myVal`. You can copy the example properties into an editor and modify them as needed for your situation.

```
#
# SubSection 1.0 # DynamicCache
#
ResourceType=DynamicCache
ImplementingResourceType=DynamicCache
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:DynamicCache=
AttributeInfo=services
#

#
#Properties
#
defaultPriority=1 #integer,required,default(1)
diskOffloadLocation=null
```

```
context=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer= #ObjectName(ApplicationServer),readonly
flushToDiskOnStop=false #boolean,default(false)
enableCacheReplication=false #boolean,default(false)
diskCachePerformanceLevel=BALANCED #ENUM(HIGH|CUSTOM|BALANCED|LOW),default(BALANCED)
enableDiskOffload=false #boolean,default(false)
replicationType=NONE #ENUM(PULL|PUSH|PUSH_PULL|NONE),default(NONE)
diskCacheEntrySizeInMB=0 #integer,default(0)
enable=true #boolean,default(false)
cacheSize=2000 #integer,required,default(2000)
diskCacheSizeInGB=0 #integer,default(0)
pushFrequency=1 #integer,default(1)
hashSize=0 #integer,default(0)
diskCacheCleanupFrequency=0 #integer,default(0)
diskCacheSizeInEntries=0 #integer,default(0)
enableTagLevelCaching=false #boolean,default(false)


#
# Header
#
ResourceType=DynamicCache
ImplementingResourceType=DynamicCache
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:DynamicCache=
AttributeInfo=properties(name,value)
#

#
#Properties
#

myName=myVal
#

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to change a DynamicCache configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Delete dynamic cache properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage dynamic cache properties.

What to do next

Save the changes to your configuration.

Working with end point properties file

You can use properties files to change the end points of a server.

Before you begin

Determine the changes that you want to make to your end point configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify end point properties.

Run administrative commands using wsadmin to extract a properties file for an end point, validate the properties, and apply them to your configuration.

Table 514. Actions for end point properties files. You can modify end point properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Edit end point properties under a server.

   Open an editor on a properties file. Example properties for an EndPoint under a server follow. The example specifies three different host names in the Environment Variables section: `myHostName`,`*` and `localhost`. You can copy the example properties into an editor and modify them as needed for your situation.

```
#
# Header
#
ResourceType=EndPoint
ImplementingResourceType=Server
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}
#
#
#Properties
#
SOAP_CONNECTOR_ADDRESS=8888:!{hostName} # integer
SIP_DEFAULTHOST_SECURE=5067:!{hostName1} # integer
SIP_DEFAULTHOST=5066:!{hostName1} # integer
SIB_ENDPOINT_ADDRESS=7283:!{hostName1} # integer
WC_defaulthost_secure=9447:!{hostName1} # integer
DCS_UNICAST_ADDRESS=9364:!{hostName1} # integer
SIB_MQ_ENDPOINT_SECURE_ADDRESS=5582:!{hostName1} # integer
WC_adminhost_secure=9052:!{hostName1} # integer
CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS=9424:!{hostName} # integer
ORB_LISTENER_ADDRESS=0:!{hostName} # integer
BOOTSTRAP_ADDRESS=2813:!{hostName} # integer
CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS=9423:!{hostName} # integer
IPC_CONNECTOR_ADDRESS=9634:!{hostName2} # integer
SIB_ENDPOINT_SECURE_ADDRESS=7291:!{hostName1} # integer
WC_defaulthost=9084:!{hostName1} # integer
SIB_MQ_ENDPOINT_ADDRESS=5562:!{hostName1} # integer
SAS_SSL_SERVERAUTH_LISTENER_ADDRESS=9422:!{hostName} # integer
WC_adminhost=9069:!{hostName1} # integer
#
EnvironmentVariablesSection
#
#Environment Variables
hostName2=localhost
hostName1=*
hostName=myHostName
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to configure and manage the end point properties.

What to do next

Save the changes to your configuration.

Working with EJB container properties files

You can use properties files to change Enterprise JavaBeans (EJB) container properties and associated stateManagement, cacheSettings and timerSettings attributes under a server.

Before you begin

Determine the changes that you want to make to your EJB container configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify an EJB container object and its configuration properties.

Run administrative commands using wsadmin to change a properties file for an EJB container, validate the properties, and apply them to your configuration.

Table 515. Actions for EJB container properties files. You can create, modify and delete EJB container properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available
create Property	Set properties and then run the applyConfigProperties command.
delete Property	To delete one or more properties, run deleteConfigProperties with only those properties to delete in the properties file.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit EJB container properties under a server.
  1. Specify EJBContainer properties in a properties file.

     Open an editor on a properties file. Example properties for an EJBContainer under a server follow. A property under EJBContainer has a name of `myName` and a value of `myVal`. You can copy the example properties into an editor and modify the `stateManagement`, `cacheSettings` and `timerSettings` attributes under the server as needed for your situation.

```
#
# SubSection 1.0 # EJBContainer
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=
AttributeInfo=components
#

#
#Properties
#
EJBTimer={} #ObjectName*(null)
name=null
defaultDatasourceJNDIName=null
inactivePoolCleanupInterval=30000 #long,default(30000)
passivationDirectory="${USER_INSTALL_ROOT}/temp" #required
```

```
enableSFSBFailover=false #boolean,default(false)
server=null
parentComponent=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer= #ObjectName(ApplicationServer),readonly


#
# SubSection 1.0.1 # EJBContainer State Management
#
ResourceType=StateManageable
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=:StateManageable=
AttributeInfo=stateManagement
#

#
#Properties
#
initialState=START #ENUM(STOP|START),default(START)
managedObject=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer= #ObjectName(EJBContainer),readonly


#
# SubSection 1.0.3 # EJBTimer
#
ResourceType=EJBTimer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=:EJBTimer=
AttributeInfo=timerSettings
#

#
#Properties
#
tablePrefix=EJBTIMER_
pollInterval=300 #long,default(300)
numAlarmThreads=1 #long,default(1)
schedulerJNDIName=null
datasourceJNDIName="jdbc/DefaultEJBTimerDataSource"
datasourceAlias=null


#
# SubSection 1.0.5 # EJBCache
#
ResourceType=EJBCache
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=:EJBCache=
AttributeInfo=cacheSettings
#

#
#Properties
#
cleanupInterval=3000 #long,default(3000)
cacheSize=2053 #long,default(2053)

#
# Header
#
ResourceType=EJBContainer
ImplementingResourceType=EJBContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=
AttributeInfo=properties(name,value)
#
#

#
#Properties
#

myName=myVal
#

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to change an EJBContainer configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Delete EJB container properties.

To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the EJB container properties.

What to do next

Save the changes to your configuration.

Working with HTTP transport properties files

You can use properties files to create or change HTTP transport properties.

Before you begin

Determine the changes that you want to make to your HTTP transport configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an HTTP transport object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for an HTTP transport, validate the properties, and apply them to your configuration.

Table 516. Actions for HTTP transport properties files. You can create, modify, and delete HTTP transport properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.
create Property	Set properties and then run the applyConfigProperties command.
delete Property	To delete one or more properties, run deleteConfigProperties with only those properties to delete in the properties file.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create an HTTP transport and its properties.
  1. Create a properties file for an HTTPTransport object.

     Open an editor and create an HTTP transport properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for an HTTPTransport under a WebContainer of a server follows. An HTTPTransport property has a name of `myName` and a value of `myVal`:

```
#
# Header
#
ResourceType=HTTPTransport
```

```
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=:HTTPTransport=
AttributeInfo=transports
#DELETE=true
#
#Properties
#
sslEnabled=false #boolean,default(false)
sslConfig=mysslConfig
external=false #boolean,default(false)
#

#
# Header
#
ResourceType=HTTPTransport
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=:HTTPTransport=
AttributeInfo=properties(name,value)
#
#Properties
#
myName=myVal

EnvironmentVariablesSection

#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create an HTTPTransport configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing HTTP transport.
  1. Obtain a properties file for the HTTP transport that you want to change.

     You can extract a properties file for an HTTPTransport using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change an HTTP transport configuration.
- Delete the entire HTTP transport object or its properties.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

Results

You can use the properties file to configure and manage the HTTP transport object.

What to do next

Save the changes to your configuration.

Working with listener port properties files

You can use properties files to create or change Message Listener Service listener port properties under a server and associated StateManageable objects.

Before you begin

Determine the changes that you want to make to your listener port configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a listener port object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a listener port, validate the properties, and apply them to your configuration.

Table 517. Actions for Message Listener Service listener port properties files. You can create, modify, and delete listener port configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a listener port and its properties.
  1. Create ListenerPort properties in a properties file.

     Open an editor and specify listener port properties. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example defines a listener port under the MessageListenerService under a server, and the associated StateManageable object.

```
#
# SubSection 1.0.0 # ListenerPort
#
ResourceType=ListenerPort
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=
:MessageListenerService=:ListenerPort=myName
AttributeInfo=listenerPorts
#DELETE=true

#
#Properties
#
destinationJNDIName=dName #required
connectionFactoryJNDIName=myjName #required
name=myName #required
maxMessages=1 #integer,default(1)
description=My description
maxSessions=1 #integer,default(1)
maxRetries=0 #integer,default(0)


#
# SubSection 1.0.0.0 # ListenerPort State Management
#
ResourceType=StateManageable
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=
:MessageListenerService=:ListenerPort=myName:StateManageable=
AttributeInfo=stateManagement
#

#
#Properties
#
initialState=START #ENUM(STOP|START),default(START)
managedObject=myName
```

```
EnvironmentVariablesSection

#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create a listener port configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing listener port.
  1. Obtain a properties file for the listener port that you want to change.

     You can extract a properties file for a ListenerPort object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a listener port configuration.
- If you no longer need a listener port, you can delete the entire listener port object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the listener port.

What to do next

Save the changes to your configuration.

Working with Object Request Broker properties files

You can use properties files to change Object Request Broker (ORB) properties and associated interceptors, plug-ins, properties and thread pool attributes under a server.

Before you begin

Determine the changes that you want to make to your Object Request Broker configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete Object Request Broker configuration properties.

Run administrative commands using wsadmin to change a properties file for an Object Request Broker, validate the properties, and apply them to your configuration.

Table 518. Actions for Object Request Broker properties files. You can create, modify, and delete Object Request Broker properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Not applicable

Table 518. Actions for Object Request Broker properties files (continued). You can create, modify, and delete Object Request Broker properties.

Action	Procedure
create Property, Plug-ins, or Interceptors	Set properties and then run the applyConfigProperties command.
delete Property, Plug-ins, or Interceptors	Specify those properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create Object Request Broker properties.
  1. Specify ObjectRequestBroker properties in a properties file.

     Open an editor and specify Object Request Broker properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. You can use properties files to change Object Request Broker properties and associated interceptors, plug-ins, properties and thread pool attributes under a server.

```
# SubSection 1.0 # ObjectRequestBroker Service
#
ResourceType=ObjectRequestBroker
ImplementingResourceType=ObjectRequestBroker
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ObjectRequestBroker=
AttributeInfo=services

#
#Properties
#
requestTimeout=180 #integer,required,default(0)
context=!{serverName}
forceTunnel=never
tunnelAgentURL=null
connectionCacheMaximum=240 #integer,required,default(0)
requestRetriesDelay=0 #integer,required,default(0)
requestRetriesCount=1 #integer,required,default(0)
useServerThreadPool=false #boolean,default(false)
connectionCacheMinimum=100 #integer,required,default(0)
enable=true #boolean,default(false)
commTraceEnabled=false #boolean,default(false)
locateRequestTimeout=180 #integer,required,default(0)
noLocalCopies=false #boolean,default(false)

#
# SubSection 1.0.1 # ORBInterceptors
#
ResourceType=ObjectRequestBroker
ImplementingResourceType=ObjectRequestBroker
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ObjectRequestBroker=
AttributeInfo=interceptors(name,null)

#
#Properties
#
com.ibm.ISecurityLocalObjectBaseL13Impl.CSIClientRI=
com.ibm.debug.olt.ivbtrjrt.OLT_RI=
com.ibm.ws.wlm.client.WLMClientInitializer=
com.ibm.ws.runtime.workloadcontroller.OrbWorkloadRequestInterceptor=
com.ibm.ws.activity.remote.cos.ActivityServiceServerInterceptor=
com.ibm.ISecurityLocalObjectBaseL13Impl.ClientRIWrapper=
com.ibm.debug.DebugPortableInterceptor=
com.ibm.ws.wlm.server.WLMServerInitializer=
com.ibm.ws.Transaction.JTS.TxInterceptorInitializer=
com.ibm.ISecurityLocalObjectBaseL13Impl.SecurityComponentFactory=
com.ibm.ISecurityLocalObjectBaseL13Impl.ServerRIWrapper=
com.ibm.ISecurityLocalObjectBaseL13Impl.CSIServerRI=
com.ibm.ejs.ras.RasContextSupport=

#
# SubSection 1.0.2 # ORBPlugins
#
ResourceType=ObjectRequestBroker
ImplementingResourceType=ObjectRequestBroker
```

```
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ObjectRequestBroker=
AttributeInfo=plugins(name,null)
#

#
#Properties
#
com.ibm.ws.orbimpl.WSORBPropertyManager=
com.ibm.ws.wlm.client.WLMClient=
com.ibm.ws.pmi.server.modules.OrbPerfModule=
com.ibm.ISecurityUtilityImpl.SecurityPropertyManager=
com.ibm.ws.csi.CORBAORBMethodAccessControl=
com.ibm.ws.orbimpl.transport.WSTransport=

#
# SubSection 1.0.3 # Thread pool for ORB
#
ResourceType=ThreadPool
ImplementingResourceType=ObjectRequestBroker
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ObjectRequestBroker=:ThreadPool=
AttributeInfo=threadPool
#

#
#Properties
#
maximumSize=50 #integer,required,default(5)
name=ORB.thread.pool
inactivityTimeout=3500 #integer,required,default(5000)
minimumSize=10 #integer,required,default(1)
isGrowable=false #boolean,default(false)

#
# SubSection 1.0.4 # ORBProperties
#
ResourceType=ObjectRequestBroker
ImplementingResourceType=ObjectRequestBroker
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ObjectRequestBroker=
AttributeInfo=properties(name,value)

#
#Properties
#
com.ibm.ws.orb.transport.WSSSLClientSocketFactoryName=com.ibm.ws.security.orbssl.WSSSLClientSocketFactoryImpl
com.ibm.CORBA.RasManager=com.ibm.websphere.ras.WsOrbRasManager
com.ibm.CORBA.ConnectionInterceptorName=com.ibm.ISecurityLocalObjectBaseL13Impl.SecurityConnectionInterceptor
com.ibm.ws.orb.transport.useMultiHome=true
com.ibm.ws.orb.transport.WSSSLServerSocketFactoryName=com.ibm.ws.security.orbssl.WSSSLServerSocketFactoryImpl
com.ibm.CORBA.enableLocateRequest=true
com.ibm.websphere.management.registerServerIORWithLSD=true

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create an ObjectRequestBroker configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing Object Request Broker properties.
  1. Obtain a properties file for the Object Request Broker that you want to change.

     You can extract a properties file for an ObjectRequestBroker using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the Object Request Broker properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage Object Request Broker.

What to do next

Save the changes to your configuration.

Working with PMI service properties files

You can use properties files to create or change Performance Monitoring Infrastructure (PMI) service properties under a server.

Before you begin

Determine the changes that you want to make to your PMI service configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a PMI service configuration properties.

Run administrative commands using wsadmin to change a properties file for a PMI service, validate the properties, and apply them to your configuration.

Table 519. Actions for PMI service properties files. You can create, modify, and delete PMI service properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create PMI service properties.
  1. Specify PMIService properties in a properties file.

     Open an editor and specify PMI service properties under a server in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example shows a property under PMIService with name `myName` and value `myVal`.

```
#
# Header
#
ResourceType=PMIService
ImplementingResourceType=PMIService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:PMIService=
AttributeInfo=services
#

#
#Properties
#
```

```
synchronizedUpdate=false #boolean,default(false)
enable=true #boolean,default(false)
context=!{serverName}
statisticSet=basic
initialSpecLevel=


#
# Header
#
ResourceType=PMIService
ImplementingResourceType=PMIService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:PMIService=
AttributeInfo=properties(name,value)
#

#Properties
#
myName=myVal
#

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create a PMIService configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing PMI service properties.
  1. Obtain a properties file for the PMI service that you want to change.

     You can extract a properties file for a PMIService using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the PMI service properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the PMI service.

What to do next

Save the changes to your configuration.

Working with process definition properties files

You can use properties files to change the Java process definition of a server and the associated process execution, logs, monitoring policy, and Java virtual machine (JVM) settings under a server.

Before you begin

Determine the changes that you want to make to your Java process definition configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify a Java process definition properties.

Run administrative commands using wsadmin to extract a properties file for a Java process definition, validate the properties, and apply them to your configuration.

Table 520. Actions for Java process definition properties files. You can modify Java process definition properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Edit JavaProcessDef properties file under a server.

   Open an editor on a properties file and edit the JavaProcessDef and associated process execution, logs, monitoring policy, and JVM properties under a server. You can copy the following example properties into an editor and modify them as needed for your situation:

```
#
# Header
#
ResourceType=JavaProcessDef
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=
AttributeInfo=processDefinitions
#

#
#Properties
#
executableTarget=com.ibm.ws.runtime.WsServer
executableName=null
stopCommand=null
stopCommandArgs={}
terminateCommand=null
workingDirectory="${USER_INSTALL_ROOT}" #required
startCommandArgs={}
executableArguments={}
startCommand=null
executableTargetKind=JAVA_CLASS #ENUM(EXECUTABLE_JAR|JAVA_CLASS),default(JAVA_CLASS)
terminateCommandArgs={}
processType=null


#
# Header
#
ResourceType=ProcessExecution
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:ProcessExecution=
AttributeInfo=execution
#

#
#Properties
#
runAsUser=
runAsGroup=
runInProcessGroup=0 #integer,default(0)
umask=022 #default(022)
processPriority=20 #integer,default(20)


#
# Header
#
ResourceType=OutputRedirect
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:OutputRedirect=
AttributeInfo=ioRedirect
```

```
#

#
#Properties
#
stdinFilename=null
stderrFilename="${SERVER_LOG_ROOT}/native_stderr.log" #required
stdoutFilename="${SERVER_LOG_ROOT}/native_stdout.log" #required

# Header
#
ResourceType=MonitoringPolicy
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:MonitoringPolicy=
AttributeInfo=monitoringPolicy

#Properties
#
maximumStartupAttempts=3 #integer,required,default(0)
pingTimeout=300 #integer,required,default(0)
pingInterval=60 #integer,default(0)
nodeRestartState=STOPPED #ENUM(PREVIOUS|STOPPED|RUNNING),default(STOPPED)
autoRestart=true #boolean,default(true)

#
# Header : Make sure JavaHome is not in the property list or it is unchanged as it is readonly
#
ResourceType=JavaVirtualMachine
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:JavaVirtualMachine=
AttributeInfo=jvmEntries

#Properties
#
internalClassAccessMode=ALLOW #ENUM(ALLOW|RESTRICT),default(ALLOW)
JavaHome="C:\WAS70.cf050923.16/java" #readonly
debugArgs="-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777"
classpath={}
initialHeapSize=0 #integer,default(0)
runHProf=false #boolean,default(false)
genericJvmArguments=
hprofArguments=
osName=null
bootClasspath={}
verboseModeJNI=false #boolean,default(false)
maximumHeapSize=0 #integer,default(0)
disableJIT=false #boolean,default(false)
verboseModeGarbageCollection=false #boolean,default(false)
executableJarFileName=null
verboseModeClass=false #boolean,default(false)
debugMode=false #boolean,default(false)

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to configure and manage the Java process definition.

What to do next

Save the changes to your configuration.

Working with SOAP connector properties files

You can use properties files to create or change SOAP connector properties.

Before you begin

Determine the changes that you want to make to your SOAP connector configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete SOAP connector configuration properties.

Run administrative commands using wsadmin to change a properties file for a SOAP connector, validate the properties, and apply them to your configuration.

Table 521. Actions for SOAP connector properties files. You can create, modify, and delete SOAP connector properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create SOAP connector properties.
  1. Specify SOAPConnector properties in a properties file.

     Open an editor and specify SOAP connector properties under a server in a properties file. You can copy the following example properties file into an editor and modify the properties as needed for your situation. The example shows a SOAPConnector property with name `myName` and value `myVal`.

```
#
# Header
#
ResourceType=SOAPConnector
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:AdminService=:SOAPConnector=
AttributeInfo=connectors
#
#Properties
#
enable=true #boolean,default(true)
#

#
# Header
#
ResourceType=SOAPConnector
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:AdminService=:SOAPConnector=
AttributeInfo=properties(name,value)
#
#Properties
#
myName=myVal

EnvironmentVariablesSection

#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

  2. Run the applyConfigProperties command to create a SOAPConnector configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing SOAP connector properties.
  1. Obtain a properties file for the SOAP connector that you want to change.
     You can extract a properties file for a SOAPConnector using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.
     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the SOAP connector properties.
  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the SOAP connector.

What to do next

Save the changes to your configuration.

Modifying the errorStreamRedirect attribute of StreamRedirect properties files

You can use properties files to change errorStreamRedirect attribute of a server. The errorStreamRedirect attribute is of type StreamRedirect.

Before you begin

Determine the changes that you want to make to the errorStreamRedirect attribute of a server configuration object.

You can create a server by creating a Server properties file template and then applying the template. See the topic on server properties files.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify the errorStreamRedirect attribute of a server.

Run administrative commands using wsadmin to change a properties file for a StreamRedirect object, validate the properties, and apply them to your configuration.

Table 522. Actions for the errorStreamRedirect attribute of StreamRedirect properties files. You can modify the errorStreamRedirect attribute value.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

1. Change the errorStreamRedirect attribute in a StreamRedirect properties file.

   Open an editor and specify the errorStreamRedirect attribute under a server in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example shows an errorStreamRedirect attribute of type StreamRedirect:

```
#
# Header
#
ResourceType=StreamRedirect
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:StreamRedirect=outputStreamRedirect#
AttributeInfo=errorStreamRedirect
#DELETE=true
#
#
#Properties
#
formatWrites=true #boolean,default(false)
maxNumberOfBackupFiles=5 #integer,default(0)
suppressStackTrace=false #boolean,default(false)
rolloverType=SIZE #ENUM(TIME|SIZE|NONE|BOTH),default(SIZE)
suppressWrites=false #boolean,default(false)
baseHour=24 #integer,default(0)
fileName="${SERVER_LOG_ROOT}/SystemOut6.log"
messageFormatKind=BASIC #ENUM(BASIC|ADVANCED),default(BASIC)
rolloverSize=1 #integer,default(0)
rolloverPeriod=24 #integer,default(0)

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to change the errorStreamRedirect attribute of a server.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

Results

You can use the properties file to configure and manage the StreamRedirect object.

What to do next

Save the changes to your configuration.

Working with thread pool properties files

You can use properties files to create or change Message Listener Service thread pool properties under a server.

Before you begin

Determine the changes that you want to make to your thread pool configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a thread pool object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a thread pool, validate the properties, and apply them to your configuration.

Table 523. Actions for Message Listener Service thread pool properties files. You can create, modify, and delete thread pool configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a thread pool and its properties.
  1. Create ThreadPool properties in a properties file.

     Open an editor and specify thread pool properties. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example defines a thread pool under the MessageListenerService under a server.

```
#
# SubSection 1.0.1 # MessageListenerService ThreadPool
#
ResourceType=ThreadPool
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:EJBContainer=:MessageListenerService=:ThreadPool=
AttributeInfo=threadPool
#

#
#Properties
#
maximumSize=50 #integer,required,default(5)
name=Message.Listener.Pool
minimumSize=10 #integer,required,default(1)
inactivityTimeout=3500 #integer,required,default(5000)
description=null
isGrowable=false #boolean,default(false)

EnvironmentVariablesSection

#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

  2. Run the applyConfigProperties command to create a thread pool configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing thread pool.
  1. Obtain a properties file for the thread pool that you want to change.

     You can extract a properties file for a ThreadPool object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a thread pool configuration.
- If you no longer need a thread pool, you can delete the entire thread pool object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the thread pool.

What to do next

Save the changes to your configuration.

Working with trace service properties files

You can use properties files to create or change trace service properties and the associated trace log under a server.

Before you begin

Determine the changes that you want to make to your trace service configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

Note: This topic references one or more of the application server log files. Beginning in WebSphere Application Server Version 8.0 you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files or native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.

About this task

Using a properties file, you can create, modify, or delete a trace service object and its configuration properties.

Run administrative commands using wsadmin to change a properties file for a trace service, validate the properties, and apply them to your configuration.

Table 524. Actions for trace service properties files. You can create, modify, and delete trace service properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

`AdminTask.command_name('-interactive')`

Procedure

- Create trace service properties.
  1. Specify TraceService properties in a properties file.

Open an editor and specify trace service properties and an associated TraceLog under a server in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example shows a property under TraceService with name `myName` and value `myVal`.

```
#
# Header
#
ResourceType=TraceService
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TraceService=
AttributeInfo=services
#

#
#Properties
#
startupTraceSpecification="*=info"
enable=true #boolean,default(false)
context=!{serverName}
memoryBufferSize=8 #integer,required,default(8)
traceFormat=BASIC #ENUM(LOG_ANALYZER|BASIC|ADVANCED),default(BASIC)
traceOutputType=SPECIFIED_FILE #ENUM(SPECIFIED_FILE|MEMORY_BUFFER),default(MEMORY_BUFFER)


#
# Header
#
ResourceType=TraceLog
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TraceService=:TraceLog=
AttributeInfo=traceLog
#

#
#Properties
#
maxNumberOfBackupFiles=5 #integer,default(1)
rolloverSize=20 #integer,default(100)
fileName="${SERVER_LOG_ROOT}/trace.log"


#
# Header
#
ResourceType=TraceService
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TraceService=
AttributeInfo=properties(name,value)
#
#
#Properties
myName=myVal
#

#
EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create a TraceService configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing trace service or associated TraceLog properties.
  1. Obtain a properties file for the trace service that you want to change.

     You can extract a properties file for a TraceService using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the trace service properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the trace service object.

What to do next

Save the changes to your configuration.

Working with transaction service properties files

You can use properties files to change transaction service properties under a server.

Before you begin

Determine the changes that you want to make to your transaction service configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete transaction service configuration properties.

Run administrative commands using wsadmin to change a properties file for a transaction service, validate the properties, and apply them to your configuration.

Table 525. Actions for transaction service properties files. You can create, modify, and delete transaction service properties.

Action	Procedure
create	Not available
modify	Edit properties and then run the applyConfigProperties command.
delete	Not available
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create transaction service properties.
  1. Specify TransationService properties in a properties file.

     Open an editor and specify transaction service properties under a server in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example shows a property under TransationService with name `myName` and value `myVal`.

```
#
# SubSection 1.0 # TransactionService
#
ResourceType=TransactionService
ImplementingResourceType=TransactionService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:TransactionService=
AttributeInfo=services
#

#
#Properties
#
```

```
httpProxyPrefix=
transactionLogDirectory=null
propogatedOrBMTTranLifetimeTimeout=300 #integer,required,default(0)
context=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer= #ObjectName(ApplicationServer),readonly
asyncResponseTimeout=30 #integer,required,default(30)
maximumTransactionTimeout=0 #integer,required,default(0)
acceptHeuristicHazard=false #boolean,default(false)
wstxURLPrefixSpecified=false #default(false)
waitForCommitOutcome=false #boolean,default(false)
totalTranLifetimeTimeout=120 #integer,required,default(0)
heuristicRetryLimit=0 #integer,required,default(0)
enable=true #boolean,default(false)
enableFileLocking=true #boolean,default(true)
secureWSTXTransportChain=null
enableLoggingForHeuristicReporting=false #boolean,default(false)
WSTransactionSpecificationLevel=WSTX_10 #ENUM(WSTX_11|WSTX_10),default(WSTX_10)
heuristicRetryWait=0 #integer,required,default(0)
httpsProxyPrefix=
LPSHeuristicCompletion=ROLLBACK #ENUM(MANUAL|COMMIT|ROLLBACK),default(ROLLBACK)
clientInactivityTimeout=60 #integer,required,default(0)
enableProtocolSecurity=true #boolean,default(true)


#
# SubSection 1.0.1 # TransactionService properties
#
ResourceType=TransactionService
ImplementingResourceType=TransactionService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:TransactionService=
AttributeInfo=properties(name,value)
#

#
#Properties
#

myName=myVal
#

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create a TransationService configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing transaction service properties.
  1. Obtain a properties file for the transaction service that you want to change.

     You can extract a properties file for a TransationService using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the transaction service properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the transaction service object.

What to do next

Save the changes to your configuration.

Working with web container properties files

You can use properties files to change web container properties and associated stateManagement and threadPool attributes under a server.

Before you begin

Determine the changes that you want to make to your Web container configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a web container configuration properties.

Run administrative commands using wsadmin to change a properties file for a web container, validate the properties, and apply them to your configuration.

Table 526. Actions for web container properties files. You can create, modify, and delete web container properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create web container properties.
  1. Specify WebContainer properties in a properties file.

     Open an editor and specify web container properties WebContainer and associated stateManagement and threadPool attributes under a server in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. The example shows a property under WebContainer with name `myName` and value `myVal`.

```
#
# SubSection 1.0 # WebContainer Component
#
ResourceType=WebContainer
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=
AttributeInfo=components
#

#
#Properties
#
enableServletCaching=false #boolean,default(false)
name=null
defaultVirtualHostName=null
server=null
maximumPercentageExpiredEntries=15 #integer,default(15)
asyncIncludeTimeout=60000 #integer,default(60000)
parentComponent=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer= #ObjectName(ApplicationServer),readonly
disablePooling=false #boolean,default(false)
sessionAffinityFailoverServer=null
maximumResponseStoreSize=100 #integer,default(100)
allowAsyncRequestDispatching=false #boolean,default(false)
sessionAffinityTimeout=0 #integer,default(0)
```

```
#
# SubSection 1.0.1 # WebContainer State Management
#
ResourceType=StateManageable
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=:StateManageable=
AttributeInfo=stateManagement
#

#
#Properties
#
initialState=START #ENUM(STOP|START),default(START)
managedObject=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer= #ObjectName(WebContainer),readonly


#
# SubSection 1.0.3 # WebContainer ThreadPool
#
ResourceType=ThreadPool
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=:ThreadPool=
AttributeInfo=threadPool
#

#
#Properties
#
maximumSize=10 #integer,required,default(5)
name=null
minimumSize=0 #integer,required,default(1)
inactivityTimeout=50 #integer,required,default(5000)
description=null
isGrowable=false #boolean,default(false)


#
# SubSection 1.0.3.1 # WebContainer properties
#
ResourceType=ThreadPool
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=:ThreadPool=
AttributeInfo=customProperties(name,value)
#

#
#Properties
#
myName=myVal
#

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create a WebContainer configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing web container properties.
  1. Obtain a properties file for the web container that you want to change.

     You can extract a properties file for a WebContainer using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the web container properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the web container object.

What to do next

Save the changes to your configuration.

Working with web container session manager properties files:

You can use properties files to change session manager properties under the web container and associated tuningParams, sessionDatabasePersistence, and defaultCookieSettings attributes under a server.

Before you begin

Determine the changes that you want to make to your session manager configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a session manager configuration properties.

Run administrative commands using wsadmin to change a properties file for a session manager, validate the properties, and apply them to your configuration.

Table 527. Actions for session manager properties files. You can create, modify, and delete session manager properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create session manager properties.
  1. Specify SessionManager properties under the WebContainer in a properties file.

     In an editor, specify WebContainer SessionManager properties and associated tuningParams, sessionDatabasePersistence, and defaultCookieSettings attributes under a server in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation.

```
#
# SubSection 1.0.7 # Session Manager
#
ResourceType=SessionManager
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
:WebContainer=:SessionManager=
AttributeInfo=services
```

```
#

#
#Properties
#
enableSecurityIntegration=false #boolean,default(false)
maxWaitTime=5 #integer,default(0)
context=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=
 #ObjectName(WebContainer),readonly
allowSerializedSessionAccess=false #boolean,default(false)
enableProtocolSwitchRewriting=false #boolean,default(false)
enableUrlRewriting=false #boolean,default(false)
enable=true #boolean,default(false)
accessSessionOnTimeout=true #boolean,default(true)
enableSSLTracking=false #boolean,default(false)
sessionPersistenceMode=NONE #ENUM(DATABASE|DATA_REPLICATION|NONE),default(NONE)
enableCookies=true #boolean,default(true)

#
# SubSection 1.0.7.1 # Session Manager's Tuning Parameters
#
ResourceType=TuningParams
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=
:SessionManager=:TuningParams=
AttributeInfo=tuningParams
#

#
#Properties
#
writeContents=ONLY_UPDATED_ATTRIBUTES #ENUM(ALL_SESSION_ATTRIBUTES|ONLY_UPDATED_ATTRIBUTES),
default(ONLY_UPDATED_ATTRIBUTES)usingMultiRowSchema=false #boolean,default(false)
allowOverflow=true #boolean,default(true)
writeFrequency=TIME_BASED_WRITE #ENUM(TIME_BASED_WRITE|END_OF_SERVLET_SERVICE|MANUAL_UPDATE),
default(END_OF_SERVLET_SERVICE)
invalidationTimeout=30 #integer,default(30)
scheduleInvalidation=false #boolean,default(false)
writeInterval=10 #integer,default(120)
maxInMemorySessionCount=1000 #integer,default(1000)

#
# SubSection 1.0.7.1.1 # Session Manager's Tuning Parameters
#
ResourceType=InvalidationSchedule
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=
:SessionManager=:TuningParams=:InvalidationSchedule=
AttributeInfo=invalidationSchedule
#

#
#Properties
#
secondHour=2 #default(0)
firstHour=14 #default(0)

#
# SubSection 1.0.7.3 # Session Manager's SessionDatabasePersistence
#
ResourceType=SessionDatabasePersistence
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=
:WebContainer=:SessionManager=:SessionDatabasePersistence=
AttributeInfo=sessionDatabasePersistence
#

#
#Properties
#
password="{xor}Oz1tPjsyNjE="
userId=db2admin
tableSpaceName=
datasourceJNDIName="jdbc/Sessions" #required
db2RowSize=ROW_SIZE_4KB #ENUM(ROW_SIZE_4KB|ROW_SIZE_32KB|ROW_SIZE_16KB|ROW_SIZE_8KB),default(ROW_SIZE_4KB)

#
# SubSection 1.0.7.4 # Session Manager's Cookie
#
ResourceType=Cookie
ImplementingResourceType=WebContainer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:ApplicationServer=:WebContainer=
:SessionManager=:Cookie=
AttributeInfo=defaultCookieSettings
#

#
#Properties
#
maximumAge=-1 #integer,default(-1)
```

```
name=JSESSIONID #default(JSESSIONID)
domain=
secure=false #boolean,default(false)
path=/ #default(/)

EnvironmentVariablesSection
#
#Environment Variables
cellName=WASCell06
serverName=myServer
nodeName=WASNode04
```

2. Run the applyConfigProperties command to create a WebContainer SessionManager configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing session manager properties.
  1. Obtain a properties file for the session manager that you want to change.

     You can extract a properties file for a WebContainer SessionManager using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the session manager properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the session manager.

What to do next

Save the changes to your configuration.

Transport channel service

You can use properties files to create, modify, or delete inbound channel properties and custom properties.

Working with HTTP inbound channel properties files

You can use properties files to create, modify, or delete HTTP inbound channel properties and custom properties.

Before you begin

Determine the changes that you want to make to your HTTP inbound channel configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an HTTP inbound channel object. You can also create, modify, or delete HTTP inbound channel custom properties.

Run administrative commands using wsadmin to create or change a properties file for an HTTP inbound channel, validate the properties, and apply them to your configuration.

Table 528. Actions for HTTP inbound channel properties files. You can create, modify, and delete HTTP inbound channel objects.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire HTTPInboundChannel object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 529. Actions for HTTP inbound channel custom properties. You can create, modify, and delete HTTP inbound channel custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit an HTTP inbound channel properties file.
  1. Set HTTPInboundChannel object properties as needed.

     Open an editor on an HTTPInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example HTTPInboundChannel properties file follows:

```
#
# Header
#
ResourceType=HTTPInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:HTTPInboundChannel=myHTTPIC
#DELETE=true
#

#
#Properties
#
enableLogging=false #boolean,default(false)
name= myHTTPIC #required
readTimeout=60 #integer,required,default(60)
maxFieldSize=32768 #integer,default(32768)
useChannelAccessLoggingSettings=false #boolean,default(false)
maxRequestMessageBodySize=-1 #integer,default(-1)
maximumPersistentRequests=100 #integer,required,default(100)
discriminationWeight=10 #integer,default(0)
persistentTimeout=30 #integer,required,default(30)
maxHeaders=50 #integer,default(50)
keepAlive=true #boolean,default(true)
useChannelErrorLoggingSettings=false #boolean,default(false)
useChannelFRCALoggingSettings=false #boolean,default(false)
writeTimeout=60 #integer,required,default(60)
```

```
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

2. Run the applyConfigProperties command to change an HTTP inbound channel configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit HTTP inbound channel custom properties.
  1. Set HTTPInboundChannel custom properties as needed.

     Open an editor on an HTTPInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=HTTPInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:HTTPInboundChannel=myHTTPIC
AttributeInfo=properties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Run the applyConfigProperties command.
- If you no longer need the HTTP inbound channel or an existing custom property, you can delete the entire HTTP inbound channel object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the HTTP inbound channel object and its properties.

What to do next

Save the changes to your configuration.

Working with SSL inbound channel properties files

You can use properties files to create, modify, or delete SSL inbound channel properties and custom properties.

Before you begin

Determine the changes that you want to make to your SSL inbound channel configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete an SSL inbound channel object. You can also create, modify, or delete SSL inbound channel custom properties.

Run administrative commands using wsadmin to create or change a properties file for an SSL inbound channel, validate the properties, and apply them to your configuration.

Table 530. Actions for SSL inbound channel properties files. You can create, modify, and delete SSL inbound channel objects.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire SSLInboundChannel object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 531. Actions for SSL inbound channel custom properties. You can create, modify, and delete SSL inbound channel custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create or edit an SSL inbound channel properties file.
  1. Set SSLInboundChannel object properties as needed.

     Open an editor on an SSLInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example SSLInboundChannel properties file follows:

```
#
# Header
#
ResourceType=SSLInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:SSLInboundChannel=mySSLIC
#DELETE=true
#

#
#Properties
#
name=mySSLIC #required
discriminationWeight=1 #integer,default(0)
sslConfigAlias=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

2. Run the applyConfigProperties command to change an SSL inbound channel configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit SSL inbound channel custom properties.
  1. Set SSLInboundChannel custom properties as needed.

     Open an editor on an SSLInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=SSLInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:SSLInboundChannel=mySSLIC
AttributeInfo=properties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Run the applyConfigProperties command.
- If you no longer need the SSL inbound channel or an existing custom property, you can delete the entire SSL inbound channel object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the SSL inbound channel object and its properties.

What to do next

Save the changes to your configuration.

Working with TCP inbound channel properties files

You can use properties files to create, modify, or delete TCP inbound channel properties and custom properties.

Before you begin

Determine the changes that you want to make to your TCP inbound channel configuration or its configuration objects.

About this task

Using a properties file, you can create, modify, or delete a TCP inbound channel object. You can also create, modify, or delete TCP inbound channel custom properties.

Run administrative commands using wsadmin to create or change a properties file for a TCP inbound channel, validate the properties, and apply them to your configuration.

Table 532. Actions for TCP inbound channel properties files. You can create, modify, and delete TCP inbound channel objects.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire TCPInboundChannel object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 533. Actions for TCP inbound channel custom properties. You can create, modify, and delete TCP inbound channel custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit a TCP inbound channel properties file.
  1. Set TCPInboundChannel object properties as needed.

     Open an editor on a TCPInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example TCPInboundChannel properties file follows:

```
#
# Header
#
ResourceType=TCPInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:TCPInboundChannel=myTCPIC
AttributeInfo=transportChannels
#DELETE=true
#

#
#Properties
#
maxOpenConnections=20000 #integer,required,default(20000)
name= myTCPIC #required
hostNameIncludeList={}
hostNameExcludeList={}
endPointName=WC_adminhost
inactivityTimeout=60 #integer,required,default(60)
discriminationWeight=0 #integer,default(0)
addressIncludeList={}
addressExcludeList={}
threadPool=WebContainer #ObjectName(ThreadPool)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Run the applyConfigProperties command to change a TCP inbound channel configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit TCP inbound channel custom properties.
  1. Set TCPInboundChannel custom properties as needed.

     Open an editor on a TCPInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=TCPInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:TCPInboundChannel=myTCPIC
AttributeInfo=properties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Run the applyConfigProperties command.
- If you no longer need the TCP inbound channel or an existing custom property, you can delete the entire TCP inbound channel object or the custom property.

– To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

– To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the TCP inbound channel object and its properties.

What to do next

Save the changes to your configuration.

Working with web container inbound channel properties files

You can use properties files to create, modify, or delete web container inbound channel properties and custom properties.

Before you begin

Determine the changes that you want to make to your Web container inbound channel configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a web container inbound channel object. You can also create, modify, or delete web container inbound channel custom properties.

Run administrative commands using wsadmin to create or change a properties file for a web container inbound channel, validate the properties, and apply them to your configuration.

Table 534. Actions for web container inbound channel properties files. You can create, modify, and delete web container inbound channel objects.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire WebContainerInboundChannel object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 535. Actions for web container inbound channel custom properties. You can create, modify, and delete web container inbound channel custom properties.

Action	Procedure
create	Not applicable

Table 535. Actions for web container inbound channel custom properties (continued). You can create, modify, and delete web container inbound channel custom properties.

Action	Procedure
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit a web container inbound channel properties file.
  1. Set WebContainerInboundChannel object properties as needed.

     Open an editor on a WebContainerInboundChannel properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example WebContainerInboundChannel properties file follows:

```
#
# Header
#
ResourceType=WebContainerInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:WebContainerInboundChannel=myWCC
#DELETE=true
#

#
#Properties
#
writeBufferSize=32768 #integer,required,default(8192)
enableFRCA=true #boolean,default(true)
name= myWCC #required
discriminationWeight=10 #integer,default(0)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Run the applyConfigProperties command to change a web container inbound channel configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit web container inbound channel custom properties.
  1. Set WebContainerInboundChannel custom properties as needed.

     Open an editor on a WebContainerInboundChannel properties. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=WebContainerInboundChannel
ImplementingResourceType=TransportChannelService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:TransportChannelService=:WebContainerInboundChannel=myWCC
AttributeInfo=properties(name,value)
#

#
```

```
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

2. Run the applyConfigProperties command.

- If you no longer need the web container inbound channel or an existing custom property, you can delete the entire web container inbound channel object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the web container inbound channel object and its properties.

What to do next

Save the changes to your configuration.

Working with URL provider properties files

You can use properties files to create or change uniform resource locator (URL) provider properties.

Before you begin

Determine the changes that you want to make to your URL provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a URL provider object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a URL provider, validate the properties, and apply them to your configuration.

Table 536. Actions for URL provider properties files. You can create, modify, and delete URL provider configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a URL provider and its properties.
  1. Create a properties file for a URLProvider object.

     Open an editor and create a URL provider properties file. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for a URLProvider object named `myProvider` at cell scope follows:

```
#
# Header
#
ResourceType=URLProvider
ImplementingResourceType=URLProvider
ResourceId=Cell=!{cellName}:URLProvider=myProvider
#DELETE=true
#

#
#Properties
#
streamHandlerClassName=unused
classpath={}
name=myProvider
nativepath={}
description=null
providerType=null #readonly
protocol=unused


EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

  2. Run the applyConfigProperties command to create a URL provider configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing URL provider.
  1. Obtain a properties file for the URL provider that you want to change.

     You can extract a properties file for a URLProvider object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a URL provider configuration.
- If you no longer need a URL provider, you can delete the entire URL provider object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the URL provider object.

What to do next

Save the changes to your configuration.

Working with URL properties files

You can use properties files to create or change uniform resource locator (URL) properties.

Before you begin

Determine the changes that you want to make to your URL configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a URL object and its configuration properties.

Run administrative commands using wsadmin to create or change a properties file for a URL instance, validate the properties, and apply them to your configuration.

Table 537. Actions for URL properties files. You can create, modify, and delete URL configuration properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command.
delete	Uncomment `#DELETE=true` and run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a URL and its properties.
  1. Create a properties file for a URL object.

     Open an editor and create a URL properties file. You can copy the example properties file in this step into an editor and modify the properties as needed for your situation.

     An example properties file for a URL named `myURLJndiName` under the Default URL Provider at cell scope follows:

```
#
# Header
#
ResourceType=URL
ImplementingResourceType=GenericType
ResourceId=Cell=!{cellName}:URLProvider=Default URL Provider:URL=jndiName#myURLJndiName
#DELETE=true
#

#
#Properties
#
spec=mySpec #required
name=myName #required
description=null
category=null
providerType=null
provider=Default URL Provider #ObjectName(URLProvider)
jndiName=myURLJndiName #required

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

  2. Run the applyConfigProperties command to create a URL configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing URL.
  1. Obtain a properties file for the URL that you want to change.

You can extract a properties file for a URL object using the extractConfigProperties command.

2. Open the properties file in an editor and change the properties as needed.

   Ensure that the environment variables in the properties file match your system.

3. Run the applyConfigProperties command to change a URL configuration.

- If you no longer need a URL, you can delete the entire URL object.

  To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the URL object.

What to do next

Save the changes to your configuration.

Working with service integration properties files

You can use properties files to create, modify, or delete service integration bus objects. Service integration bus is the default Java Message Service (JMS) messaging provider for the product.

Before you begin

Determine the property values that you want to set for the service integration bus configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a service integration bus object and its configuration properties.

Table 538. Actions for service integration bus properties files. You can create, change, or delete service integration bus configuration properties.

Action	Procedure
create	Specify `commandName=createSIBus` in the properties file. Run the applyConfigProperties command.
modify	Specify `commandName=modifySIBus` in the properties file. Run the applyConfigProperties command.
delete	Specify `commandName=deleteSIBus` in the properties file. Run the applyConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Procedure

1. Create modify, or delete a service integration bus object.
   - Create a service integration bus object.

     Open an editor, specify `commandName=createSIBus` in the header, specify the service integration bus properties, and save the file. You can copy the following service integration bus configuration to the properties file and edit the properties as needed.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=createSIBus
#

#
# Properties
#
secure=null #Boolean
useServerIdForMediations=null #Boolean
interEngineAuthAlias=null #String
auditAllowed=null #Boolean
discardOnDelete=null #Boolean
scriptCompatibility=null #String
mediationsAuthAlias=null #String
busSecurity=null #Boolean
highMessageThreshold=null #Long
bus=myBus #String,required
configurationReloadEnabled=null #Boolean
securityGroupCacheTimeout=null #Long
bootstrapPolicy=null #String
description=null #String
protocol=null #String
```

- Modify a service integration bus object.

  Open an editor, specify `commandName=modifySIBus` in the header, change the service integration properties as needed, and save the file.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=modifySIBus
#

#
# Properties
#
secure=null #Boolean
useServerIdForMediations=null #Boolean
interEngineAuthAlias=null #String
auditAllowed=null #Boolean
discardOnDelete=null #Boolean
mediationsAuthAlias=null #String
busSecurity=null #Boolean
highMessageThreshold=null #Long
bus=myBus #String,required
configurationReloadEnabled=null #Boolean
securityGroupCacheTimeout=null #Long
bootstrapPolicy=null #String
permittedChains=null #String
description=null #String
protocol=null #String
```

- Delete a service integration bus object.

  Open an editor, specify `commandName=deleteSIBus` in the header, specify the bus property, and save the file.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=deleteSIBus
#

#
# Properties
#
bus=myBus #String,required
```

2. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName mySIBus.props -reportFileName report.txt '])
```

   Optionally, you can use the command in interactive mode:

```
AdminTask.applyConfigProperties('-interactive')
```

Results

You can use the properties file to configure and manage the service integration object.

Example

This section contains several example properties files that create, modify, or delete service integration objects. Run the applyConfigProperties command to apply a properties file.

- Properties file that uses createSIBJMSConnectionFactory
- Properties file that uses createSIBJMSQueue
- Properties file that uses addGroupToBusConnectorRole
- Properties file that uses many commands to create and modify SIB objects

Properties file that uses the createSIBJMSConnectionFactory command

This example creates a service integration JMS connection factory:

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=createSIBJMSConnectionFactory
#

#
# Properties
#
logMissingTransactionContext=null #Boolean
password=null #String
readAhead=null #String
type=null #String
tempQueueNamePrefix=null #String
shareDurableSubscriptions=null #String
durableSubscriptionHome=null #String
targetTransportChain=null #String
authDataAlias=null #String
userName=null #String
targetSignificance=null #String
shareDataSourceWithCMP=null #Boolean
providerEndPoints=null #String
persistentMapping=null #String
nonPersistentMapping=null #String
jndiName=mySIBJndiName #String,required
clientID=null #String
targetObject=targetObject #null,required
manageCachedHandles=null #Boolean
consumerDoesNotModifyPayloadAfterGet=null #String
category=null #String
targetType=null #String
busName=myBus #String,required
description=null #String
xaRecoveryAuthAlias=null #String
containerAuthAlias=null #String
mappingAlias=null #String
producerDoesNotModifyPayloadAfterSet=null #String
tempTopicNamePrefix=null #String
connectionProximity=null #String
target=null #String
name=mySIBJmsCF #String,required
```

Properties file that uses the createSIBJMSQueue command

This example creates a service integration JMS queue:

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=createSIBJMSQueue
#

#
# Properties
#
name=mySIBJmsQ #String,required
queueName=mySIBJmsQ #String,required
producerPreferLocal=null #Boolean
jndiName=mySIBQJndiName #String,required
readAhead=null #String
busName=myBus #String
priority=null #Integer
gatherMessages=null #Boolean
scopeToLocalQP=null #Boolean
deliveryMode=null #String
description=null #String
```

```
# target object can be either configId or scope format such as cell=cellName:node=nodeName:JDBCProvider=...
targetObject=targetObject #ObjectName,required
producerBind=null #Boolean
timeToLive=null #Long
```

Properties file that uses the addGroupToBusConnectorRole command

This example adds a bus to a service integration group:

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=addGroupToBusConnectorRole
#

#
# Properties
#
uniqueName=null #String
bus=myBus #String,required
group=mySibGroup #String,required
```

Properties file that uses many commands to create and modify service integration objects

This example creates and modifies service integration objects. You can change the Environment Variables `nodeName` and `serverName` at the end of the file to match your system. Set `dontCreate` to `false` to create and modify the service integration configuration. Set `dontDelete` to `false` to delete all the created service integration configuration.

```
#
# Create SIBus
#
CreateDeleteCommandProperties=true
SKIP=!{dontCreate}
commandName=createSIBus

#
# parameters
#
bus=!{sibus} #String,required
#

#
# Modify SIBus
#
CreateDeleteCommandProperties=true
SKIP=!{dontCreate}
commandName=modifySIBus

#
# parameters
#
bus=!{sibus} #String,required
description="modified description of this bus"
busSecurity=true

#
# Add SIBus Member
#
CreateDeleteCommandProperties=true
SKIP=!{dontCreate}
commandName=addSIBusMember
#

#
# parameters
#
bus=!{sibus} #String,required
node=!{nodeName}
server=!{serverName}

#
# Modify SIBEngine
#
CreateDeleteCommandProperties=true
SKIP=!{dontCreate}
commandName=modifySIBEngine
#

#
# parameters
#
bus=!{sibus} #String,required
```

```
node=!{nodeName}
server=!{serverName}
#engine=single_engine // not required if single engine

description="message engine"
initialState=STOPPED
#

#
# Create new SIB Destination
#
CreateDeleteCommandProperties=true
SKIP=!{dontCreate}
commandName=createSIBDestination

#
# parameters
#
bus=!{sibus} #String,required
name=myQueue
type=QUEUE
node=!{nodeName}
server=!{serverName}
#

#
# Delete SIB Destination
#
CreateDeleteCommandProperties=true
SKIP=!{dontDelete}
commandName=deleteSIBDestination
#

#
# parameters
#
bus=!{sibus} #String,required
name=myQueue
#

#
# Delete SIB Engine
#
CreateDeleteCommandProperties=true
SKIP=!{dontDelete}
commandName=deleteSIBEngine

#
# parameters
#
bus=!{sibus} #String,required
node=!{nodeName}
server=!{serverName}
#engine=single_engine // not required if single engine
#

#
# Delete SIB Bus Member
#
CreateDeleteCommandProperties=true
SKIP=!{dontDelete}
commandName=removeSIBusMember
#

#
# parameters
#
bus=!{sibus} #String,required
node=!{nodeName}
server=!{serverName}

#
# Delete SIBus
#
CreateDeleteCommandProperties=true
SKIP=!{dontDelete}
commandName=deleteSIBus
#

#
# parameters
#
bus=!{sibus} #String,required
#

EnvironmentVariablesSection
#
#  Environment Variables
sibus=newSib
```

```
serverName=server1
nodeName=myNode05
dontDelete=true
dontCreate=false
```

What to do next

Save the changes to your configuration.

Working with the service integration bus member properties files

You can use properties files to add and remove service integration bus member objects. Service integration is the default Java Message Service (JMS) messaging provider for the product.

Before you begin

Determine the property values that you want to set for the service integration bus member configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can add a member to a service integration object or remove a member from a service integration object.

Table 539. Actions for the service integration member properties files. You can add or remove service integration members.

Action	Procedure
create	Specify `commandName=addSIBusMember` in the properties file. Run the applyConfigProperties command.
modify	Not applicable
delete	Specify `commandName=removeSIBusMember` in the properties file. Run the applyConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Procedure

1. Add or remove a service integration bus member object.
   - Add a member to a service integration object.

     Open an editor, specify `commandName=addSIBusMember` in the header, specify the service integration bus member properties, and save the file.

     You can copy the following service integration bus member configuration to the properties file and edit the properties as needed. Set values for a server and either a cluster or node.

     ```
     #
     # Header
     #
     CreateDeleteCommandProperties=true
     commandName=addSIBusMember
     #

     #
     #Properties
     #
     minPermanentStoreSize=null #java.lang.Long
     trustUserIds=null #Boolean
     maxTemporaryStoreSize=null #java.lang.Long
     host=null #String
     createTables=null #Boolean
     enableAssistance=null #Boolean
     temporaryStoreDirectory=null #String
     securityAuthAlias=null #String
     maxHeapSize=null #Integer
     ```

```
datasourceJndiName=null #String
bus=myBus #String,required
schemaName=null #String
dataStore=null #String
unlimitedPermanentStoreSize=null #Boolean
policyName=null #String
preferredServersOnly=null #Boolean
logSize=null #java.lang.Long
failback=null #Boolean
minTemporaryStoreSize=null #java.lang.Long
cluster=null #String
node=myNode #String
failover=null #Boolean
permanentStoreDirectory=null #String
maxPermanentStoreSize=null #java.lang.Long
server=myServer #String
wmqServer=null #String
channel=null #String
port=null #Integer
virtualQueueManagerName=null #String
unlimitedTemporaryStoreSize=null #Boolean
logDirectory=null #String
fileStore=null #String
initialHeapSize=null #Integer
transportChain=null #String
authAlias=null #String
createDefaultDatasource=null #Boolean
```

- Remove a member from the service integration object.

  Open an editor, specify `commandName=removeSIBusMember` in the header, specify the bus, server, node, and cluster properties, and then save the file.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=removeSIBusMember
#

#
#Properties
#
cluster=null #String
bus=myBus #String,required
node=myNode #String
server=myServer #String
```

2. Run the applyConfigProperties command to add or remove the service integration bus member.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName mySIBusMember.props -reportFileName report.txt '])
```

   Optionally, you can use the command in interactive mode:

```
AdminTask.applyConfigProperties('-interactive')
```

Results

You can use the properties file to configure and manage the service integration bus member.

What to do next

Save the changes to your configuration.

Working with the service integration destination properties files

You can use properties files to create, modify, or delete bus destinations for the service integration bus objects. Service integration is the default Java Message Service (JMS) messaging provider for the product.

Before you begin

Determine the property values that you want to set for the service integration destination object configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a service integration destination object and its configuration properties.

Table 540. Actions for service integration bus destination properties files. You can create, modify, or delete the service integration destination configuration properties.

Action	Procedure
create	Specify `commandName=createSIBDestination` in the properties file. Run the applyConfigProperties command.
modify	Specify `commandName=modifySIBDestination` in the properties file. Run the applyConfigProperties command.
delete	Specify `commandName=deleteSIBDestination` in the properties file. Run the applyConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Procedure

1. Create, modify, or delete a service integration destination object.
   - Create a service integration destination object.

     Open an editor, specify `commandName=createSIBDestination` in the header, specify the service integration bus destination properties, and save the file.

     You can copy the following service integration bus destination configuration to the properties file and edit the properties as needed. Also specify the server and the node or cluster.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=createSIBDestination
#

#
# Properties
#
delegateAuthorizationCheckToTarget=null #Boolean
receiveAllowed=null #String
defaultPriority=null #Integer
nonPersistentReliability=null #String
persistentReliability=null #String
type=Queue #String,required
mqRfh2Allowed=null #Boolean
aliasBus=null #String
maxReliability=null #String
receiveExclusive=null #Boolean
exceptionDestination=null #String
foreignBus=null #String
overrideOfQOSByProducerAllowed=null #String
useRFH2=null #Boolean
blockedRetryTimeout=null #java.lang.Long
wmqQueueName=null #String
topicAccessCheckRequired=null #Boolean
wmqServer=null #String
```

```
targetBus=null #String
targetName=null #String
bus=myBus #String,required
reliability=null #String
server=myServer #String
node=myNode #String
replyDestination=null #String
auditAllowed=null #Boolean
description=null #String
cluster=null #String
maintainStrictMessageOrder=null #Boolean
sendAllowed=null #String
replyDestinationBus=null #String
maxFailedDeliveries=null #Integer
name=myDest #String,required
```

- Modify a service integration bus destination object.

  Open an editor, specify `commandName=modifySIBDestination` in the header, change the service integration bus destination properties as needed, and save the file.

- Delete a service integration destination object.

  Open an editor, specify `commandName=deleteSIB destination` in the header, specify the bus property, and save the file.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=deleteSIBDestination
#

#
# Properties
#
bus=myBus #String,required
foreignBus=null #String
aliasBus=null #String
name=myDest #String,required
```

2. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

   ```
   AdminTask.applyConfigProperties(['-propertiesFileName mySIBDestination.props -reportFileName report.txt '])
   ```

   Optionally, you can use the command in interactive mode:

   ```
   AdminTask.applyConfigProperties('-interactive')
   ```

Results

You can use the properties file to configure and manage the service integration bus destination object.

What to do next

Save the changes to your configuration.

Working with SIB engine properties files

You can use properties files to create, modify, or delete a service integration bus (SIBus) messaging engine. SIBus is the default Java Message Service (JMS) messaging provider for the product.

Before you begin

Determine the property values that you want to set for the SIB engine configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a SIB engine and its configuration properties.

Table 541. Actions for SIB engine properties files. You can create, change, or delete SIB engine configuration properties.

Action	Procedure
create	Specify `commandName=createSIBusEngine` in the properties file. Run the applyConfigProperties command.
modify	Specify `commandName=modifySIBEngine` in the properties file. Run the applyConfigProperties command.
delete	Specify `commandName=deleteSIBEngine` in the properties file. Run the applyConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Procedure

1. Create modify, or delete a SIB engine.
   - Create a SIB engine.

     Open an editor, specify `commandName=createSIBEngine` in the header, specify the SIB engine properties, and save the file.

     You can copy the following SIB engine configuration to the properties file and edit the properties as needed.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=createSIBEngine
#

#
# Properties
#
initialState=null #String
node=myNode #String
defaultBlockedRetryTimeout=null #java.lang.Long
server=myServer #String
highMessageThreshold=null #java.lang.Long
bus=myBus #String,required
engine=null #String
cluster=null #String
description=null #String
```

   - Modify a SIB engine.

     Open an editor, specify `commandName=modifySIBEngine` in the header, change the SIB engine properties as needed, and save the file. Set the required property to be modified. Also set cluster or node and server properties. The engine property is not required if it is a single engine.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=modifySIBEngine
#

#
# Properties
#
initialState=null #String
node=myNode #String
defaultBlockedRetryTimeout=null #java.lang.Long
server=myServer #String
highMessageThreshold=null #java.lang.Long
bus=myBus #String,required
engine=null #String
cluster=null #String
description=null #String
```

   - Delete a SIB engine object.

Open an editor, specify `commandName=deleteSIBEngine` in the header, specify the bus and any other required properties, specify cluster or node and server properties, and then save the file.

```
#
# Header
#
CreateDeleteCommandProperties=true
commandName=deleteSIBEngine
#

#
# Properties
#
cluster=null #String
bus=myBus #String,required
engine=null #String
node=myNode #String
server=myServer #String
```

2. Run the applyConfigProperties command.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName mySIBEngine.props -reportFileName report.txt '])
```

   Optionally, you can use the command in interactive mode:

```
AdminTask.applyConfigProperties('-interactive')
```

Results

You can use the properties file to configure and manage the SIB engine.

What to do next

Save the changes to your configuration.

Working with timer manager provider properties files

You can use properties files to create, modify, or delete timer manager provider properties and custom properties.

Before you begin

Determine the changes that you want to make to your timer manager provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a timer manager provider object. You can also create, modify, or delete timer manager provider custom properties.

Run administrative commands using wsadmin to create or change a properties file for a timer manager provider, validate the properties, and apply them to your configuration.

Table 542. Actions for timer manager provider properties files. You can create, modify, and delete timer manager provider properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.

Table 542. Actions for timer manager provider properties files (continued). You can create, modify, and delete timer manager provider properties.

Action	Procedure
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire TimerManagerProvider object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a timer manager provider properties file.
  1. Set TimerManagerProvider properties as needed.

     Open an editor on a TimerManagerProvider properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example TimerManagerProvider properties file follows:

```
#
# Header
#
ResourceType=TimerManagerProvider
ImplementingResourceType=TimerManagerProvider
ResourceId=Cell=!{cellName}:TimerManagerProvider=myTimerManagerProvider
#DELETE=true
#

#
#Properties
#
classpath={}
name=myTimerManagerProvider #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Default TimerManager Provider
providerType=null #readonly
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to create or change a timer manager provider configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the timer manager provider that you want to change.

     You can extract a properties file for a TimerManagerProvider object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the timer manager provider or an existing custom property, you can delete the entire timer manager provider object or the custom property.
  – To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

– To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the timer manager provider object and its properties.

What to do next

Save the changes to your configuration.

Working with timer manager information properties files

You can use properties files to create, modify, or delete timer manager information properties and custom properties.

Before you begin

Determine the changes that you want to make to your timer manager information configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a timer manager information object. You can also create, modify, or delete timer manager information custom properties.

Run administrative commands using wsadmin to create or change a properties file for a timer manager information, validate the properties, and apply them to your configuration.

Table 543. Actions for timer manager information properties files. You can create, modify, and delete timer manager information properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire TimerManagerInfo object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create timer manager information properties.
  1. Create a TimerManagerInfo properties file.

Open an editor and set TimerManagerInfo properties. You can copy the following example TimerManagerInfo properties file into an editor and modify the properties as needed for your situation. Modify the Environment Variables section to match your system and set any property value that needs to be changed.

```
#
# Header
#
ResourceType=TimerManagerInfo
ImplementingResourceType=TimerManagerProvider
ResourceId=Cell=!{cellName}:TimerManagerProvider=myTimerManagerProvider:TimerManagerInfo=jndiName#myTimerManagerJndiName
#DELETE=true
#

#
#Properties
#
referenceable=null
name=myTimerManager #required
category=null
defTranClass=null
providerType=null #readonly
numAlarmThreads=2 #integer,required,default(2)
jndiName=myTimerManagerJndiName #required
serviceNames={}
#provider=TimerManagerProvider#ObjectName(TimerManagerProvider),readonly
description=WebSphere Default TimerManager
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create or change a timer manager information configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for a TimerManagerInfo object that you want to change.

     You can extract a properties file for a TimerManagerInfo object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the timer manager information or an existing custom property, you can delete the entire timer manager information object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the timer manager information object and its properties.

What to do next

Save the changes to your configuration.

Working with timer manager information J2EE resource properties files:

You can use properties files to create, modify, or delete timer manager information Java 2 Platform, Enterprise Edition (J2EE) resource custom properties.

Before you begin

Determine the changes that you want to make to your timer manager information J2EE resource configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete timer manager information J2EE resource custom properties.

Run administrative commands using wsadmin to extract a properties file for a timer manager information J2EE resource, validate the properties, and apply them to your configuration.

Table 544. Actions for timer manager information J2EE resource properties. You can create, modify, and delete timer manager information J2EE resource custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create timer manager information J2EE resource properties.
  1. Specify TimerManagerInfo J2EEResourcePropertySet custom properties in a properties file.

     Open an editor and specify timer manager information J2EE resource properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=TimerManagerProvider
ResourceId=Cell=!{cellName}:TimerManagerProvider=myTimerManagerProvider:TimerManagerInfo=jndiName
#myTimerManagerJndiName:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
```

```
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create or change a timer manager information J2EE resource configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing timer manager information J2EE resource properties.
  1. Obtain a properties file for the timer manager information J2EE resource that you want to change.

     You can extract a properties file for a TimerManagerInfo J2EEResourcePropertySet using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete timer manager information J2EE resource custom properties.

  To delete one or more properties, specify only the properties to delete in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the timer manager information J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with variable map properties files

You can use properties files to create, modify, or delete property-value pairs in variable maps.

Before you begin

Determine the changes that you want to make to your variable map.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete one or more variable substitution entries in a variable map.

Run administrative commands using wsadmin to create or change a properties file for a variable map, validate the variable substitution entries, and apply them to your configuration.

Table 545. Actions for variable map properties. You can create, modify, and delete properties and values in variable maps.

Action	Procedure
create	Not applicable

Table 545. Actions for variable map properties (continued). You can create, modify, and delete properties and values in variable maps.

Action	Procedure
modify	Edit property values in the variable map properties file and then run the applyConfigProperties command to modify the values of existing variables.
delete	Not applicable
create Property	Edit the variable map properties file and then run the applyConfigProperties command to create a variable substitution entry.
delete Property	Run the deleteConfigProperties command to delete an existing variable substitution entry. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify an existing properties file.
  1. Obtain a properties file for the variable map that you want to change.

     You can extract a properties file for a VariableMap object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a variable substitution entry, edit the `AttributeInfo` value and properties values. An example VariableMap properties file follows:

```
#
# Header
#
ResourceType=VariableMap
ImplementingResourceType=VariableMap
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:VariableMap=
AttributeInfo=entries(symbolicName,value)

#
#Properties
#
SERVER_LOG_ROOT=${LOG_ROOT}/!{serverName}
WAS_SERVER_NAME=!{serverName}

#
EnvironmentVariablesSection
#
#Environment Variables
cellName=myCell04
nodeName=myNode
serverName=myServer
```

  3. Run the applyConfigProperties command to create or change a VariableMap configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need a variable substitution entry, you can delete the entry.

  To delete a variable substitution entry, specify only the entry to be deleted in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the variable map properties.

What to do next

Save the changes to your configuration.

Working with virtual host properties files

You can use properties files to create or change virtual host properties.

Before you begin

Determine the changes that you want to make to your virtual host configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a virtual host object. You can also work with host aliases and mime entries of a virtual host.

Run administrative commands using wsadmin to create or change a properties file for a virtual host, validate the properties, and apply them to your configuration.

Table 546. Actions for virtual host properties files. You can create, modify, and delete virtual host configuration properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit required properties and then run the applyConfigProperties command.
delete	To delete an entire virtual host object, uncomment `#DELETE=true` and run the deleteConfigProperties command.
create Property	To add a host alias, add an entry such as the example `9999=*` alias to the Host Alias section and then run the applyConfigProperties command. To add a mime type, add an entry such as the example `newMime={a,b,c}` to the Mime Types section and then run the applyConfigProperties command.
delete Property	To delete an existing host alias, list only that alias in the Properties section of the properties file and then run the deleteConfigProperties command. For example, to delete the example `9999=*` alias, keep only that `9999=*` alias and remove other properties from the Host Alias section and then run deleteConfigProperties.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a virtual host and its properties.
  1. Create a properties file for a VirtualHost object.

     Open an editor and create a virtual host properties file. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     The following example defines a virtual host named `myHost` at the cell scope:

```
#
# Header
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=myHost
#DELETE=true
```

```
#

#
#Properties
#
name=myHost

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

The following example defines a virtual host named `myVh`, mime types, and host aliases:

```
#
# Header
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=myVh
#DELETE=true
#

#
#Properties
#
name=myVh #required


#
# Header MimeTypes section
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=myVh
AttributeInfo=mimeTypes(type,extensions)
#

#
#Properties
#
video/x-sgi-movie={movie}
application/x-csh={csh}
text/richtext={rtx}
image/tiff={tif,tiff}
application/x-bsh={bsh}
application/x-tcl={tcl}
application/drafting={DRW}
application/pdf={pdf}
application/SLA={STL,stl}
audio/x-wav={wav}
video/mpeg={MPE,MPEG,MPG,mpe,mpeg,mpg}
newMime={a,b,c}
...

#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=myVh
AttributeInfo=aliases(port,hostname)
#

#
#Properties
#
80=*
9080=*
9453=*
9096=*
9092=*
443=*
9999=*


EnvironmentVariablesSection

#Environment Variables
cellName=myCell
```

2. Run the applyConfigProperties command to create a virtual host configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the virtual host that you want to change.

You can extract a properties file for a VirtualHost object using the extractConfigProperties command.

2. Open the properties file in an editor and change the properties as needed.

   Ensure that the environment variables in the properties file match your system.

3. Run the applyConfigProperties command to change a virtual host configuration.

- If you no longer need the virtual host or an existing property, you can delete the entire virtual host object or property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete an alias or mime entry, specify only the alias or mime entry to be deleted in the host alias Properties section of the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the virtual host object.

What to do next

Save the changes to your configuration.

Working with host alias properties

You can use virtual host properties files to create or change host alias properties.

Before you begin

Determine the changes that you want to make to your host alias conifguration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a virtual host properties file, you can create, modify, or delete a host alias. Run administrative commands using wsadmin to configure or delete a host alias.

Table 547. Actions for host alias properties. You can create, modify, and delete host alias properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.
modify	Edit the properties and then run the applyConfigProperties command.
delete	Specify the host aliases to delete in the Properties section and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a host alias.
  1. Edit a VirtualHost properties file so that it specifies a host alias property.

     Open an editor on a virtual host properties file. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     The following example defines a host alias inside a virtual host named `default_host` at the cell scope with host name `newHost` and port `5555`:

```
#
# Header
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=default_host
AttributeInfo=aliases(port,hostname)
#

#
#Properties
#
5555=newHost

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

2. Run the applyConfigProperties command to create a host alias.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing host alias.
  1. Obtain a virtual host properties file that defines the host alias that you want to change.

     You can extract a properties file for a VirtualHost object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the host alias properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a host alias.
- Delete an existing host alias.

  If you no longer need one or more host aliases, list the host aliases that you want removed in the Properties section of the virtual host properties file and then run the deleteConfigProperties command:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the host alias.

What to do next

Save the changes to your configuration.

Working with mime entry properties

You can use virtual host properties files to create or change mime entry properties.

Before you begin

Determine the changes that you want to make to your mime entry configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a virtual host properties file, you can create, modify, or delete a mime entry. Run administrative commands using wsadmin to configure or delete a mime entry.

Table 548. Actions for mime entry properties. You can create, modify, and delete mime entry properties.

Action	Procedure
create	Set properties and then run the applyConfigProperties command.

Table 548. Actions for mime entry properties (continued). You can create, modify, and delete mime entry properties.

Action	Procedure
modify	Edit the properties and then run the applyConfigProperties command.
delete	Specify the mime entries to delete in the Properties section and then run the deleteConfigProperties command.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a mime entry.
  1. Edit a VirtualHost properties file so that it specifies a mime entry property.

     Open an editor on a virtual host properties file. You can copy an example properties file in this step into an editor and modify the properties as needed for your situation.

     The following example defines a mime entry inside a virtual host named `default_host` at the cell scope with type `myType` and extension `myExt`:

```
#
# Header
#
ResourceType=VirtualHost
ImplementingResourceType=VirtualHost
ResourceId=Cell=!{cellName}:VirtualHost=default_host
AttributeInfo=mimeTypes(type,extensions)
#

#
#Properties
#
myType={myExt}

EnvironmentVariablesSection
#
#
#Environment Variables
cellName=WASCell06
```

  2. Run the applyConfigProperties command to create a mime entry.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing mime entry.
  1. Obtain a virtual host properties file that defines the mime entry that you want to change.

     You can extract a properties file for a VirtualHost object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the mime entry properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command to change a mime entry.
- Delete an existing mime entry.

  If you no longer need one or more mime entries, list the mime entries that you want removed in the Properties section of the virtual host properties file and then run the deleteConfigProperties command:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the mime entry.

What to do next

Save the changes to your configuration.

Working with web server properties files

You can use properties files to create, modify, or delete web server properties.

Before you begin

Determine the changes that you want to make to your Web server configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create a web server instance. You can also modify or delete web server properties.

Run administrative commands using wsadmin to extract a properties file for a web server, validate the properties, and apply them to your configuration.

Table 549. Actions for web server properties files. You can create web server instances and modify or delete web server properties.

Action	Procedure
create	1. Create a properties file that specifies the following commands in the header and sets values for web server properties: `CreateDeleteCommandProperties=true` `SKIP=false` `commandName=createWebServer` 2. Run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create a properties file for a WebServer instance.
  1. Open an editor and create a properties file that specifies commands to create a WebServer instance and its properties.

     For example, create a properties file such as the following for a WebServer instance:

```
#
# Header
#
CreateDeleteCommandProperties=true
SKIP=false
commandName=createWebServer
#

#
#Properties
#
name=IHS #String,required
templateLocation=null #javax.management.ObjectName
```

```
genUniquePorts=null #Boolean
bitmode=null #String
specificShortName=null #String
clusterName=null #String
targetObject=targetObject #null,required
templateName=null #String
genericShortName=null #String


#
# Step parameters
#
CreateDeleteCommandProperties=true
SKIP=true
stepName=serverConfig
#

#
#Properties
#
webAppMapping=null #String
pluginInstallRoot=null #String
webProtocol=null #String
webPort=null #Integer
configurationFile=null #String
serviceName=null #String
errorLogfile=null #String
webInstallRoot=null #String
accessLogfile=null #String


#
# Step parameters
#
CreateDeleteCommandProperties=true
SKIP=true
stepName=remoteServerConfig
#

#
#Properties
#
adminPort=null #Integer
adminProtocol=null #String
adminPasswd=null #String
adminUserID=null #String
```

The properties file sets the CreateDeleteCommandProperties command to `true` in the header and steps. It creates a web server named `IHS` and, in the steps, configures a server and a remote server.

2. Run the applyConfigProperties command to create a Web server configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify the properties file.
  1. Open the properties file in an editor and change the properties as needed. Ensure that the environment variables in the properties file match your system.

     To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#

ResourceType=WebServer
ImplementingResourceType=WebServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WebServer=
AttributeInfo=components
#

#
#Properties
#
name=IHS
webserverAdminProtocol=HTTP #ENUM(HTTPS|HTTP),default(HTTP)
logFilenameError="${WEB_INSTALL_ROOT}/logs/error.log"
configurationFilename="${WEB_INSTALL_ROOT}/conf/httpd.conf"
server=IHS#ObjectName(Server)
logFilenameAccess="${WEB_INSTALL_ROOT}/logs/access.log"
parentComponent=null
webserverProtocol=HTTP #ENUM(HTTPS|HTTP),default(HTTP)
serviceName=IBMHTTPServer7.0
webserverInstallRoot="C:\Program Files\IBM\HTTPServer"
```

```
webserverType=IHS #ENUM(HTTP_SERVER|IIS|DOMINO|APACHE|SUNJAVASYSTEM|HTTPSERVER_ZOS|IHS),default(IHS)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

2. Run the applyConfigProperties command to change a Web server configuration.

   Running the applyConfigProperties command applies the properties file to the configuration.

- Delete web server properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the web server instance and its properties.

What to do next

Save the changes to your configuration.

Working with plug-in properties files

You can use properties files to create, modify, or delete web server plug-in properties and custom properties.

Before you begin

Determine the changes that you want to make to your plug-in properties configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a plug-in object. You can also create, modify, or delete plug-in custom properties.

Run administrative commands using wsadmin to change a properties file for a plug-in, validate the properties, and apply them to your configuration.

Table 550. Actions for plug-in properties files. You can modify and delete plug-in objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Table 551. Actions for plug-in custom properties. You can create, modify, and delete plug-in custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit a web server plug-in properties file.
  1. Set PluginProperties object properties as needed.

     Open an editor on a PluginProperties properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example PluginProperties properties file follows:

```
#
# Header
#
ResourceType=PluginProperties
ImplementingResourceType=WebServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WebServer=:PluginProperties=
AttributeInfo=pluginProperties
#

#
#Properties
#
LogFilename="c:\Program Files\IBM\HTTPServer\Plugins\logs\IHS\http_plugin.log"
ConfigFilename=plugin-cfg.xml #default(plugin-cfg.xml)
RemoteKeyRingFilename="c:\Program Files\IBM\HTTPServer\Plugins\config\IHS\plugin-key.kdb"
LogLevel=ERROR #ENUM(DETAIL|DEBUG|ERROR|WARN|STATS|TRACE),default(ERROR)
PluginPropagation=AUTOMATIC #ENUM(MANUAL|AUTOMATIC),default(AUTOMATIC)
AcceptAllContent=false #boolean,default(false)
PluginInstallRoot="c:\Program Files\IBM\HTTPServer\Plugins"
IgnoreDNSFailures=false #boolean,default(false)
ESIInvalidationMonitor=false #boolean,default(false)
IISDisableNagle=false #boolean,default(false)
RemoteConfigFilename="c:\Program Files\IBM\HTTPServer\Plugins\config\IHS\plugin-cfg.xml"
PluginGeneration=AUTOMATIC #ENUM(MANUAL|AUTOMATIC),default(AUTOMATIC)
ResponseChunkSize=64 #integer,default(64)
RefreshInterval=60 #integer,default(60)
VHostMatchingCompat=false #boolean,default(false)
ASDisableNagle=false #boolean,default(false)
IISPluginPriority=HIGH #ENUM(MEDIUM|HIGH|LOW),default(HIGH)
KeyRingFilename=plugin-key.kdb #default(plugin-key.kdb)
ChunkedResponse=false #boolean,default(false)
ESIEnable=true #boolean,default(true)
ESIMaxCacheSize=1024 #integer,default(1024)
AppServerPortPreference=HOSTHEADER #ENUM(WEBSERVERPORT|HOSTHEADER),default(WEBSERVERPORT)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

  2. Run the applyConfigProperties command to create or change a plug-in properties configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit web server plug-in custom properties.

1. Set PluginProperties custom properties as needed.

   Open an editor on a PluginProperties properties. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=PluginProperties
ImplementingResourceType=WebServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WebServer=:PluginProperties=
AttributeInfo=properties(name,value)

#
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

   2. Run the applyConfigProperties command.

- Delete a web server plug-in property.
  - To delete a property, run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to delete in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the plug-in object and its properties.

What to do next

Save the changes to your configuration.

Working with plug-in server cluster properties files

You can use properties files to modify or delete plug-in server cluster properties of web servers.

Before you begin

Determine the changes that you want to make to your plug-in server cluster properties configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a plug-in server cluster object.

Run administrative commands using wsadmin to change a properties file for a plug-in server cluster, validate the properties, and apply them to your configuration.

Table 552. Actions for plug-in server cluster properties files. You can modify or delete plug-in server cluster objects.

Action	Procedure
create	Not applicable

Table 552. Actions for plug-in server cluster properties files (continued). You can modify or delete plug-in server cluster objects.

Action	Procedure
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify existing plug-in server cluster properties.
  1. Obtain a properties file for the plug-in server cluster that you want to change.

     You can extract a PluginServerClusterProperties properties file using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Open an editor and specify plug-in server cluster custom properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values. Ensure that the environment variables in the properties file match your system.

```
#
# Header
#
ResourceType=PluginServerClusterProperties
ImplementingResourceType=WebServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WebServer=:PluginProperties=:PluginServerClusterProperties=
AttributeInfo=pluginServerClusterProperties
#

#
#Properties
#
RetryInterval=60 #integer,default(60)
PostSizeLimit=-1 #integer,default(-1)
PostBufferSize=64 #integer,default(64)
LoadBalance=ROUND_ROBIN #ENUM(ROUND_ROBIN|RANDOM),default(ROUND_ROBIN)
CloneSeparatorChange=false #boolean,default(false)
RemoveSpecialHeaders=true #boolean,default(true)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

  3. Run the applyConfigProperties command to change a plug-in server cluster properties configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt'])
```

- Delete plug-in server cluster properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the plug-in server cluster object and its properties.

What to do next

Save the changes to your configuration.

Working with key store file properties files

You can use properties files to modify or delete key store file properties of web servers.

Before you begin

Determine the changes that you want to make to your key store file properties configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a key store file object.

Run administrative commands using wsadmin to change a properties file for a key store file, validate the properties, and apply them to your configuration.

Table 553. Actions for key store file properties files. You can modify or delete key store file objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify existing key store file properties.
  1. Obtain a properties file for the key store file that you want to change.

     You can extract a KeyStoreFile properties file using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Open an editor and specify key store file custom properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values. Ensure that the environment variables in the properties file match your system.

```
#
# Header
#
ResourceType=KeyStoreFile
```

```
ImplementingResourceType=WebServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WebServer=:KeyStoreFile=name#keyFile,directory#keys
AttributeInfo=keyStoreFiles
#

#
#Properties
#
directory=keys
name=keyFile


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

3. Run the applyConfigProperties command to change a key store file properties configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt'])
```

- Delete key store file properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the key store file object and its properties.

What to do next

Save the changes to your configuration.

Working with administrative server authentication properties files

You can use properties files to modify or delete administrative server authentication properties of web servers.

Before you begin

Determine the changes that you want to make to your administrative server authentication properties configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete an administrative server authentication object.

Run administrative commands using wsadmin to change a properties file for an administrative server authentication, validate the properties, and apply them to your configuration.

Table 554. Actions for administrative server authentication properties files. You can modify or delete administrative server authentication objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.

Table 554. Actions for administrative server authentication properties files (continued). You can modify or delete administrative server authentication objects.

Action	Procedure
delete	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify existing administrative server authentication properties.
  1. Obtain a properties file for the administrative server authentication that you want to change.

     You can extract an AdminServerAuthentication properties file using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Open an editor and specify administrative server authentication custom properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values. Ensure that the environment variables in the properties file match your system.

```
#
# Header
#
ResourceType=AdminServerAuthentication
ImplementingResourceType=WebServer
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WebServer=:AdminServerAuthentication=userid#user1
AttributeInfo=adminServerAuthentication
#

#
#Properties
#
userid=user1
password=passwd

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

  3. Run the applyConfigProperties command to change an administrative server authentication properties configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt'])
```

- Delete administrative server authentication properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the administrative server authentication object and its properties.

What to do next

Save the changes to your configuration.

Working with web server process definition properties files

You can use properties files to modify or delete process definition properties of web servers.

Before you begin

Determine the changes that you want to make to your process definition properties configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a process definition object.

Run administrative commands using wsadmin to change a properties file for a process definition, validate the properties, and apply them to your configuration.

Table 555. Actions for process definition properties files. You can modify or delete process definition objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify existing process definition properties.
  1. Obtain a properties file for the web server JavaProcessDef that you want to change.

     You can extract a JavaProcessDef properties file using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Open an editor and specify JavaProcessDef custom properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values. Ensure that the environment variables in the properties file match your system.

```
#
# Header
#
ResourceType=JavaProcessDef
ImplementingResourceType=Server
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=
AttributeInfo=processDefinitions
#

#
#Properties
#
```

```
executableTarget=null
executableName=apache.exe
stopCommand="${WEB_INSTALL_ROOT}/bin/apache.exe"
stopCommandArgs={-k,stop,-n,IBMHTTPServer7.0,-f,${WEB_INSTALL_ROOT}/conf/httpd.conf}
terminateCommand=null
workingDirectory="${WEB_INSTALL_ROOT}" #required
startCommandArgs={-k,start,-n,IBMHTTPServer7.0,-f,${WEB_INSTALL_ROOT}/conf/httpd.conf}
executableArguments={}
startCommand="${WEB_INSTALL_ROOT}/bin/apache.exe"
executableTargetKind=JAVA_CLASS #ENUM(EXECUTABLE_JAR|JAVA_CLASS),default(JAVA_CLASS)
terminateCommandArgs={}
processType=null

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

3. Run the applyConfigProperties command to change a process definition properties configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt'])
```

- Delete process definition properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the process definition object and its properties.

What to do next

Save the changes to your configuration.

Working with web server JVM properties files

You can use properties files to modify or delete Java virtual machine (JVM) properties of web servers.

Before you begin

Determine the changes that you want to make to your JVM properties configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a JVM object.

Run administrative commands using wsadmin to change a properties file for a JVM, validate the properties, and apply them to your configuration.

Table 556. Actions for JVM properties files. You can modify or delete JVM objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.

Table 556. Actions for JVM properties files (continued). You can modify or delete JVM objects.

Action	Procedure
delete	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Modify existing web server JVM properties.
  1. Obtain a properties file for the JVM that you want to change.

     You can extract a JavaVirtualMachine properties file using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Open an editor and specify JVM custom properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values. Ensure that the environment variables in the properties file match your system.

```
#
# Header
#
ResourceType=JavaVirtualMachine
ImplementingResourceType=Server
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:JavaVirtualMachine=
AttributeInfo=jvmEntries

#
#Properties
#
internalClassAccessMode=ALLOW #ENUM(ALLOW|RESTRICT),default(ALLOW)
JavaHome="C:\cf50922.30\test/java" #readonly
debugArgs=
classpath={}
initialHeapSize=0 #integer,default(0)
runHProf=false #boolean,default(false)
genericJvmArguments=
hprofArguments=
osName=null
bootClasspath={}
verboseModeJNI=false #boolean,default(false)
maximumHeapSize=0 #integer,default(0)
disableJIT=false #boolean,default(false)
verboseModeGarbageCollection=false #boolean,default(false)
executableJarFileName=null
verboseModeClass=false #boolean,default(false)
debugMode=false #boolean,default(false)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

  3. Run the applyConfigProperties command to change a JVM properties configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt'])
```

- Delete web server JVM properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the JVM object and its properties.

What to do next

Save the changes to your configuration.

Working with web server JVM system properties files

You can use properties files to create, modify, or delete web server Java virtual machine (JVM) system custom properties.

Before you begin

Determine the changes that you want to make to your JVM system configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete JVM system custom properties.

Run administrative commands using wsadmin to change a properties file for a JVM system, validate the properties, and apply them to your configuration.

Table 557. Actions for JVM system properties. You can create, modify, and delete JVM system custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create web server JVM system properties.
  1. Specify JavaVirtualMachine custom properties in a properties file.

     Open an editor and specify JVM system properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
ResourceType=JavaVirtualMachine
ImplementingResourceType=Server
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:JavaProcessDef=:JavaVirtualMachine=
AttributeInfo=systemProperties(name,value)

#
#Properties
#
```

```
existingProp=newValue
newProp=value

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myNode04Cell
nodeName=myNode04
serverName=IHS
```

2. Run the applyConfigProperties command to create a JavaVirtualMachine configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing web server JVM system properties.
  1. Obtain a properties file for the JVM system that you want to change.

     You can extract a properties file for a JavaVirtualMachine using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- Delete the JVM system properties.

  To delete one or more properties, specify only those properties to delete in the properties file and run deleteConfigProperties.

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the JVM system properties.

What to do next

Save the changes to your configuration.

Working with work area service properties files

You can use properties files to change work area service configuration objects and custom properties.

Before you begin

Determine the changes that you want to make to your work area service configuration objects or custom properties.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can modify or delete a work area service object. You can also create, modify, or delete work area service custom properties.

Run administrative commands using wsadmin to change a properties file for a work area service, validate the properties, and apply them to your configuration.

Table 558. Actions for work area service configuration objects. You can modify and delete work area service objects.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command.
delete	Run the deleteConfigProperties command. Deleting a property sets the default value, if there is a default value for the property
create Property	Not applicable
delete Property	Not applicable

Table 559. Actions for work area service custom properties. You can create, modify, and delete work area service custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Edit a work area service properties file.
  1. Set WorkAreaService object properties as needed.

     Open an editor on a WorkAreaService properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example WorkAreaService properties file follows:

```
#
# Header
#
ResourceType=WorkAreaService
ImplementingResourceType=PMEServerExtension
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:PMEServerExtension=:WorkAreaService=
AttributeInfo=workAreaService
#

#
#Properties
#
maxReceiveSize=10000 #integer,required,default(32768)
enable=false #boolean,default(false)
context=null
maxSendSize=10000 #integer,required,default(32768)
enableWebServicePropagation=false #boolean,default(false)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Set WorkAreaService custom properties as needed.

     To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=WorkAreaService
ImplementingResourceType=PMEServerExtension
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:PMEServerExtension=:WorkAreaService=
AttributeInfo=properties(name,value)
#

#
#Properties
#
existingProp=value
newProp=newValue

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

3. Run the applyConfigProperties command to change a work area service configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- If you no longer need the work area service or an existing custom property, you can delete the entire work area service object or the custom property.
  - To delete the entire object, run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to manage the work area service object and its properties.

What to do next

Save the changes to your configuration.

Working with work area partition service properties files

You can use properties files to create, modify, or delete work area partition service properties and custom properties.

Before you begin

Determine the changes that you want to make to your work area partition service configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a work area partition service object. You can also create, modify, or delete work area partition service custom properties.

Run administrative commands using wsadmin to create or change properties for a work area partition service, validate the properties, and apply them to your configuration.

Table 560. Actions for work area partition service properties files. You can create, modify, and delete work area partition service objects.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire WorkAreaPartition object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Table 561. Actions for work area partition service custom properties. You can create, modify, and delete work area partition service custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create or edit a work area partition service properties file.
  1. Set WorkAreaPartitionService object properties as needed.

     Open an editor on a WorkAreaPartitionService properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example WorkAreaPartitionService properties file follows:

```
#
# Header. Required to create WorkAreaPartitionService before creating WorkAreaPartition. Parent should exist first.
#
ResourceType=WorkAreaPartitionService
ImplementingResourceType=WorkAreaPartitionService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WorkAreaPartitionService=
AttributeInfo=services
#

#
#Properties
#
enable=null


#
# Header
#
ResourceType=WorkAreaPartition
ImplementingResourceType=WorkAreaPartitionService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WorkAreaPartitionService=:WorkAreaPartition=myWap
DELETE=false
AttributeInfo=partitions
#

#
#Properties
```

```
#
maxReceiveSize=32768 #integer,required,default(32768)
name=myWap #required
enable=true #boolean,default(true)
deferredAttributeSerialization=false #boolean,default(false)
description=null
maxSendSize=32768 #integer,required,default(32768)
bidirectional=false #boolean,default(false)
enableWebServicePropagation=false #boolean,default(false)

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

2. Run the applyConfigProperties command to change a work area partition service configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Create or edit work area partition custom properties.
  1. Set WorkAreaPartition custom properties as needed.

     Open an editor on a WorkAreaPartition properties file that has `ImplementingResourceType=WorkAreaPartitionService`. Modify the Environment Variables section to match your system and set any property value that needs to be changed. To specify a custom property, edit the `AttributeInfo` value and properties values; for example:

```
#
# Header
#
ResourceType=WorkAreaPartition
ImplementingResourceType=WorkAreaPartitionService
ResourceId=Cell=!{cellName}:Node=!{nodeName}:Server=!{serverName}:WorkAreaPartitionService=:WorkAreaPartition=myWap
AttributeInfo=properties(name,value)
#

#
#Properties
#
existingProp=value
newProp=newValue

#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
serverName=server1
nodeName=myNode05
```

  2. Run the applyConfigProperties command.
- If you no longer need the work area partition service or an existing custom property, you can delete the entire work area partition service object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the work area partition service object and its properties.

What to do next

Save the changes to your configuration.

Working with work manager provider properties files

You can use properties files to create, modify, or delete work manager provider properties and custom properties.

Before you begin

Determine the changes that you want to make to your work manager provider configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a work manager provider object. You can also create, modify, or delete work manager provider custom properties.

Run administrative commands using wsadmin to create or change a properties file for a work manager provider, validate the properties, and apply them to your configuration.

Table 562. Actions for work manager provider properties files. You can create, modify, and delete work manager provider properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire WorkManagerProvider object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

`AdminTask.`*`command_name`*`('-interactive')`

Procedure

- Create a properties file for a work manager provider.
  1. Set WorkManagerProvider properties as needed.

     Open an editor on a WorkManagerProvider properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example WorkManagerProvider properties file follows:

```
#
# Header
#
ResourceType=WorkManagerProvider
ImplementingResourceType=WorkManagerProvider
ResourceId=Cell=!{cellName}:WorkManagerProvider=myWorkManagerProvider
#DELETE=true
#

#
#Properties
#
classpath={}
name=myWorkManagerProvider #required
isolatedClassLoader=false #boolean,default(false)
nativepath={}
description=Default WorkManager Provider
```

```
providerType=null #readonly
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

2. Run the applyConfigProperties command to create or change a work manager provider configuration.

   Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the work manager provider that you want to change.

     You can extract a properties file for a WorkManagerProvider object using the extractConfigProperties command.
  2. Open the properties file in an editor and change the properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need the work manager provider or an existing custom property, you can delete the entire work manager provider object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the work manager provider object and its properties.

What to do next

Save the changes to your configuration.

Working with work manager information properties files

You can use properties files to create, modify, or delete work manager information properties and custom properties.

Before you begin

Determine the changes that you want to make to your work manager information configuration or its configuration objects.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete a work manager information object. You can also create, modify, or delete work manager information custom properties.

Run administrative commands using wsadmin to create or change a properties file for a work manager information, validate the properties, and apply them to your configuration.

Table 563. Actions for work manager information properties files. You can create, modify, and delete work manager information properties.

Action	Procedure
create	Set required properties and then run the applyConfigProperties command.
modify	Edit properties and then run the applyConfigProperties command to modify the value of a custom property.
delete	Run the deleteConfigProperties command to delete a property. If the deleted property has a default value, the property is set to the default value. To delete the entire WorkManagerInfo object, uncomment `#DELETE=true` and then run the deleteConfigProperties command.
create Property	Not applicable
delete Property	Not applicable

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create a work manager information properties file.
  1. Set WorkManagerInfo properties as needed.

     Open an editor on a WorkManagerInfo properties file. Modify the Environment Variables section to match your system and set any property value that needs to be changed. An example WorkManagerInfo properties file follows:

```
#
# Header
#
ResourceType=WorkManagerInfo
ImplementingResourceType=WorkManagerProvider
ResourceId=Cell=!{cellName}:WorkManagerProvider=myWorkManagerProvider:WorkManagerInfo=jndiName#myWorkManagerJndiName
#DELETE=true
#

#
#Properties
#
isDistributable=false #boolean,default(false)
daemonTranClass=null
providerType=null #readonly
threadPriority=5 #integer,required,default(5)
workReqQSize=0 #integer,default(0)
minThreads=1 #integer,required,default(0)
jndiName=myWorkManagerJndiName #required
maxThreads=10 #integer,required,default(2)
isGrowable=false #boolean,default(true)
category=Default
description=WebSphere Default WorkManager
serviceNames={AppProfileService,com.ibm.ws.i18n,security,UserWorkArea,zos.wlm}
defTranClass=null
workTimeout=0 #integer,default(0)
#provider=WorkManagerProvider#ObjectName(WorkManagerProvider),readonly
referenceable=null
numAlarmThreads=5 #integer,required,default(2)
workReqQFullAction=0 #integer,default(0)
name=myWorkManager #required
#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to create or change a work manager information configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify an existing properties file.
  1. Obtain a properties file for the work manager information that you want to change.

You can extract a properties file for a WorkManagerInfo object using the extractConfigProperties command.

2. Open the properties file in an editor and change the properties as needed.

   Ensure that the environment variables in the properties file match your system.

3. Run the applyConfigProperties command.

- If you no longer need the work manager information or an existing custom property, you can delete the entire work manager information object or the custom property.
  - To delete the entire object, specify `DELETE=true` in the header section of the properties file and run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

  - To delete a custom property, specify only the property to be deleted in the properties file and then run the deleteConfigProperties command.

Results

You can use the properties file to configure and manage the work manager information object and its properties.

What to do next

Save the changes to your configuration.

Working with work manager information J2EE resource properties files:

You can use properties files to create, modify, or delete work manager information Java 2 Platform, Enterprise Edition (J2EE) resource custom properties.

Before you begin

Determine the changes that you want to make to your work manager information J2EE resource configuration.

Start the wsadmin scripting tool. To start wsadmin using the Jython language, run the `wsadmin -lang Jython` command from the `bin` directory of the server profile.

About this task

Using a properties file, you can create, modify, or delete work manager information J2EE resource custom properties.

Run administrative commands using wsadmin to change a properties file for a work manager information J2EE resource, validate the properties, and apply them to your configuration.

Table 564. Actions for work manager information J2EE resource properties. You can create, modify, and delete work manager information J2EE resource custom properties.

Action	Procedure
create	Not applicable
modify	Edit properties and then run the applyConfigProperties command to modify the value of an existing custom property.
delete	Not applicable
create Property	Set properties and then run the applyConfigProperties command to create a custom property.

Table 564. Actions for work manager information J2EE resource properties (continued). You can create, modify, and delete work manager information J2EE resource custom properties.

Action	Procedure
delete Property	Specify the properties to delete in the properties file and then run the deleteConfigProperties command to delete an existing custom property. The properties file must contain only the properties to be deleted.

Optionally, you can use interactive mode with the commands:

```
AdminTask.command_name('-interactive')
```

Procedure

- Create work manager information J2EE resource properties.
  1. Specify WorkManagerInfo J2EEResourcePropertySet custom properties in a properties file.

     Open an editor and specify work manager information J2EE resource properties in a properties file. You can copy the following example properties into an editor and modify the properties as needed for your situation. To specify a custom property, edit the `AttributeInfo` value and properties values.

```
#
# Header
#
ResourceType=J2EEResourcePropertySet
ImplementingResourceType=WorkManagerProvider
ResourceId=Cell=!{cellName}:WorkManagerProvider=myWorkManagerProvider:WorkManagerInfo=jndiName#myWorkManagerJndiName:J2EEResourcePropertySet=
AttributeInfo=resourceProperties(name,value)
#

#
#Properties
#
existingProp=newValue
newProp=newValue


#
EnvironmentVariablesSection
#
#
#Environment Variables
cellName=myCell04
```

  2. Run the applyConfigProperties command to create or change a work manager information J2EE resource configuration.

     Running the applyConfigProperties command applies the properties file to the configuration. In this Jython example, the optional `-reportFileName` parameter produces a report named `report.txt`:

```
AdminTask.applyConfigProperties(['-propertiesFileName myObjectType.props -reportFileName report.txt '])
```

- Modify existing work manager information J2EE resource properties.
  1. Obtain a properties file for the work manager information J2EE resource that you want to change.

     You can extract a properties file for a WorkManagerInfo J2EEResourcePropertySet using the extractConfigProperties command.
  2. Open the properties file in an editor and change the custom properties as needed.

     Ensure that the environment variables in the properties file match your system.
  3. Run the applyConfigProperties command.
- If you no longer need a work manager information J2EE resource custom property, you can delete the custom property.

  To delete one or more custom properties, specify only the properties to delete in the properties file and then run the deleteConfigProperties command; for example:

```
AdminTask.deleteConfigProperties('[-propertiesFileName myObjectType.props -reportFileName report.txt]')
```

Results

You can use the properties file to configure and manage the work manager information J2EE resource properties.

What to do next

Save the changes to your configuration.

Working with web services endpoint URL fragment property files

You can use property files to manage or change endpoint URL fragments for web services accessed through HTTP, SOAP and Java Message Service (JMS), or directly as enterprise beans. URL fragments are used to form complete web services endpoint addresses included in published Web Services Description Language (WSDL) files.

Before you begin

Endpoint URL fragments are optional metadata for web services applications. You can use either the administrative console or property files to configure and manage URL fragments. Before you can query the URL fragments, you must first set the URL fragments using either the administrative console or the applyConfigProperties command. After you initially set the URL fragments using the administrative console or property files, you can now modify the web services endpoint URL fragment using property files or the administrative console.

To learn about using the administrative console to set the URL fragments, see the information about configuring endpoint URL information for HTTP bindings or configuring endpoint URL information for JMS bindings.

About this task

Note: Version 8.0 supports using property files to manage endpoint URL fragments for web services accessed through HTTP, SOAP and Java Message Service (JMS), or directly as enterprise beans.

You can specify a portion of the endpoint URL to use in each web service module. The portion that you specify is used to create the actual endpoint URL when publishing a WSDL file. In a published WSDL file, the URL defining the target endpoint address is found in the location attribute of the port soap:address element. This page applies for both Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

The web service endpoint URL fragment properties are extensions to the application properties file. When using the extractConfigProperties command to get a properties file for an application, you also get the endpoint URL fragments along with other application properties. Use the applyConfigProperties, validateConfigProperties, and deleteConfigProperties commands; as described in the procedure, to update, validate and delete endpoint URL fragments in a web services application.

Procedure

1. Extract the property file.

   For example, to extract the properties for the application, sampleApplication, use the extractConfigProperties command.

```
AdminTask.extractConfigProperties(['-propertiesFileName', 'myProperties.props', '-configData',
'Deployment=sampleApplication' ])
```

   If you previously configured your web services endpoint URL fragments, the system extracts the properties files and the result contains a section for endpoint URL fragments, as the following example displays:

```
#
# CWSAD0103I: URLPrefixMap Section: module=TestApp.jar
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
ExtensionId=ApplicationWebServicesExtension
#

#
#Properties
http=http://myhostname:80
module=TestApp.jar
```

2. Create a new instance of the property file.

   If you modify existing URL fragment properties or create new properties, use the following command to apply URL fragment properties. This command creates or updates the corresponding metadata file in the specified application. In this example, the optional `-reportFileName` parameter is used to specify to produce a report from the command.

```
AdminTask.applyConfigProperties('[-propertiesFileName myProperties.props -reportFileName report.txt ]')
```

3. Validate the property file.

   The validateConfigProperties command validates the property names and values in the properties file. If all the names and values are valid, the command result is true. Otherwise, the command returns a false value; for example:

```
AdminTask.validateConfigProperties(['-propertiesFileName', 'myProperties.props', '-reportFileName',
'report.txt'])
```

4. Delete the property file.

   When a deleteConfigProperties command is invoked, the command deletes all of the properties that are specified within the properties file. If you are starting with a previously extracted properties file, remove all sections within the properties file, except for the sections that you want to apply the deleteConfigProperties command. For example, if you want to delete a URL Prefix Map, remove all sections from the extracted properties file except the URLPrefix map section and possibly the environment variables section, if you are using variables.

   The deleteConfigProperties command deletes the properties specified in the properties file; for example:

```
AdminTask.deleteConfigProperties(['-propertiesFileName', 'myProperties.props', '-reportFileName',
 'report.txt'])
```

   The following `myProperties.txt` file is an example of a properties file that you can use to perform a delete of the `http` URL Prefix map for the `TestApp.jar` module:

```
#
# CWSAD0103I: URLPrefixMap Section: module=TestApp.jar
#
ResourceType=Application
ImplementingResourceType=Application
ResourceId=Cell=!{cellName}:Deployment=!{applicationName}
ExtensionId=ApplicationWebServicesExtension
#

#
#Properties
http=http://myhostname:80
module=TestApp.jar

#
#
EnvironmentVariablesSection
#
#
#Environment Variables
applicationName=TestApp
cellName=XYZNode01Cell
serverName=server1
nodeName=XYZNode01
```

Results

You can use web services property files to query, configure, and manage the web service endpoint URL fragment for different protocols.

Properties for web services endpoint URL fragments using property files

You can use properties files to work with web services endpoint URL fragments.

The web services endpoint URL fragment is a portion of the endpoint URL that you can specify in each web services module. In a published Web Services Description Language (WSDL) file, the URL fragment is used to create the actual endpoint URL that defines the target endpoint address, which is found in the location attribute of the port `soap:address` element.

The following property names exist for URL fragments:

- "ejb"
- "http "
- "jms" on page 782
- "module" on page 782

ejb

This property is only applicable for Java API for XML-based RPC (JAX-RPC) web services.

Specifies a URL fragment for web services accessed through an Enterprise JavaBeans (EJB) module binding. The URL fragment value entered is a suffix that is appended to the initial part of the URL obtained by examining the deployment information of the web service. The following code snippet is an example of a URL fragment from the deployment information of an EJB:

```
wsejb:/com.acme.sample.MyStockQuoteHome?jndiName=ejb/MyStockQuoteHome
```

In this case, by entering the following information in the URL fragment field,

```
jndiProviderURL=corbaloc:iiop:myhost.mycompany.com:2809
```

the resulting URL becomes

```
wsejb:/com.acme.sample.MyStockQuoteHome?jndiName=ejb/MyStockQuoteHome&jndiProviderURL=corbaloc:iiop:myhost.mycompany.com:2809
```

http

This property is applicable for both Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

Specifies a URL fragment for web services accessed through an HTTP transport. The URL fragment format is `protocol://host_name:port_number`, where the `protocol` is either `http` or `https`; for example, `http://myHost:9045`. The URL fragment is a prefix that is followed by the context-root of the module and the web services `url-pattern` specified in the published WSDL file; for example, `http://myHost:9045/services/myService`.

If the web services in a module are accessed directly from the web services application server, use the host name for the application server and one of the ports from the virtual host for the module.

If the web services in a module are accessed through an intermediate node, such as the web services gateway or an IBM HTTP Server web server, specify the `protocol`, `host`, and `port_number` parameters of the intermediate service. This configuration specifies a custom HTTP URL prefix; therefore, you must also configure the custom JVM property, `com.ibm.ws.webservices.enableHTTPPrefix`, and set the value to `true`. Restart the application server for your changes to take effect.

jms

This property is applicable for both Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

Specifies a URL fragment for web services accessed through a JMS transport. The URL fragment is a prefix to which the `targetService` property is appended to form a complete JMS URL endpoint. The default value is obtained by examining the deployment information of the installed service; for example: `jms:jndi:jms/MyQueue&jndiConnectionFactoryName=jms/MyCF`.

You can modify the URL fragment by adding properties; for example:

```
jms:jndi:jms/MyQueue&jndiConnectionFactoryName=jms/MyCF&priority=5
```

The URL fragment is then combined with the `targetService` property to form the complete URL; for example:

```
jms:jndi:jms/MyQueue&jndiConnectionFactoryName=jms/MyCF&priority=5&targetService=GetQuote
```

module

This property is applicable for both Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

Specifies the module name that contains the URL fragments for HTTP, EJB, and JMS protocols.

Note: Ensure that you do not configure multiple URL prefix map sections for the same module. Configuring multiple URL prefix map sections that specify the same module creates ambiguity regarding which updates are being requested.

Chapter 19. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
The default Application Client for IBM WebSphere Application Server profile root is the `/QIBM/UserData/WebSphere/AppClient/V8/client/profiles/`*`profile_name`* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppServer/V8/Base` directory.

java_home

Table 565. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit`
64–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit`

plugins_profile_root
The default Web Server Plug-ins profile root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/`*profile_name* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the `/QIBM/ProdData/WebSphere/Plugins/V8/webserver` directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver` directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to `.exe`, `.dll`, `.so` objects) for the installed product. The product library name is `QWAS8`*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is `QWAS8A`. The *`app_server_root`*`/properties/product.properties` file contains the value for the product library of the installation, `was.install.library`, and is located under the *`app_server_root`* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/`*`profile_name`* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS8`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base` directory.

The `profiles` and `profileRegistry` subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Chapter 20. Using the Administration Thin Client

With the Administration Thin Client, you can run the wsadmin tool or a standalone administrative Java program with only a couple of Java archive (JAR) files. This reduces the amount of time that it takes for the wsadmin tool to start and improved performance. This information should be used to set up JMX client programs.

Before you begin

Verify that the IBM Software Development Kit (SDK) is installed on the Administration Thin Client. It is recommended that you use the same IBM SDK as the server that it will connect to in the administrative Thin Client environment.

Important: The Administration Thin Client is supported for the IBM SDKs. It is also supported for the corresponding Oracle Java Development Kit (JDK) with the same major version, provided that:

- The client administration application uses only the SOAP connector.
- The client administration application uses the `com.ibm.websphere.management.AdminClientFactory` API to get the JMX client.

Attention: You cannot run a thin administrative client with the -conntype NONE option.

About this task

You cannot use the Administration Thin Client to manage feature packs nor deploy application artifacts that are specific to feature packs.

The Administration Thin Client does not support the installation of store archive (SAR) files or editing applications that use an external Java Authorization Contract for Containers (JACC) provider, for example, Tivoli Access Manager.

The Administrative Thin Client does not support passing the Kerberos token to a server. It only supports basic authentication, which passes the user ID and password.

The Administration Thin Client does not support co-existence with other thin clients.

The Administration Thin Client on the z/OS platform does not use localcomm.

For tracing and logging information for the Administration Thin Client, see Enabling trace on client and stand-alone applicationsin the *Troubleshooting and support* PDF.

Procedure

1. Make the Administration Thin Client JAR files available by copying com.ibm.ws.admin.clientXXX.jar from a WebSphere Application Server environment to an environment outside of WebSphere Application Server, for example, `c:\`*`MyThinClient`*. The `com.ibm.ws.admin.client_8.0.0.jar` Administration Thin Client JAR file is located in one of the following locations:
   - The *`AppServer`*`/runtimes` directory.
   - The *`AppClient`*`/runtimes` directory, if you optionally selected the Administration Thin Client when you installed the application client.
2. Use the Administration Thin Client JAR files to compile and test administration client programs. For Java applications, you can compile and run the JAR files within a standard Java 2 Platform, Standard Edition environment. For more information, see the “Compiling an administration application using the Thin Administration Client” on page 787 topic.

3. Copy the messages directory from the *app_server_root*/properties directory to the C:\MyThinClient\properties directory.
4. If security is turned on, you also need the following files:
   - Copy the com.ibm.ws.security.crypto.jar file from either the AppServer/plugins directory or the AppClient/plugins directory and put it in the C:*MyThinClient* directory.
   - If you are using the IPC connector, optionally copy the ipc.client.props file from the AppServer\profiles*profileName*/properties or the AppClient\properties directory and put it in the C:\MyThinClient\properties directory. Alternatively, you can set the properties in the ipc.client.props file programmatically in your Java code.
   - If you are using the SOAP connector, optionally copy the soap.client.props file from the AppServer\profiles*profileName*\properties directory and put it in the C:*MyThinClient*\properties directory. Then, enable the client security by setting the com.ibm.CORBA.securityEnabled property to true. Alternatively, you can set the properties in the soap.client.props file programmatically in your Java code.
   - If you are using RMI or JSR160RMI connectors, copy the sas.client.props file from the AppServer/profiles/*profileName*/properties directory and put it in the C:*MyThinClient*\properties directory.

     If there is a firewall in your installation and your application client uses RMI to receive notifications, you might get a RemoteException error after trying to deploy an application. A notifications listener with RMI does not work across a firewall because it requires a listener port on the client listening for notifications. This listener port is not directly accessible when the server tries to send notifications back to the client port. Instead of RMI, use a SOAP connector, which polls for notifications.
   - Copy or generate the ssl.client.props file from either the AppServer\profiles*profileName*/properties directory or the AppClient/properties directory and put it in the C:*MyThinClient*\properties directory.

     Attention: This file contains the user.root property. You must modify the value to your thin client directory, for example, C:*MyThinClient*.
   - If you are using the IBM SDK, copy the key.p12 and trust.p12 files from AppServer\profiles*profileName**etc* directory and put it to C:*MyThinClient**etc* directory. To complete this task, copy the file to your thin client directory or run a script to generate the file. For more information, see the following topics:
     – ssl.client.props client configuration file
     – Interoperating with previous product versions
     – retrieveSigners command
     – Secure installation for client signer retrieval in SSL
   - Copy the wsjaas_client.conf files from either the AppServer\profiles*profileName*/properties directory or the AppClient/properties directory, and put them in the C:\MyThinClient\properties directory.
   - If you are using the Sun JDK, change the following properties in the ssl.client.props file so that it uses JKS key and truststores and Sun Microsystems implementations on the key and trust managers:

     ```
     com.ibm.ssl.alias=DefaultSSLSettings
     com.ibm.ssl.protocol=SSL
     com.ibm.ssl.securityLevel=HIGH
     com.ibm.ssl.trustManager=SunX509
     com.ibm.ssl.keyManager=SunX509
     com.ibm.ssl.contextProvider=SunJSSE
     com.ibm.ssl.enableSignerExchangePrompt=gui

     # Keystore information
     com.ibm.ssl.keyStoreName=ClientDefaultKeyStore
     com.ibm.ssl.keyStore=${user.root}/etc/keystore.jks
     com.ibm.ssl.keyStorePassword=keystore_password
     com.ibm.ssl.keyStoreType=JKS
     ```

```
com.ibm.ssl.keyStoreProvider=SUN
com.ibm.ssl.keyStoreFileBased=true

# Truststore information
com.ibm.ssl.trustStoreName=ClientDefaultTrustStore
com.ibm.ssl.trustStore=${user.root}/etc/truststore.jks
com.ibm.ssl.trustStorePassword=truststore_password
com.ibm.ssl.trustStoreType=JKS
com.ibm.ssl.trustStoreProvider=SUN
com.ibm.ssl.trustStoreFileBased=true
```

5. Launch the Administration Thin Client or run the wsadmin tool remotely in a Java 2 Platform, Standard Edition environment. To launch your administration application, use the following sample launch scripts:

   Note: If you are using the Oracle JDK, set the com.ibm.websphere.thinclient JVM property to true.

```
#!/bin/bash
WAS_HOME=/MyThinClient
USER_INSTALL_ROOT=${WAS_HOME}
JAVA_HOME=location_of_the_JRE_file

# C_PATH is the class path.  Add to it as needed.
C_PATH=${WAS_HOME}/com.ibm.ws.admin.client_8.0.0.jar:${WAS_HOME}/com.ibm.ws.security.crypto.jar
SOAPURL=-Dcom.ibm.SOAP.ConfigURL=${WAS_HOME}/properties/soap.client.props

TC=-Dcom.ibm.websphere.thinclient=true

if [[ -f ${JAVA_HOME}/bin/java ]]; then
   JAVA_EXE="${JAVA_HOME}/bin/java"
else
   JAVA_EXE="${JAVA_HOME}/jre/bin/java"
fi

${JAVA_EXE}  -classpath "${C_PATH}" $TC -Duser.install.root=${USER_INSTALL_ROOT}
-Dcom.ibm.SSL.ConfigURL=file:${WAS_HOME}/properties/ssl.client.props ${SOAPURL} your_class_file
```

6. Deploy the stand-alone JAR files and the administrative client applications and start applications outside of a WebSphere Application Server environment.

Compiling an administration application using the Thin Administration Client

Use the wsadmin tool to use the thin administrative client to compile an application.

About this task

To use the thin administrative client JAR files to compile an application, perform the following steps:

Procedure

1. Include the `com.ibm.ws.admin.client_8.0.0.jar` JAR file in the CLASSPATH.
2. Compile the application.

Example

For example, the JAR file is located in the *app_server_root*`/runtimes` directory and to compile the ThinAdminClientApplication.java file in the current directory. Use the following to compile the file::

```
export CLASSPATH=app_server_root/runtimes/com.ibm.ws.admin.client_8.0.0.jar:${CLASSPATH}
${JAVA_HOME}/bin/javac ThinAdminClientApplication.java
```

Running the wsadmin tool remotely in a Java 2 Platform, Standard Edition environment

The thin administrative client adds Java archive (JAR) files that support administrative client functions that you can use with IBM Developer Kits For the Java Platform.

About this task

Attention: Be aware of the following limitations:

- You cannot run a thin administrative client with the -conntype NONE option.
- The Administration Thin Client is supported for the IBM SDKs. The Administration Thin Client is supported for the corresponding Oracle Java Development Kit (JDK) with the same major version, provided that the client administration application uses only the SOAP connector.
- The client administration application must use `the com.ibm.websphere.management.AdminClientFactory` API to get the JMX client.
- Thin administrative clients do not support the installation of SAR files or the editing of applications that use an external JACC provider such as Tivoli Access Manager.
- Thin administrative clients do not support the installation of SAR files or the editing of applications that use an external JACC provider such as Tivoli Access Manager.

For tracing and logging information for the thin administrative client, see the Enabling trace on client and stand-alone applications article in the *Troubleshooting and support* PDF.

Procedure

1. Obtain the thin administrative client JAR file and other required files that are required when security is on from the WebSphere Application Server, Network Deployment installation. Refer to the topic Using the administration thin client for details about the files that you need to perform this task.
2. Generate the wsadmin.sh or the wsadmin.bat file from the server machine.

   There is an example of wsadmin.bat step 4.
3. Copy the Java directory from the server installation to your thin client environment.
4. Start the wsadmin tool in a non-OSGi environment. Examples of the wsadmin.bat file and the wsadmin.sh file are as follows:

   Example: wsadmin.bat

```
@REM wsadmin launcher
@echo off
@REM Usage: wsadmin arguments setlocal
@REM was home should point to whatever directory you decide for your thin client environment

set WAS_HOME=c:\MyThinClient
set USER_INSTALL_ROOT=%WAS_HOME%

@REM Java home should point to where you installed java for your thinclient
set JAVA_HOME="%WAS_HOME%\java"
set WAS_LOGGING=-Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager -Djava.util.logging.configureByServer=true
set THIN_CLIENT=-Dcom.ibm.websphere.thinclient=true

if exist "%JAVA_HOME%\bin\java.exe" (
    set JAVA_EXE="%JAVA_HOME%\bin\java" )
 else (
    set JAVA_EXE="%JAVA_HOME%\jre\bin\java" )

@REM CONSOLE_ENCODING controls the output encoding used for stdout/stderr
@REM console - encoding is correct for a console window
@REM file - encoding is the default file encoding for the system
@REM other - the specified encoding is used.  e.g. Cp1252, Cp850, SJIS
@REM SET CONSOLE_ENCODING=-Dws.output.encoding=console
@REM For debugging the utility itself
@REM set WAS_DEBUG=-Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777

set CLIENTSOAP=-Dcom.ibm.SOAP.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\soap.client.props
set CLIENTSAS=-Dcom.ibm.CORBA.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\sas.client.props
set CLIENTSSL=-Dcom.ibm.SSL.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\ssl.client.props
set CLIENTIPC=-Dcom.ibm.IPC.ConfigURL=file:"%USER_INSTALL_ROOT%"\properties\ipc.client.props
set JAASSOAP=-Djava.security.auth.login.config=%USER_INSTALL_ROOT%\properties\wsjaas_client.conf
```

```
@REM the following are wsadmin property
@REM you need to change the value to enabled to turn on trace
set wsadminTraceString=-Dcom.ibm.ws.scripting.traceString=com.ibm.*=all=disabled
set wsadminTraceFile=-Dcom.ibm.ws.scripting.traceFile="%USER_INSTALL_ROOT%"\logs\wsadmin.traceout
set wsadminValOut=-Dcom.ibm.ws.scripting.validationOutput="%USER_INSTALL_ROOT%"\logs\wsadmin.valout

@REM this will be the server host that you will connecting to
set wsadminHost=-Dcom.ibm.ws.scripting.host=myhost.austin.ibm.com

@REM you need to make sure the port number is the server SOAP port number you want to connect to, in this
example the server SOAP port is 8887
set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=SOAP
set wsadminPort=-Dcom.ibm.ws.scripting.port=8887

@REM you need to make sure the port number is the server RMI port number you want to connect to, in this example
the server RMI Port is 2815
@REM set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=RMI
@REM set wsadminPort=-Dcom.ibm.ws.scripting.port=2815

@REM you need to make sure the port number is the server JSR160RMI port number you want to connect to,
in this example the server JSR160RMI Port is 2815
@REM set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=JSR160RMI
@REM set wsadminPort=-Dcom.ibm.ws.scripting.port=2815

@REM you need to make sure the port number is the server IPC port number you want to connect to,
in this example the server IPC Port is 9632 and the host for IPC should be localhost
@REM set wsadminHost=-Dcom.ibm.ws.scripting.ipchost=localhost
@REM set wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=IPC
@REM set wsadminPort=-Dcom.ibm.ws.scripting.port=9632

@REM specify what language you want to use with wsadmin
set wsadminLang=-Dcom.ibm.ws.scripting.defaultLang=jacl
@REM set wsadminLang=-Dcom.ibm.ws.scripting.defaultLang=jython

set SHELL=com.ibm.ws.scripting.WasxShell

:prop set WSADMIN_PROPERTIES_PROP=
if not defined WSADMIN_PROPERTIES goto workspace
set WSADMIN_PROPERTIES_PROP="-Dcom.ibm.ws.scripting.wsadminprops=%WSADMIN_PROPERTIES%"

:workspace set WORKSPACE_PROPERTIES=
if not defined CONFIG_CONSISTENCY_CHECK goto loop
set WORKSPACE_PROPERTIES="-Dconfig_consistency_check=%CONFIG_CONSISTENCY_CHECK%"

:loop
if '%1'=='-javaoption' goto javaoption
if '%1'=='' goto runcmd
goto nonjavaoption

:javaoption
shift
set javaoption=%javaoption% %1
goto again

:nonjavaoption
set nonjavaoption=%nonjavaoption% %1

:again
shift
goto loop

:runcmd

set C_PATH="%WAS_HOME%\properties;%WAS_HOME%\com.ibm.ws.admin.client_8.0.0.jar;%WAS_HOME%\com.ibm.ws.security.crypto.jar"

set PERFJAVAOPTION=-Xms256m -Xmx256m -Xj9 -Xquickstart

if "%JAASSOAP%"=="" set JAASSOAP=-Djaassoap=off

"%JAVA_EXE%" %PERFJAVAOPTION% %WAS_LOGGING% %javaoption% %CONSOLE_ENCODING% %WAS_DEBUG% "%THIN_CLIENT%"
"%JAASSOAP%" "%CLIENTSOAP%" "%CLIENTSAS%" "%CLIENTIPC%" "%CLIENTSSL%" %WSADMIN_PROPERTIES_PROP%
%WORKSPACE_PROPERTIES% "-Duser.install.root=%USER_INSTALL_ROOT%" "-Dwas.install.root=%WAS_HOME%"
%wsadminTraceFile% %wsadminTraceString% %wsadminValOut% %wsadminHost% %wsadminConnType% %wsadminPort%
 %wsadminLang% -classpath %C_PATH% com.ibm.ws.scripting.WasxShell %*

set RC=%ERRORLEVEL%

goto END

:END

@endlocal
set MYERRORLEVEL=%ERRORLEVEL% if defined PROFILE_CONFIG_ACTION exit %MYERRORLEVEL% else exit /b %MYERRORLEVEL%
```

Example: wsadmin.sh

```
#!/bin/bash
# example wsadmin launcher
# WAS_HOME should point to the directory for the thin client

WAS_HOME="/MyThinClient"
USER_INSTALL_ROOT="/MyThinClient"

# JAVA_HOME should point to where java is installed for the thin client
JAVA_HOME="$WAS_HOME/java"

WAS_LOGGING="-Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager -Djava.util.logging.configureByServer=true"

if [[ -f ${JAVA_HOME}/bin/java ]]; then
   JAVA_EXE="${JAVA_HOME}/bin/java"
else
   JAVA_EXE="${JAVA_HOME}/jre/bin/java"
fi

# For debugging the utility itself
# WAS_DEBUG=-Djava.compiler="NONE -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777"

CLIENTSOAP="-Dcom.ibm.SOAP.ConfigURL=file:${USER_INSTALL_ROOT}/properties/soap.client.props"
CLIENTSAS="-Dcom.ibm.CORBA.ConfigURL=file:${USER_INSTALL_ROOT}/properties/sas.client.props"
CLIENTSSL="-Dcom.ibm.SSL.ConfigURL=file:${USER_INSTALL_ROOT}/properties/ssl.client.props"
CLIENTIPC="-Dcom.ibm.IPC.ConfigURL=file:${USER_INSTALL_ROOT}/properties/ipc.client.props"

# the following are wsadmin property
# you need to change the value to enabled to turn on trace
wsadminTraceString=-Dcom.ibm.ws.scripting.traceString=com.ibm.*=all=enabled
wsadminTraceFile=-Dcom.ibm.ws.scripting.traceFile=${USER_INSTALL_ROOT}/logs/wsadmin.traceout
wsadminValOut=-Dcom.ibm.ws.scripting.validationOutput=${USER_INSTALL_ROOT}/logs/wsadmin.valout

# this will be the server host that you will be connecting to
wsadminHost=-Dcom.ibm.ws.scripting.host=myhost.austin.ibm.com

# you need to make sure the port number is the server SOAP port number you want to connect to,
in this example the server SOAP port is 8875
wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=SOAP
wsadminPort=-Dcom.ibm.ws.scripting.port=8875

# you need to make sure the port number is the server RMI port number you want to connect to,
in this example the server RMI port is 2811
#wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=RMI
#wsadminPort=-Dcom.ibm.ws.scripting.port=2811

# you need to make sure the port number is the server JSR160RMI port number you want to connect to,
 in this example the server JSR160RMI port is 2811
#wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=JSR160RMI
#wsadminPort=-Dcom.ibm.ws.scripting.port=2811

# you need to make sure the port number is the server IPC port number you want to connect to,
 in this example the server IPC port is 9630
#wsadminHost=-Dcom.ibm.ws.scripting.ipchost=localhost
#wsadminConnType=-Dcom.ibm.ws.scripting.connectionType=IPC
#wsadminPort=-Dcom.ibm.ws.scripting.port=9630

# specify what language you want to use with wsadmin
wsadminLang=-Dcom.ibm.ws.scripting.defaultLang=jacl
#wsadminLang=-Dcom.ibm.ws.scripting.defaultLang=jython

SHELL=com.ibm.ws.scripting.WasxShell

# If wsadmin properties is set, use it
if [[ -n "${WSADMIN_PROPERTIES+V}" ]]; then
     WSADMIN_PROPERTIES_PROP="-Dcom.ibm.ws.scripting.wsadminprops=${WSADMIN_PROPERTIES}"
else
     # Not set, do not use it
     WSADMIN_PROPERTIES_PROP=
fi

# If config consistency check is set, use it
if [[ -n "${CONFIG_CONSISTENCY_CHECK+V}" ]]; then
     WORKSPACE_PROPERTIES="-Dconfig_consistency_check=${CONFIG_CONSISTENCY_CHECK}"
else
     WORKSPACE_PROPERTIES=
fi


# Parse the input arguments
isJavaOption=false
nonJavaOptionCount=1
for option in "$@" ; do
  if [ "$option" = "-javaoption" ] ; then
     isJavaOption=true
  else
     if [ "$isJavaOption" = "true" ] ; then
        javaOption="$javaOption $option"
        isJavaOption=false
     else
        nonJavaOption[$nonJavaOptionCount]="$option"
```

```
        nonJavaOptionCount=$((nonJavaOptionCount+1))
      fi
  fi
done

DELIM=" "
C_PATH="${WAS_HOME}/properties:${WAS_HOME}/com.ibm.ws.admin.client_8.0.0.jar:${WAS_HOME}/com.ibm.ws.security.crypto.jar"

#Platform specific args...
PLATFORM=`/bin/uname`
case $PLATFORM in
  AIX | Linux | SunOS | HP-UX)
    CONSOLE_ENCODING=-Dws.output.encoding=console ;;
  OS/390)
    CONSOLE_ENCODING=-Dfile.encoding=ISO8859-1
    EXTRA_X_ARGS="-Xnoargsconversion" ;;
esac

# Set java options for performance
PLATFORM=`/bin/uname`
case $PLATFORM in
  AIX)
      PERF_JVM_OPTIONS="-Xms256m -Xmx256m -Xquickstart" ;;
  Linux)
      PERF_JVM_OPTIONS="-Xms256m -Xmx256m -Xj9 -Xquickstart" ;;
  SunOS)
      PERF_JVM_OPTIONS="-Xms256m -Xmx256m -XX:PermSize=40m" ;;
  HP-UX)
      PERF_JVM_OPTIONS="-Xms256m -Xmx256m -XX:PermSize=40m" ;;
  OS/390)
      PERF_JVM_OPTIONS="-Xms256m -Xmx256m" ;;
esac

if [[ -z "${JAASSOAP}" ]]; then
     JAASSOAP="-Djaassoap=off"
fi

"${JAVA_EXE}" \
    ${PERFJAVAOPTION} \
    ${EXTRA_X_ARGS} \
    -Dws.ext.dirs="$WAS_EXT_DIRS" \
    ${EXTRA_D_ARGS} \
    ${WAS_LOGGING} \
    ${javaoption} \
    ${CONSOLE_ENCODING} \
    ${WAS_DEBUG} \
    "${CLIENTSOAP}" \
    "${JAASSOAP}" \
    "${CLIENTSAS}" \
    "${CLIENTSSL}" \
    "${CLIENTIPC}" \
    ${WSADMIN_PROPERTIES_PROP} \
    ${WORKSPACE_PROPERTIES} \
    "-Duser.install.root=${USER_INSTALL_ROOT}" \
    "-Dwas.install.root=${WAS_HOME}" \
    "-Dcom.ibm.websphere.thinclient=true" \

    ${wsadminTraceFile} \
    ${wsadminTraceString} \
    ${wsadminValOut} \
    ${wsadminHost} \
    ${wsadminConnType} \
    ${wsadminPort} \
    ${wsadminLang} \
    -classpath \
    "${C_PATH}" \
    com.ibm.ws.scripting.WasxShell \
    "${nonJavaOption[@]}"

exit $?
```

Auditing invocations of the wsadmin tool using wsadmin scripting

Run the following wsadmin scripts as part of the environment setup: create the cluster definition, create data sources and JMS object configuration, or install one or more EAR files that comprise the hosted software on the application server. Each of the scripts, wsadmin and non-wsadmin, need to support the ability to capture a log of the activity performed when you run the script.

About this task

In order to set up your application server environment, you must perform multiple tasks. For example, the following non-wsadmin scripts: create the persistent session database, install the JDBC driver for the

database on the system, set up MQ and create MQ queues on the system, or place PDF files in specific locations that are required as part of the application structure. You must also run the following wsadmin scripts as part of the environment setup: create the cluster definition, create data sources and JMS object configuration, or install one or more EAR files that comprise the hosted software on WebSphere Application Server. Each of the scripts, wsadmin and non- wsadmin, need to support the ability to capture a log of the activity performed when you run the script. All of the logs from the scripts are written in a specific directory that archives each time you create an environment. Each time you set up an environment, the overall process is considered a job and each job has an associated identifier. The identifier is a string that includes the date, environment name, machine name, operator, and approval code as indicated by company policy. To examine the logs at a later time, after the environment provisioning is complete, and verify that all of the log files for the wsadmin and non-wsadmin scripts reflect the actual output of the script that you ran for a specific job, and that no other logs are mixed in with the ones from that job, perform the following steps:

Procedure

1. Start the wsadmin tool using the –jobid *string*, –appendTrace *string*, or the -tracefile *string* option. For more information about these options, see the Wsadmin tool topic. For more information about starting the wsadmin tool, see the Starting the wsadmin scripting client topic. Use the -tracefile option to name the logs based on the activity performed by the script that you want to run and to locate the log files in the specific directory for the job. Uses the -appendtrade `true` option to append to an existing log file, if one already exists. Use the -jobid option to embed an identifier within the log file so that you can validate that all of the logs were the result of the same specific provisioning activity and not some other job.

   You can change the name and location of a file. Modifying the contents of the log file can prove difficult. Also, different log files can have the same job ID and each log file needs a unique name. So the -jobid option provides an important audit and correlation function that the -tracefile option cannot provide.
2. Examine the log file for the job ID that you specified. Use the log files to audit or correlate the wsadmin tool.

Example

The following example outputs to the log of the wsadmin tool when you use th -jobid *string* parameter:

```
[5/16/05 15:45:49:449 CDT] 0000000a AbstractShell A   JobID= scriptTest1
```

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
: The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
: The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
The default Application Client for IBM WebSphere Application Server profile root is the /QIBM/UserData/WebSphere/AppClient/V8/client/profiles/*profile_name* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the /QIBM/ProdData/WebSphere/AppServer/V8/Base directory.

java_home

Table 566. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit
64–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit

plugins_profile_root
The default Web Server Plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/*profile_name* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V8/webserver directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V8/webserver directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS8*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is QWAS8A. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, was.install.library, and is located under the *app_server_root* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/*profile_name* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is QWAS8. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V8/Base directory.

The profiles and profileRegistry subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Chapter 21. Troubleshooting with scripting

Use these topics to learn more about troubleshooting with scripting.

About this task

This topic contains the following tasks:

Procedure

- "Tracing operations using the wsadmin scripting tool"
- "Configuring traces using scripting" on page 797
- "Turning traces on and off in servers processes using scripting" on page 798
- "Dumping threads in server processes using scripting" on page 799
- "Setting up profile scripts to make tracing easier using wsadmin scripting" on page 800
- "Enabling the Runtime Performance Advisor tool using scripting" on page 801

Example

You can set the trace string using either of the following supported formats. For example:

```
$AdminControl trace com.ibm.*=all=enabled
```

or

```
$AdminControl trace com.ibm.*=all
```

For more instructions, see the tracing and logging information. To set the log level, see the log level settings information.

Tracing operations using the wsadmin scripting tool

You can enable and disable tracing with scripting and the wsadmin tool.

Before you begin

Before starting this task, the wsadmin tool must be running. See the topic on starting the wsadmin tool.

About this task

Use a trace command to trace operations.

Procedure

- Enable wsadmin client tracing.
  - Using Jacl:

```
$AdminControl trace com.ibm.*=all
```

  - Using Jython:

```
AdminControl.trace('com.ibm.*=all')
```

  where:

Table 567. Syntax explanation. Run the trace command with `com.ibm.*=all` to enable tracing.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process

Table 567. Syntax explanation (continued). Run the trace command with `com.ibm.*=all` to enable tracing.

`trace`	is an AdminControl command
`com.ibm.*=all`	indicates to turn on tracing

- Disable wsadmin client tracing.
  - Using Jacl:

```
$AdminControl trace com.ibm.*=info
```

  - Using Jython:

```
AdminControl.trace('com.ibm.*=info')
```

  where:

Table 568. Syntax explanation. Run the trace command with `com.ibm.*=info` to disable tracing.

`$`	is a Jacl operator for substituting a variable name with its value
`AdminControl`	is an object that enables the manipulation of MBeans running in a WebSphere server process
`trace`	is an AdminControl command
`com.ibm.*=info`	indicates to turn off tracing

Results

The trace command changes the trace settings for the current session. You can change this setting persistently by editing the `wsadmin.properties` file. The property `com.ibm.ws.scripting.traceString` is read by the launcher during initialization. If it has a value, the value is used to set the trace.

The property com.ibm.ws.scripting.traceString can also be passed in as a javaoption through the command line. Passing the property through the command line changes the trace setting for the current session.

```
Enable Tracing
wsadmin.sh -javaoption -Dcom.ibm.ws.scripting.traceString=com.ibm.*=all=enabled
```

A related property, `com.ibm.ws.scripting.traceFile`, designates a file to receive all trace and logging information. The `wsadmin.properties` file contains a value for this property. Run the wsadmin tool with a value set for this property. It is possible to run without this property set, where all logging and tracing goes to the administrative console.

Extracting properties files to troubleshoot your environment using wsadmin scripting

Use this topic to create a properties file that displays configuration information for a node, application server, application, or other resource. You can use this file to troubleshoot a problem in your environment.

About this task

To debug problems in your environment, you can use the wsadmin tool to create a properties file to review your configuration. The properties file includes the most commonly used attributes or configuration data and values for the resource of interest. You can create a properties file for any of the following resources:

- Nodes
- Profiles
- Application servers
- Virtual hosts
- Authorization tables

- Data replication domains
- Variable maps
- Java™ Database Connectivity (JDBC) providers
- Uniform Resource Locator (URL) providers
- Mail providers
- Resource environment providers
- Java 2 Connector (J2C) resource adapters

Use properties files to troubleshoot your configuration. If you cannot resolve the error, you can provide IBM Support with a copy of the properties file.

Procedure

1. Start the wsadmin scripting tool.
2. Extract the application server configuration of interest.

Table 569. extractConfigProperties parameter descriptions. Run the extractConfigProperties command with parameters to extract a specific object configuration.

Parameter	Description
-propertiesFileName	Specifies the name of the properties file to extract. (String, required)
-configData	Specifies the configuration object instance in the format `Node=node1`. This parameter is required if you do not specify the configuration object name as the target object. (String, optional)
-options	Specifies additional configuration options, such as GENERATETEMPLATE=true. (Properties, optional)
-filterMechanism	Specifies filter information for extracting configuration properties. Specify All to extract all configuration properties. Specify NO_SUBTYPES to exclude properties specified with the selectedSubTypes parameter. Specify SELECTED_SUBTYPES to extract specific configuration properties specified with the selectedSubTypes parameter. (String, optional)
-selectedSubTypes	Specifies the configuration properties to include or exclude when the command extracts the properties. Specify this parameter if you set the filterMechanism parameter to NO_SUBTYPES or SELECTED_SUBTYPES. The following strings are examples of sever subtypes: ApplicationServer, EJBContainer. (String, optional)

The following example extracts the properties configuration for the `server1` application server:

```
AdminTask.extractConfigProperties('-propertiesFileName ConfigProperties_server1.props -configData Server=server1')
```

The system extracts the properties file, which contains each of the configuration objects and attributes for the `server1` application server.

The system extracts the properties file, which contains each of the configuration objects and attributes for the `dmgr` deployment manager.

Results

The system creates a properties file based on the resource configuration of interest.

Configuring traces using scripting

Use the wsadmin tool and scripting to configure traces for a configured server.

Before you begin

Before starting this task, the wsadmin tool must be running. See the starting the wsadmin scripting client information.

About this task

Perform the following steps to set the trace for a configured server:

Procedure

1. Identify the server and assign it to the server variable:
   - Using Jacl:

     ```
     set server [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     server = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
     print server
     ```

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

2. Identify the trace service belonging to the server and assign it to the tc variable:
   - Using Jacl:

     ```
     set tc [$AdminConfig list TraceService $server]
     ```

   - Using Jython:

     ```
     tc = AdminConfig.list('TraceService', server)
     print tc
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#TraceService_1)
   ```

3. Set the trace string. The following example sets the trace string for a single component:
   - Using Jacl:

     ```
     $AdminConfig modify $tc {{startupTraceSpecification
     com.ibm.websphere.management.*=all=enabled}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(tc, [['startupTraceSpecification',
     'com.ibm.websphere.management.*=all=enabled']])
     ```

4. The following command sets the trace string for multiple components:
   - Using Jacl:

     ```
     $AdminConfig modify $tc {{startupTraceSpecification
     com.ibm.websphere.management.*=all=enabled:com.ibm.ws.
     management.*=all=enabled:com.ibm.ws.runtime.*=all=enabled}}
     ```

   - Using Jython:

     ```
     AdminConfig.modify(tc, [['startupTraceSpecification',
     'com.ibm.websphere.management.*=all=enabled:com.ibm.ws.
     management.*=all=enabled:com.ibm.ws.runtime.*=all=enabled']])
     ```

5. Save the configuration changes. See the saving configuration changes with the wsadmin tool information.

Turning traces on and off in servers processes using scripting

You can use scripting to turn traces on or off in server processes.

Before you begin

Before starting this task, the wsadmin tool must be running. See the starting the wsadmin scripting client information.

About this task

Perform the following steps to turn traces on and off in server processes:

Procedure

1. Identify the object name for the TraceService MBean running in the process:
   - Using Jacl:
     ```
     $AdminControl completeObjectName type=TraceService,node=mynode,process=server1,*
     ```
   - Using Jython:
     ```
     AdminControl.completeObjectName('type=TraceService,node=mynode,process=server1,*')
     ```
2. Obtain the name of the object and set it to a variable:
   - Using Jacl:
     ```
     set ts [$AdminControl completeObjectName type=TraceService,process=server1,*]
     ```
   - Using Jython:
     ```
     ts = AdminControl.completeObjectName('type=TraceService,process=server1,*')
     ```
3. Turn tracing on or off for the server. For example:
   - To turn tracing on, perform the following step:
     - Using Jacl:
       ```
       $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=enabled
       ```
     - Using Jython:
       ```
       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=enabled')
       ```
   - To turn tracing off, perform the following step:
     - Using Jacl:
       ```
       $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=disabled
       ```
     - Using Jython:
       ```
       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=disabled')
       ```

Dumping threads in server processes using scripting

Use the AdminControl object to dump the Java threads of a running server.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 article for more information.

About this task

Procedure

Issue one of the following commands to produce a Java core dump.

- Using Jacl:
  ```
  set jvm [$AdminControl completeObjectName type=JVM,process=server1,*]
  $AdminControl invoke $jvm dumpThreads
  ```
- Using Jython:

```
jvm = AdminControl.completeObjectName('type=JVM,process=server1,*')
AdminControl.invoke(jvm, 'dumpThreads')
```

Setting up profile scripts to make tracing easier using wsadmin scripting

You can use scripting and the wsadmin tool to set up profile scripts to facilitate tracing.

Before you begin

Before starting this task, the wsadmin tool must be running. See the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 article for more information.

Procedure

Set up a profile script to make tracing easier. The following profile script example turns tracing on and off for `server1`:

- Using Jacl:

```
proc ton {} {
  global AdminControl
  set ts [$AdminControl queryNames type=TraceService,node=mynode,process=server1,*]
  $AdminControl setAttribute $ts traceSpecification com.ibm.=all=enabled
}

proc toff {} {
  global AdminControl
  set ts [$AdminControl queryNames type=TraceService,node=mynode,process=server1,*]
  $AdminControl setAttribute $ts traceSpecification com.ibm.*=all=disabled
}

proc dt {} {
  global AdminControl
  set jvm [$AdminControl queryNames type=JVM,node=mynode,process=server1,*]
  $AdminControl invoke $jvm dumpThreads
}
```

- Using Jython:

```
def ton():
       global lineSeparator
       ts = AdminControl.queryNames('type=TraceService,node=mynode,process=server1,*')

       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.=all=enabled')

def toff():
       global lineSeparator
       ts = AdminControl.queryNames('type=TraceService,node=mynode,process=server1,*')

       AdminControl.setAttribute(ts, 'traceSpecification', 'com.ibm.*=all=disabled')

def dt():
       global lineSeparator
       jvm = AdminControl.queryNames('type=JVM,node=mynode,process=server1,*')
       AdminControl.invoke(jvm, 'dumpThreads')
```

If you start the wsadmin tool with this profile script, you can use the **ton** command to turn on tracing in the server, the **toff** command to turn off tracing, and the **dt** command to dump the Java threads. For more information about running scripting commands in a profile script, see the Chapter 10, “Starting the wsadmin scripting client using wsadmin scripting,” on page 85 article.

Enabling the Runtime Performance Advisor tool using scripting

You can configure the Runtime Performance Advisor (RPA) using the wsadmin tool or the administrative console.

Before you begin

Before starting this task, the wsadmin tool must be running. See the starting the wsadmin scripting client information.

About this task

The RPA provides advice to help tune systems for optimal performance. See the using the Runtime Performance Advisor information on how to enable this tool using the administrative console. The recommendations display as text in the `SystemOut.log` file.

Note: This topic references one or more of the application server log files. Beginning in WebSphere Application Server Version 8.0 you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files or native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.

The processing of RPA is governed by various rules and corresponding rule IDs. The following table summarizes the mapping between rule IDs and the advice the RPA would process.

Table 570. Rule IDs and nature of advice.

This table maps rule IDs to the nature of advice process by the RPA.

Rule ID	Nature of Advice
ServerRule01	No room for new sessions rule
ServerRule02	Live session
ServerRule03	Session read and write size
ServerRule04	Session read and write time
ServerRule05	Servlet engine unbounded rule
ServerRule06	Servlet engine thread pool rule
ServerRule07	ORB unbounded
ServerRule08	ORB pool rule
ServerRule09	DataSource connection pool min and max size rule
ServerRule10	DataSource prepared statement discard rule
ServerRule11	Memory leak detection rule
surgeModeAlert	Surge mode rule
poolLowEffAlert	Pool low percent efficiency rule
hungConnModeAlert	Hung connection alert rule
connLowEffAlert	Connection low percent efficiency rule
connErrorAlert	Connection error alert rule
LTCSerialReuseViolationAlert	LTC serial reuse violation alert rule
LTCNestingAlert	LTC nesting rule

Table 570. Rule IDs and nature of advice (continued).

This table maps rule IDs to the nature of advice process by the RPA.

Rule ID	Nature of Advice
LTCConnPerThreadLimitAlert	LTC connection per thread limit rule
multiThreadUseViolationAlert	Multi-thread use JCA programming Model Violation
xComponentUseViolationAlert	Cross component use JCA programming model violation

The Runtime Performance Advisor (RPA) requires that the Performance Monitoring Service (PMI) is enabled. It does not require that individual counters be enabled. When a counter that is needed by the RPA is not enabled, the RPA will enable it automatically.

There is no MBean/object available for wsadmin to create a RPA configuration. You can use wsadmin to change the settings and make them effective at runtime. These changes will not be persisted. The changes remain until you stop the server. Since the RPA is disabled once you stop the server, you may want to disable the PMI Service or the counters that were enabled while it was active. You can enable the following counters using the Runtime Performance Advisor:

```
ThreadPools (module)
Web Container (module)
Pool Size
Active Threads
Object Request Broker (module)
Pool Size
Active Threads
JDBC Connection Pools (module)
Pool Size
Percent used
Prepared Statement Discards
Servlet Session Manager (module)
External Read Size
External Write Size
External Read Time
External Write Time
No Room For New Session
System Data (module)
CPU Utilization
Free Memory
```

The following provides an explanation for some of the settings that you can use:

- Calculation interval PMI data - This setting is taken over an interval of time and averaged to provide advice. The calculation interval specifies the length of the time over which data is taken for this advice. Details within the advice messages will appear as averages over this interval.
- Maximum warning sequence - This setting refers to the number of consecutive warnings issued before the threshold is relaxed. For example, if the maximum warning sequence is set to 3, then the advisor only sends three warnings to indicate that the prepared statement cache is overflowing. After that, a new alert is only issued if the rate of discards exceeds the new threshold setting.
- Number of processors - This setting specifies the number of processors on the server. It is critical in order to ensure accurate advice for the specific configuration of the system.

To enable the Runtime Performance Advisor tool using the wsadmin tool, perform the following steps:

Procedure

Setup the Runtime Performance Advisor (RPA), for example:

- Using Jacl:

```
set perf [$AdminControl queryNames mbeanIdentifier=ServerRuleDriverMBean2,process=server1,*]
set enabledVal [java::new java.lang.Boolean true]
set attr [java::new javax.management.Attribute enabled $enabledVal]
set perfObject [$AdminControl makeObjectName $perf]
set ObjectArray [java::new {java.lang.Object[]} 1]
set sigArray [java::new {java.lang.String[]} 1]
$ObjectArray set 0 $attr
$sigArray set 0 "javax.management.Attribute"
$AdminControl invoke_jmx $perfObject setRPAAttribute $ObjectArray  $sigArray

$AdminConfig save
```

What to do next

After completing the previous steps, start the server and monitor RPA.

AdministrationReports command group for the AdminTask object using wsadmin scripting

You can use the Jython or Jacl scripting languages to troubleshoot your configuration with the wsadmin tool. The commands in the AdministrationReports group can be used to create a report of inconsistencies in your system configuration or a report that describes the port usage in the system.

The following commands are available for the AdministrationReports group of the AdminTask object:

- "reportConfigInconsistencies"
- "reportConfiguredPorts"

reportConfigInconsistencies

Use the **reportConfigInconsistencies** command to create a report of inconsistencies in the system configuration.

Target object

None

Required parameters and return values

- Parameters: None
- Returns: A report that describes inconsistencies found in the system.

Interactive mode example usage

Example output:

```
Configuration consistency report for  cell yardbirdCell  cells/yardbirdCell/test.xml is a zero
 length file. cells/yardbirdCell/nodes/DummyNode  does not contain a serverindex.xml  document.
 cells/yardbirdCell/applications/Test. ear/deployments/Test does not contain  a deployment.xml document.
 3 consistency problems were found.
```

reportConfiguredPorts

Use the **reportConfig uredPorts** command to create a report of all the ports configured in the cell.

Target object

None

Required parameters and return values

- Parameters: None
- Returns: A report that describes the port usage in the system.

Examples

Interactive mode example usage:

Example output:

```
Ports configured in cell yardbirdCell  Node yardbirdCellMgr / Server dmgr yardbird:7283
CELL_DISCOVERY_ADDRESS yardbird:9809 BOOTSTRAP_ADDRESS ...  Node dizzyNode1 / Server server1
dizzy:2813 BOOTSTRAP_ADDRESS dizzy:8880 SOAP_CONNECTOR_ADDRESS ...  Node dizzyNode1 / Server
nodeagent dizzy:2814 BOOTSTRAP_ADDRESS dizzy:9904 ORB_LISTENER_ADDRESS
```

Configuring HPEL with wsadmin scripting

You can configure the High Performance Extensible Logging (HPEL) log and trace framework using wsadmin scripting. Use the examples in this topic as a guide to build your own wsadmin scripts.

About this task

HPEL provides faster log and trace handling capabilities and more flexible ways to use log and trace content than the basic mode. You can configure the HPEL mode using the administrative console, or using wsadmin scripting. The examples in this topic show how to configure HPEL using wsadmin. If you complete this task using the deployment manager, then you might need to synchronize the node agent on the target node and restart the server before configuration changes take effect.

Table 571. Variable Names. The table applies to all examples in this topic. All examples use the Jython scripting language.

Variable	Description
myCell	The name of the cell
myNode	The host name of the node
myServer	The name of the server

Procedure

- Use the AdminConfig object to configure HPEL.

  Changes you make using the AdminConfig object take effect the next time you start the server.

  1. Change the trace specification.

     The following example shows how to change the trace specification to *=info:com.ibm.ws.classloader.*=all

     ```
     HPELService = AdminConfig.getid("/Cell:myCell/Node:myNode/Server:myServer/
     HighPerformanceExtensibleLogging:/")
     AdminConfig.modify(HPELService, "[[startupTraceSpec *=info:com.ibm.ws.classloader.*=all]]")
     AdminConfig.save()
     ```

  2. Change the size of the log repository.

     The following example shows how to set HPEL to automatically delete the oldest log content from the log repository when the repository size approaches 65 MB. Specify HPELTrace or HPELTextLog instead of HPELLog to change the setting for the HPEL trace repository or HPEL text log.

     ```
     HPELService = AdminConfig.getid("/Cell:myCell/Node:myNode/Server:myServer/
     HighPerformanceExtensibleLogging:/")
     HPELLog = AdminConfig.list("HPELLog", HPELService)
     AdminConfig.modify(HPELLog, "[[purgeMaxSize 65]]")
     AdminConfig.save()
     ```

  3. Change the log repository location.

The following example shows how to change the HPEL log repository directory name to /tmp/myDirectory. Specify HPELTrace or HPELTextLog instead of HPELLog to change the setting for the HPEL trace repository or HPEL text log.

```
HPELService = AdminConfig.getid("/Cell:myCell/Node:myNode/Server:myServer/
HighPerformanceExtensibleLogging:/")
HPELLog = AdminConfig.list("HPELLog", HPELService)
AdminConfig.modify(HPELLog, "[[dataDirectory  /tmp/myDirectory]]")
AdminConfig.save()
```

4. Disable log record buffering.

   The following example shows how to change the HPEL log repository to not use log record buffering. Specify HPELTrace or HPELTextLog instead of HPELLog to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELService = AdminConfig.getid("/Cell:myCell/Node:myNode/Server:myServer/
   HighPerformanceExtensibleLogging:/")
   HPELLog = AdminConfig.list("HPELLog", HPELService)
   AdminConfig.modify(HPELLog, "[[bufferingEnabled false]]")
   AdminConfig.save()
   ```

   Note: Enable log record buffering in almost all cases. Only disable log record buffering when your server is failing unexpectedly and cannot write buffered content to disk before stopping.

5. Start writing to a new log file each day at a specified time.

   The following example shows how to enable the HPEL log repository to start a new log file each day at 3pm. Specify HPELTrace or HPELTextLog instead of HPELLog to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELService = AdminConfig.getid("/Cell:myCell/Node:myNode/Server:myServer/
   HighPerformanceExtensibleLogging:/")
   HPELLog = AdminConfig.list("HPELLog", HPELService)
   AdminConfig.modify(HPELLog, "[[fileSwitchTime 15]]")
   AdminConfig.modify(HPELLog, "[[fileSwitchEnabled true]]")
   AdminConfig.save()
   ```

6. Change the out of space action for the log repository.

   The following example shows how to change the out of space action for the HPEL log repository. Specify HPELTrace or HPELTextLog instead of HPELLog to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELService = AdminConfig.getid("/Cell:myCell/Node:myNode/Server:myServer/
   HighPerformanceExtensibleLogging:/")
   HPELLog = AdminConfig.list("HPELLog", HPELService)
   AdminConfig.modify(HPELLog, "[[outOfSpaceAction PurgeOld]]")
   AdminConfig.save()
   ```

- Use the AdminControl object to configure HPEL. Changes you make using the AdminControl object take effect immediately.

  1. Change the trace specification.

     The following example shows how to change the trace specification to *=info:com.ibm.ws.classloader.*=all

     ```
     HPELControlMBean = AdminControl.queryNames('cell=myCell,node=myNode,
     type=HPELControlService,process=myServer,*')
     AdminControl.setAttribute(HPELControlMBean, "traceSpecification",
     "*=info:com.ibm.ws.classloader.*=all")
     ```

  2. Change the size of the log repository.

     The following example shows how to set HPEL to automatically delete the oldest log content from the log repository when the repository size approaches 65 MB. Specify HPELTraceDataService or HPELTextLogService instead of HPELLogDataService to change the setting for the HPEL trace repository or HPEL text log.

     ```
     HPELLogDataMBean = AdminControl.queryNames('cell=myCell,
     node=myNode,type=HPELLogDataService,process=myServer,*')
     AdminControl.setAttribute(HPELLogDataMBean, "purgeMaxSize", "65")
     ```

3. Change the log repository location.

   The following example shows how to change the HPEL log repository directory name to /tmp/myDirectory. Specify HPELTraceDataService or HPELTextLogService instead of HPELLogDataService to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELLogDataMBean = AdminControl.queryNames('cell=myCell,
   node=myNode,type=HPELLogDataService,process=myServer,*')
   AdminControl.setAttribute(HPELLogDataMBean, "dataDirectory", "/tmp/myDirectory")
   ```

4. Disable log record buffering.

   The following example shows how to change the HPEL log repository to not use log record buffering. Specify HPELTraceDataService or HPELTextLogService instead of HPELLogDataService to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELLogDataMBean = AdminControl.queryNames('cell=myCell,node=myNode,
   type=HPELLogDataService,process=myServer,*')
   AdminControl.setAttribute(HPELLogDataMBean, "bufferingEnabled", "false")
   ```

   Note: Enable log record buffering in almost all cases. Only disable log record buffering when your server is failing unexpectedly and cannot write buffered content to disk before stopping.

5. Start writing to a new log file each day at a specified time.

   The following example shows how to enable the HPEL log repository to start a new log file each day at 3pm. Specify HPELTrace or HPELTextLog instead of HPELLog to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELLogDataMBean = AdminControl.queryNames('cell=myCell,node=myNode,
   type=HPELLogDataService,process=myServer,*')
   AdminControl.setAttribute(HPELLogDataMBean, "fileSwitchTime", "15")
   AdminControl.setAttribute(HPELLogDataMBean, "fileSwitchEnabled", "true")
   ```

6. Change the out of space action for the log repository.

   The following example shows how to change the out of space action for the HPEL log repository. Specify HPELTraceDataService or HPELTextLogService instead of HPELLogDataService to change the setting for the HPEL trace repository or HPEL text log.

   ```
   HPELLogDataMBean = AdminControl.queryNames('cell=myCell,node=myNode,
   type=HPELLogDataService,process=myServer,*')
   AdminControl.setAttribute(HPELLogDataMBean, "outOfSpaceAction", "PurgeOld")
   ```

Results

HPEL is now configured. If you made changes with the AdminConfig command, restart the server to make the changes take effect.

Chapter 22. Scripting and command line reference material using wsadmin scripting

Use this topic to locate wsadmin tool commands for the AdminTask, AdminControl, AdminConfig, and AdminApp scripting objects. This topic also provides a pointer to command line commands and options.

About this task

All reference topics are located in the **Reference** section of the information center. Use the navigation paths described in this topic to locate specific reference information.

Procedure

- View command line reference topics. This includes administrative commands such as the startServer, manageprofiles, and backupConfig commands. To view all command reference information, use the following navigation path in the information center:

  Reference > **Commands**
- View command reference topics for the wsadmin tool. This includes administrative scripting commands for the AdminTask, AdminConfig, AdminApp, and AdminControl objects. All commands are organized by command group name. To view all scripting reference information, use the following navigation path in the information center:

  Reference > **Administrator scripting interfaces**

wsadmin scripting tool

The wsadmin tool runs scripts. You can use the wsadmin tool to manage application servers as well as the configuration, application deployment, and server runtime operations.

Important: All users who run commands from a specific profile must have authority to modify files that are created by other users that use the same profile. Otherwise, you might see a permission denied error in the log files. To avoid this issue, consider one of the following policies:

- Use specific profiles for distinct user authorities.
- The same user must run all commands in a profile.
- Ensure that all users of a specific profile belong to the same group. In addition, ensure that each user of a group has the read and write authority to the files that are created by other members in the same profile.

The options for the wsadmin tool are case insensitive. If you specify an empty string in place of a command options, the wsadmin tool displays general help information.

gotcha:

- You must enter the wsadmin commands in the wsadmin interactive shell on one line. That is, type the command continuously and do not split the command into multiple lines. Splitting a long wsadmin command into multiple lines is not supported. Trying to run wsadmin commands that are split into multiple lines in the wsadmin interactive shell results in a syntax error.
- The wsadmin tool removes any leading and trailing white space including \n, \r, \t, \f and space when parsing a string to avoid any user errors. For example, someone might accidentally hit the spacebar key or the Tab key and add some white space to the command string. This white space might cause the command or script to failure. If you need to include white space in a command either use the list syntax instead of the string syntax, or enclose the string containing the white space within [ ] (brackets).

Use the following command-line invocation syntax for the wsadmin scripting client:

```
wsadmin [-h(help)]
[-?]

[-c command]

[-p properties_file_name]

[-profile profile_script_name]

[-profileName profile_name]

[-f script_file_name]

[-javaoption java_option]

[-lang language]

[-wsadmin_classpath classpath]

[-profile profile]

[-conntype SOAP [-host host_name] [-port port_number] [-user userid] [-password password]

[-conntype RMI [-host host_name] [-port port_number] [-user userid] [-password password]

[-conntype JSR160RMI [-host host_name] [-port port_number] [-user userid] [-password password]

[-conntype IPC [-ipchost host_name] [-port port_number] [-user userid] [-password password]

[-jobid jobid_string]

[-tracefile trace_file]

[-appendtrace true/false]

[script parameters]
```

The element `script parameters` represents any argument other than the ones listed previously. The *argc* variable contains the number of arguments, and the *argv* variable contains a list of arguments in the order that they were coded.

Options

-c Specifies to run a single command. The -c option must be followed by a command. Multiple -c options can exist on the command line. They run in the order that you designate.

If you invoke the wsadmin tool with the -c option, any changes that you have made to the configuration are then saved automatically. If you make configuration changes and you are not using the -c option, then you must use the **save** command of the AdminConfig object to save the changes. Read about saving configuration changes with the wsadmin tool for more information.

-f Specifies a script to run. The -f option must be followed by a file name.

Only one -f option can exist on the command line.

You can use the -f option to run scripts that contain nested Jython scripts. In the following example, the test2 script imports the test1 script:

```
#test1.py

def   listServer():
       print AdminConfig.list("Server")
```

```
#test2.py

import test1
test1.listServer()
```

To run the caller script, run the following command from the *app_server_root*/bin directory:

```
wsadmin -lang jython -f  test2.py
```

After the scripts run, the system returns the following sample command output:

```
server1(cells/myCell/nodes/myNode/servers/myServer|server.xml#Server_1183122130078)
```

-javaoption

Specifies a valid Java standard or a non-standard option. Multiple -javaoption options can exist on the command line.

```
wsadmin -javaoption java_option -javaoption java_option
```

To shorten the length of the command, you can type the command in the following way:

```
wsadmin -javaoption "java_option java_option"
```

-lang

Specifies the language of the script file, the command, or an interactive shell. The possible languages include: Jacl and Jython. These language options are expressed as `jacl` and `jython`.

This option overrides language determinations that are based on a script file name, a profile script file name, or the com.ibm.ws.scripting.defaultLang property. The -lang argument has no default value.

If you do not specify the -lang argument but you have the -f *script_file_name* argument specified, then the wsadmin tool determines the language based on a target script file name. If you do not specify the -lang argument and the -f argument, the wsadmin tool determines the language based on a profile script file name if the -profile *profile_script_name* argument is specified. If the command line or the property does not supply the script language, and the wsadmin tool cannot determine it, then an error message is generated.

-p

Specifies a properties file. The -p option must be followed by a file name.

The file listed after -p, represents a Java properties file that the scripting process reads. Three levels of default properties files load before the properties file that you specified on the command line:

- The first level is the installation default, `wsadmin.properties`, which is located in the product properties directory.
- The second level is the user default, `wsadmin.properties`, which is located in your home directory.
- The third level is the properties file to which the environment variable `WSADMIN_PROPERTIES` references.

Multiple -p options can exist on the command line. Those options invoke in the order that you supply them.

You can also use the `com.ibm.ws.scripting.noechoParamNo` custom property with this option. Use this custom property to specify script parameters that you do not want to be displayed in a trace file or standard output. To enable this custom property, create a text file such as `noecho.prop`. In this text file, specify positions within your wsadmin command that contain sensitive data, such as passwords, which you do not want to be displayed in a trace file or standard output. The text file can contain a single number or multiple parameter numbers that are separated by a comma. For example, the text file might contain the following information:

```
com.ibm.ws.scripting.noechoParamNo=3,5
```

When you reference this text file in your wsadmin command, the third and fifth parameter values are not displayed in a trace file or standard output. To use the custom property, run the wsadmin command and pass the text file with the -p option. For example:

Using Jacl:

```
wsadmin -f script_file script_arguments -p text_file
```

Using Jython:

```
wsadmin -lang jython -f script_file script_arguments -p text_file
```

For example:

```
wsadmin -f text.py server1 dbuser dbpassword user1 userpassword -p noecho.prop
```

`dbpassword` and `userpassword` are the third and fifth parameters that are not displayed in a trace file or standard output.

-profile
Specifies a profile script.

The profile script runs before other commands, or scripts. If you specify `-c`, then the profile script runs before it invokes this command. If you specify `-f`, then the profile script runs before it runs the script. In interactive mode, you can use the profile script to perform any standard initialization that you want. You can specify multiple -profile options on the command line, and they are invoked in the order that you specify them.

-profileName
Specifies the profile from which the wsadmin tool runs. Specify this option if one the following reasons apply:

- You run the wsadmin tool from the *app_server_root*/`bin` directory, and you do not have a default profile, or you want to run in a profile other than the default profile.
- You are currently in a profile `bin` directory but want to run the wsadmin tool from a different profile.

-?
Provides syntax help.

-help
Provides syntax help.

-conntype
Specifies the type of connection to use.

This argument consists of a string that determines the type, for example, SOAP, and the options that are specific to that connection type. Possible types include: `SOAP`, `RMI`, `JSR160RMI`, `IPC` and `NONE`. For each connection type, you can specify additional attributes about the connection.

For the `SOAP` connection type, you can specify the following attributes:

Table 572. -conntype SOAP connection type attribute descriptions. Use this attribute to specify a SOAP connection type.

Attribute	Description
[-host *host_name*]	Specifies the host name for the connection. The default is the local host.
[-port *port_number*]	Specifies the port number for the connection.
[-user *userid*]	Specifies the user ID to use to establish the connection.
[-password *password*]	Specifies the password to use to establish the connection.

For the `RMI` connection type, you can specify the following attributes:

Table 573. -conntype RMI connection type attribute descriptions. Use this attribute to specify an RMI connection type.

Attribute	Description
[-host *host_name*]	Specifies the host name for the connection. The default is the local host.
[-port *port_number*]	Specifies the port number for the connection.
[-user *userid*]	Specifies the user ID to use to establish the connection.

Table 573. -conntype RMI connection type attribute descriptions (continued). Use this attribute to specify an RMI connection type.

Attribute	Description
[-password *password*]	Specifies the password to use to establish the connection.

For the `JSR160RMI` connection type, you can specify the following attributes:

Table 574. -conntype JSR160RMI connection type attribute descriptions. Use this attribute to specify a JSR160RMI connection type.

Attribute	Description
[-host *host_name*]	Specifies the host name for the connection.
[-port *port_number*]	Specifies the port number for the connection.
[-user *userid*]	Specifies the user ID to use to establish the connection.
[-password *password*]	Specifies the password to use to establish the connection.

For the `IPC` connection type, you can specify the following attributes:

Table 575. -conntype IPC connection type attribute descriptions. Use this attribute to specify an IPC connection type.

Attribute	Description
[-ipchost *host_name*]	Specifies the host name for the connection. This attribute overrides the host name specified for the com.ibm.ws.scripting.ipchost property in the wsadmin.properties properties file.
[-port *port_number*]	Specifies the port number for the connection.
[-user *userid*]	Specifies the user ID to use to establish the connection.
[-password *password*]	Specifies the password to use to establish the connection.

Use the `-conntype NONE` option to run in local mode. The result is that the scripting client is not connected to a running server. You can manage server configuration, the installation and the uninstallation of applications without the application server running.

Note: You should eventually switch from the Remote Method Invocation (RMI) connector to the JSR160RMI connector because support for the RMI connector is deprecated.

-wsadmin_classpath

Use this option to make additional classes available to your scripting process.

Use the following option with a class path string:

```
/home/MyDir/Myjar.jar;/yourdir/yourdir.jar
```

The class path is then added to the class loader for the scripting process.

You can also specify this option in a properties file that is used by the wsadmin tool. The property is com.ibm.ws.scripting.classpath. If you specify -wsadmin_classpath on the command line, the value of this property overrides any value that is specified in a properties file. The class path property and the command-line options are not concatenated.

-host

Specify a host name to which wsadmin attempts to connect. The default wsadmin.properties file located in the properties directory of each profile provides `localhost` as the value of the host property, if this option is not specified.

-password

Specify a password to be used by the connector to connect to the server, if security is enabled in the server.

Attention: On UNIX systems, the use of -password option might result in security exposure because the password information is displayed in the system status program. For example, this information might be exposed when you use the **ps** command, which can be invoked by another user to display all running processes. Do not use this option if security exposure is a concern. To avoid exposure, you can:

- Specify user and password information in the soap.client.props file for the SOAP connector, the sas.client.props file for the JSR160RMI connector or the Remote Method Invocation (RMI) connector, or the ipc.client.props file for the Inter-Process Communications (IPC) connector. The soap.client.props, sas.client.props, and ipc.client.props files are located in the properties directory of your application server profile.
- Wait for the wsadmin tool to prompt the user for login information instead of providing the login information within a properties file.

-user or -username
Specifies a user name to be used by the connector to connect to the server if security is enabled in the server.

-port
Specifies a port to be used by the connector. The default wsadmin.properties file located in the properties directory of each application server profile provides a value in the port property to connect to the local server.

-jobid
Specifies a job ID string so that you can keep track of each invocation of the wsadmin tool for auditing purposes. The jobI D string (jobID=xxxx) is displayed at the beginning of the wsadmin log file.

-tracefile
Specifies the name of the log file and location where the log output is directed. This option overrides the com.ibm.ws.scripting.traceFile property in the wsadmin.properties file.

-appendtrace
Determines if a trace appends to or overrides the end of the existing log file. Specify `true` to append the trace to the end of a log file or specify `false` to override the log file for each wsadmin invocation. The default value is `false`.

The following example specifies the job ID option, log location and appends the trace to the log file.

```
wsadmin -jobid wsadmin_test_1 -tracefile /temp/wsadmin_test_1.log -appendtrace true
```

In the following syntax examples, *mymachine* is the name of the host in the wsadmin.properties file that is specified by the com.ibm.ws.scripting.port property:

SOAP connection to the local host
Use the options that are defined in the `wsadmin.properties` file.

SOAP connection to the *mymachine* host
Using Jacl, enter the following example code:

```
wsadmin -f test1.jacl -profile setup.jacl -conntype SOAP
-port mymachine_soap_portnumber -host mymachine
```

Using Jython:

```
wsadmin -lang jython -f  test1.py -profile setup.py -conntype
SOAP -port mymachine_soap_portnumber -host mymachine
```

Change initial and maximum Java heap size

Using Jacl:

Specify multiple Java options together or separately:

```
wsadmin -javaoption "-Xms128m -Xmx256m" -f test.jacl
```

or

```
wsadmin -javaoption -Xms128m -javaoption -Xmx256m -f test.jacl
```

Using Jython:

Specify multiple Java options together or separately:

```
wsadmin -lang jython -javaoption "-Xms128m -Xmx256m" -f test.py
```

or

```
wsadmin -lang jython -javaoption -Xms128m -javaoption -Xmx256m -f test.py
```

Change system property value

Using Jacl:

```
wsadmin -javaoption "-Dcom.ibm.websphere.management.application.client.jspReloadEnabled=true"
```

Using Jython:

```
wsadmin -lang jython -javaoption "-Dcom.ibm.websphere.management.application.client.jspReloadEnabled=true"
```

Enclose multiple Java options with double-quotation marks (""); for example:

```
wsadmin -javaoption "-Dcom.ibm.websphere.management.application.client.jspReloadEnabled=true
  -Dcom.ibm.websphere.management.application.enableDistribution=true"
```

JSR160RMI connection with security

Using Jacl:

```
wsadmin -conntype JSR160RMI -port  JSR160rmiportnumber -user userid
-password password
```

Using Jython:

```
wsadmin -lang jython -conntype JSR160RMI -port  JSR160portnumber -user userid
-password password
```

The element, *rmiportnumber*, for your connection is displayed in the administrative console as BOOTSTRAP_ADDRESS.

Attention: On UNIX systems, the use of -password option might result in security exposure as the password information becomes visible to the system status program such as **ps** command which can be invoked by another user to display all the running processes. Do not use this option if security exposure is a concern. To avoid exposure, you can:

- Specify user and password information in the soap.client.props file for the SOAP connector, the sas.client.props file for the JSR160RMI connector or the Remote Method Invocation (RMI) connector, or the ipc.client.props file for the Inter-Process Communications (IPC) connector. The soap.client.props, sas.client.props, and ipc.client.props files are located in the properties directory of your application server profile.
- Wait for the wsadmin tool to prompt the user for login information instead of providing the login information within a properties file.

RMI connection with security

Using Jacl:

```
wsadmin -conntype RMI -port  rmiportnumber -user userid
-password password
```

Using Jython:

```
wsadmin -lang jython -conntype RMI -port  rmiportnumber -user userid
-password password
```

The element, *rmiportnumber*, for your connection is displayed in the administrative console as BOOTSTRAP_ADDRESS.

Attention: On UNIX systems, the use of -password option might result in security exposure as the password information becomes visible to the system status program such as **ps** command which can be invoked by another user to display all the running processes. Do not use this option if security exposure is a concern. To avoid exposure, you can:

- Specify user and password information in the soap.client.props file for the SOAP connector, the sas.client.props file for the JSR160RMI connector or the Remote Method Invocation (RMI) connector, or the ipc.client.props file for the Inter-Process Communications (IPC) connector. The soap.client.props, sas.client.props, and ipc.client.props files are located in the properties directory of your application server profile.
- Wait for the wsadmin tool to prompt the user for login information instead of providing the login information within a properties file.

Local mode of operation to perform a single command

Using Jacl:

```
wsadmin -conntype NONE -c "$AdminApp uninstall app"
```

Using Jython:

```
wsadmin -lang jython -conntype NONE -c "AdminApp.uninstall('app')"
```

wsadmin tool performance tips

Follow these tips to get the best performance from the wsadmin tool.

The following performance tips are for the wsadmin tool:

- When you launch a script using the wsadmin tool, a new process is created with a new Java virtual machine (JVM) API. If you use scripting with multiple wsadmin **-c** commands from a batch file or a shell script, these commands run slower than if you use a single wsadmin **-f** command. The -f option runs faster because only one process and JVM API are created for installation and the Java classes for the installation load only once.

  The following example, illustrates running multiple application installation commands from a batch file.

  Using Jacl:

```
wsadmin -c "$AdminApp install /home/myDir/myApps/App1.ear {-appname appl1}"
wsadmin -c "$AdminApp install /home/myDir/myApps/App2.ear {-appname appl2}"
wsadmin -c "$AdminApp install /home/myDir/myApps/App3.ear {-appname appl3}"
```

  Using Jython:

```
wsadmin -lang jython -c "AdminApp.install('/home/myDir/myApps/App1.ear', '[-appname appl1]')"
wsadmin -lang jython -c "AdminApp.install('/home/myDir/myApps/App2.ear', '[-appname appl2]')"
wsadmin -lang jython -c "AdminApp.install('/home/myDir/myApps/App3.ear', '[-appname appl3]')"
```

  Or, for example, using Jacl, you can create the *appinst.jacl* file that contains the commands:

```
$AdminApp install /home/myDir/myApps/App1.ear {-appname appl1}
$AdminApp install /home/myDir/myApps/App2.ear {-appname appl2}
$AdminApp install /home/myDir/myApps/App3.ear {-appname appl3}
```

  Invoke this file using the following command: `wsadmin -f appinst.jacl`

  Or using Jython, you can create the *appinst.py* file, that contains the commands:

```
AdminApp.install('/home/myDir/myApps/App1.ear', '[-appname appl1]')
AdminApp.install('/home/myDir/myApps/App2.ear', '[-appname appl2]')
AdminApp.install('/home/myDir/myApps/App3.ear', '[-appname appl3]')
```

  Then invoke this file using the following command: `wsadmin -lang jython -f appinst.py`.
- Use the AdminControl **queryNames** and **completeObjectName** commands carefully with a large installation. For example, if only a few beans exist on a single machine, the `$AdminControl queryNames *` command performs well.
- The WebSphere Application Server is a distributed system, and scripts perform better if you minimize remote requests. If some action or interrogation is required on several items, for example, servers, it is more efficient to obtain the list of items once and iterate locally. This procedure applies to the actions that the AdminControl object performs on running MBeans, and actions that the AdminConfig object performs on configuration objects.

Commands for the Help object using wsadmin scripting

You can use the Jython or Jacl scripting languages to find general help and dynamic online information about the currently running MBeans with the wsadmin tool. Use the Help object as an aid in writing and running scripts with the AdminControl object.

The following commands are available for the Help object:

AdminApp

Use the **AdminApp** command to view a summary of each available method for the AdminApp object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7095I: The AdminApp object allows application objects to
be manipulated -- this includes installing, uninstalling, editing,
and listing.  Most of the commands supported by AdminApp operate in two
modes: the default mode is one in which AdminApp communicates with the
product to accomplish its tasks.  A local mode is also
possible, in which no server communication takes place.  The local
mode of operation is invoked by bringing up the scripting client with
no server connected using the command line "-conntype NONE" option or setting the
"com.ibm.ws.scripting.connectionType=NONE" property in the wsadmin.properties.

The following commands are supported by AdminApp; more detailed information
about each of these commands is available by using the "help" command of AdminApp and
supplying the name of the command as an argument.

deleteUserAndGroupEntries  Deletes all the user/group information for all the roles and all
the username/password information for RunAs roles for a given application.

edit Edit the properties of an application
```

```
editInteractive Edit the properties of an application interactively

export Export application to a file

exportDDL Export DDL from application to a directory

help Show help information

install Installs an application, given a file name and an option string.

installInteractive Installs an application in interactive mode, given a file name
and an option string.

isAppReady Checks whether the application is ready to be run

list List all installed applications, either all applications or applications on a given target scope.

listModules List the modules in a specified application

options Shows the options available, either for a given file, or in general.

publishWSDL Publish WSDL files for a given application

taskInfo Shows detailed information pertaining to a given installation task for a given file

uninstall Uninstalls an application, given an application name and an option string

updateAccessIDs Updates the user/group binding information with accessID from user
registry for a given application

view View an application or module, given an application or module name
```

Examples

- Using Jacl:

  ```
  $Help AdminApp
  ```

- Using Jython:

  ```
  print Help.AdminApp()
  ```

AdminConfig

Use the **AdminConfig** command to view a summary of each available method for the AdminConfig object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7053I: The following functions are
supported by AdminConfig:


create Creates a configuration object,
given a type, a parent, and

a list of attributes
```

```
create Creates a configuration object,
given a type, a parent, a

list of attributes, and an attribute
name for the new object

remove Removes the specified configuration
object

list Lists all configuration objects of
a given type

list Lists all configuration objects of
a given type, contained

within the scope supplied

show Show all the attributes of a given
configuration object

show Show specified attributes of a
given configuration object

modify Change specified attributes of a
given configuration object

getId Show the configId of an object,
given a string version of

its containment

contents Show the objects which a given
type contains

parents Show the objects which contain
a given type

attributes Show the attributes for a
given type

types Show the possible types for
configuration

help Show help information
```

Examples

- Using Jacl:

  ```
  $Help AdminConfig
  ```

- Using Jython:

  ```
  print Help.AdminConfig()
  ```

AdminControl

Use the **AdminControl** command to view a summary of the help commands and ways to invoke an administrative command.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7027I: The following functions are supported by AdminControl:

getHost returns String representation of connected host

getPort returns String representation of port in use

getType returns String representation of connection type in use

reconnect reconnects with server

queryNames Given ObjectName and QueryExp, retrieves set of ObjectNames that match.

queryNames Given String version of ObjectName, retrieves String of ObjectNames that match.

getMBeanCount returns number of registered beans

getDomainName returns "WebSphere"

getDefaultDomain returns "WebSphere"

getMBeanInfo Given ObjectName, returns MBeanInfo structure for MBean

isInstanceOf Given ObjectName and class name, true if MBean is of that class

isRegistered true if supplied ObjectName is registered

isRegistered true if supplied String version of ObjectName is registered

getAttribute Given ObjectName and name of attribute, returns value of attribute

getAttribute Given String version of ObjectName and name of attribute, returns value of attribute

getAttributes Given ObjectName and array of attribute names, returns AttributeList

getAttributes Given String version of ObjectName and attribute names,
returns String of name value pairs

setAttribute Given ObjectName and Attribute object, set attribute for MBean specified

setAttribute Given String version of ObjectName, attribute name and attribute value,
set attribute for MBean specified

setAttributes Given ObjectName and AttributeList object, set attributes for the MBean specified

invoke Given ObjectName, name of method, array of parameters and signature, invoke method
on MBean specified

invoke Given String version of ObjectName, name of method, String version of parameter list, invoke
method on MBean specified.

invoke Given String version of ObjectName, name of method, String version of parameter list, and
String version of array of signatures, invoke method on MBean specified.

makeObjectName Return an ObjectName built with the given string

completeObjectName Return a String version of an object name given a template name

trace Set the wsadmin trace specification

help Show help information
```

Examples

- Using Jacl:

```
$Help AdminControl
```

- Using Jython:

```
print Help.AdminControl()
```

AdminTask

Use the **AdminTask** command to view a summary of help commands and ways to invoke an administrative command with the AdminTask object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX8001I: The AdminTask object enables the available administrative commands. AdminTask commands
operate in two modes: the default mode is one whichAdminTask communicates with the
 product to accomplish its task. A local mode
is also available in which no server communication takes place. The local mode of operation is invoked by
bringing up the scripting client using the command line "-conntype NONE" option or setting the
"com.ibm.ws.scripting.connectiontype=NONE" property in wsadmin.properties file.

The number of administrative commands varies and depends on your
product installation. Use the following help
 commands to obtain a list of supported commands and their parameters:

help -commands        list all the administrative commands
help -commandGroups     list all the administrative command groups
help commandName       display detailed information for the specified command
help commandName stepName  display detailed information for the specified step belonging to the specified command
help commandGroupName    display detailed information for the specified command group

There are various flavors to invoke an administrative  command. They are

commandName invokes an administrative command that does not require any argument.

commandName targetObject invokes an admin command with the target object string, for example,
the configuration object name of a resource adapter. The expected target object varies with the
administrative command invoked.

Use help command to get information on the target object of an administrative command.

commandName  options invokes an administrative command with the specified option strings. This invocation
 syntax is used to invoke an administrative command that does not require a target object.

It is also used to enter interactive mode if "-interactive" mode is included in the options string.

commandName targetObject options invokes an administrative command with the specified target object
and options strings.

If "-interactive" is included in the options string, then interactive mode is entered.
The target object and options strings vary depending on the admin command invoked.
Use help command to get information on the target object and options.
```

Examples

- Using Jacl:

```
$AdminTask help
```

- Using Jython:

```
print AdminTask.help()
```

all

Use the **all** command to view a summary of the information that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Name: WebSphere:cell=pongo,name=TraceService,mbeanIdentifier=cells/pongo/nodes/pongo/servers/server1/
server.xml#TraceService_1,type=TraceService,node=pongo,process=server1
Description: null
Class name: javax.management.modelmbean.RequiredModelMBean

Attribute                       Type                            Access
ringBufferSize                  int                             RW
traceSpecification              java.lang.String                RW

Operation
int getRingBufferSize()
void setRingBufferSize(int)
java.lang.String getTraceSpecification()
void setTraceState(java.lang.String)
void appendTraceString(java.lang.String)
void dumpRingBuffer(java.lang.String)
void clearRingBuffer()
[Ljava.lang.String; listAllRegisteredComponents()
[Ljava.lang.String; listAllRegisteredGroups()
[Ljava.lang.String; listComponentsInGroup
(java.lang.String)
[Lcom.ibm.websphere.ras.TraceElementState;
getTracedComponents()
[Lcom.ibm.websphere.ras.TraceElementState;
getTracedGroups()
java.lang.String getTraceSpecification(java.
lang.String)
void processDumpString(java.lang.String)
void checkTraceString(java.lang.String)
void setTraceOutputToFile(java.lang.String,
int, int, java.lang.String)
void setTraceOutputToRingBuffer(int, java.
lang.String)
java.lang.String rolloverLogFileImmediate
(java.lang.String, java.lang.String)


Notifications
jmx.attribute.changed

Constructors
```

Examples

- Using Jacl:

```
$Help all [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
```

- Using Jython:

```
print Help.all(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
```

attributes

Use the **attributes** command to view a summary of all the attributes that the MBean defines by name. If you provide the MBean name parameter, the command displays information about the attributes, operations, constructors, description, notifications, and classname of the specified MBean. If you specify the MBean name and attribute name, the command displays information about the specified attribute for the specified MBean.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

attribute name
Specifies the attribute of interest. (String)

Sample output

```
Attribute Type Access

ringBufferSize java.lang.Integer RW

traceSpecification string RW
```

Examples

- Using Jacl:

```
$Help attributes [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
```

- Using Jython:

```
print Help.attributes(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
```

classname

Use the **classname** command to view a class name that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
javax.management.modelmbean.RequiredModelMBean
```

Examples

- Using Jacl:

  ```
  $Help classname [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

  ```
  print Help.classname(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

constructors

Use the **constructors** command to view a summary of all of the constructors that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Constructors
```

Examples

- Using Jacl:

  ```
  $Help constructors [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

  ```
  print Help.constructors(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

description

Use the **description** command to view a description that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name that represents the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Managed object for overall server process.
```

Examples

- Using Jacl:

```
$Help description [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
```

- Using Jython:

```
print Help.description(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
```

help

Use the **help** command to view a summary of all the available methods for the Help object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

```
WASX7028I: The Help object has two purposes:

First, provide general help information for the objects supplied by the wsadmin tool
for scripting: Help, AdminApp, AdminConfig, and AdminControl.

Second, provide a means to obtain interface information about the MBeans that run in
the system.  For this purpose, a variety of commands are available to get
information about the operations, attributes, and other interface information
about particular MBeans.

The following commands are supported by Help; more detailed information about
each of these commands is available by using the "help" command of Help and by
supplying the name of the command as an argument.

attributes     given an MBean, returns help for attributes
operations     given an MBean, returns help for operations
constructors   given an MBean, returns help for constructors
description    given an MBean, returns help for description
notifications  given an MBean, returns help for notifications
classname      given an MBean, returns help for class name
all            given an MBean, returns help for all the previous
help           returns this help text
AdminControl    returns general help text for the AdminControl object
AdminConfig    returns general help text for the AdminConfig object
AdminApp       returns general help text for the AdminApp object
AdminTask      returns general help text for the AdminTask object
wsadmin        returns general help text for the wsadmin script launcher
message        given a message ID, returns an explanation and a user action
```

Examples

- Using Jacl:

```
$Help help
```

- Using Jython:

```
print Help.help()
```

message

Use the **message** command to view information for a message ID.

Target object

None.

Required parameters

message ID
Specifies the message ID of the message of interest. (String)

Optional parameters

None.

Sample output

```
Explanation: The container was unable to
passivate an enterprise bean due to
exception {2}
User action: Take action based upon
message in exception {2}
```

Examples

- Using Jacl:

  ```
  $Help message CNTR0005W
  ```

- Using Jython:

  ```
  print Help.message('CNTR0005W')
  ```

notifications

Use the **notifications** command to view a summary of all the notifications that the MBean defines by name.

Target object

None.

Required parameters

MBean name
Specifies the object name of the MBean of interest. (String)

Optional parameters

None.

Sample output

```
Notification

websphere.messageEvent.audit

websphere.messageEvent.fatal

websphere.messageEvent.error

websphere.seriousEvent.info
```

```
websphere.messageEvent.warning

jmx.attribute.changed
```

Examples

- Using Jacl:

  ```
  $Help notifications [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
  ```

- Using Jython:

  ```
  print Help.notifications(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
  ```

operations

Use the **operations** command with the MBean name parameter to view a summary of all the operations that the MBean defines by name. Specify a value for the MBean name and operation name to display the signature of the operation for the MBean that is defined by name.

Target object

None.

Required parameters

MBean name
Specifies the object name of the MBean of interest. (String)

Optional parameters

operation name
Specifies the operation of interest. (String)

Sample output

The command returns output that is similar to the following example if you specify only the MBean name parameter:

```
Operation
int getRingBufferSize()
void setRingBufferSize(int)
java.lang.String getTraceSpecification()
void setTraceState(java.lang.String)
void appendTraceString(java.lang.String)
void dumpRingBuffer(java.lang.String)
void clearRingBuffer()
[Ljava.lang.String; listAllRegisteredComponents()
[Ljava.lang.String; listAllRegisteredGroups()
[Ljava.lang.String; listComponentsInGroup(java.lang.String)
[Lcom.ibm.websphere.ras.TraceElementState; getTracedComponents()
[Lcom.ibm.websphere.ras.TraceElementState; getTracedGroups()
java.lang.String getTraceSpecification(java.lang.String)
void processDumpString(java.lang.String)
void checkTraceString(java.lang.String)
void setTraceOutputToFile(java.lang.String, int, int, java.lang.String)
void setTraceOutputToRingBuffer(int, java.lang.String)
java.lang.String rolloverLogFileImmediate(java.lang.String, java.lang.String)
```

The command returns output that is similar to the following example if you specify the MBean name and operation name parameters:

```
void processDumpString(string)

Description: Write the contents of the Ras
services ring buffer to the specified file.
```

```
Parameters:

Type         string
Name         dumpString
Description  A String in the specified format to process or null.
```

Examples

- Using Jacl:

```
$Help operations [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
$Help operations [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]
 processDumpString
```

- Using Jython:

```
print Help.operations(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))
print Help.operations(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'),
'processDumpString')
```

Commands for the AdminConfig object using wsadmin scripting

Use the AdminConfig object to invoke configuration commands and to create or change elements of the WebSphere Application Server configuration, for example, creating a data source.

You can start the scripting client without a running server, if you only want to use local operations. To run in local mode, use the -conntype NONE option to start the scripting client. You receive a message that you are running in the local mode. If a server is currently running, do not run the AdminConfig tool in local mode. Configuration changes that are made in local mode are not be reflected in the running server configuration. If you save a conflicting configuration, you could corrupt the configuration.

Note: You can use the Jython list or string syntax to pass parameters to a wsadmin command. If you have a parameter that includes a comma as a character, though, you need to use the Jython string syntax to pass the parameters.

To use the create command, for example, you could enter command similar to the following:

```
params='[[name name1] [nameInNameSpace nameSpace_Name] [string_to_bind "value, withComma"]]'
AdminConfig.create(type, parent, params)
```

You can also use the modify command:

```
AdminConfig.modify(type, params)
```

The following commands are available for the AdminConfig object:

attributes

Use the attributes command to return a list of the top level attributes for a given type.

Target object

None.

Required parameters

object type
Specifies the name of the object type that is based on the XML configuration files. The object type does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

```
"properties Property*" "serverSecurity ServerSecurity"
"server Server@" "id Long" "stateManagement StateManageable"
"name String" "moduleVisibility EEnumLiteral(MODULE,
COMPATIBILITY, SERVER, APPLICATION)" "services Service*"
"statisticsProvider StatisticsProvider"
```

Examples

- Using Jacl:

  ```
  $AdminConfig attributes ApplicationServer
  ```

- Using Jython:

  ```
  print AdminConfig.attributes('ApplicationServer')
  ```

checkin

Use the checkin command to check a file into the configuration repository that is described by the document Uniform Resource Identifier (URI). This method only applies to deployment manager configurations.

Target object

None.

Required parameters

URI
The document URI is relative to the root of the configuration repository, for example:

- `/WebSphere/AppServer/config`

file name
Specifies the name of the source file to check in.

opaque object
Specifies an object that the extract command of the AdminConfig object returns by a prior call.

Optional parameters

None.

Sample output

```
"properties Property*" "serverSecurity ServerSecurity"
"server Server@" "id Long" "stateManagement StateManageable"
"name String" "moduleVisibility EEnumLiteral(MODULE,
COMPATIBILITY, SERVER, APPLICATION)" "services Service*"
"statisticsProvider StatisticsProvider"
```

Examples

- Using Jacl:

  ```
  $AdminConfig checkin cells/MyCell/Node/MyNode/serverindex.xml /mydir/myfile $obj
  ```

- Using Jython:

  ```
  print AdminConfig.checkin('cells/MyCell/Node/MyNode/serverindex.xml', '/mydir/myfile',  obj)
  ```

convertToCluster

Use the convertToCluster command to convert a server so that it is the first member of a new server cluster.

Target object

None.

Required parameters

server ID
The configuration ID of the server of interest.

cluster name
Specifies the name of the cluster of interest.

Optional parameters

None.

Sample output

```
myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)
```

Examples

- Using Jacl:

```
set serverid [$AdminConfig getid /Server:myServer/]
$AdminConfig convertToCluster $serverid myCluster
```

- Using Jython:

```
serverid = AdminConfig.getid('/Server:myServer/')
print AdminConfig.convertToCluster(serverid, 'myCluster')
```

create

Use the create command to create configuration objects.

Target object

None.

Required parameters

type
Specifies the name of the object type that is based on the XML configuration files. This parameter value does not have to be the same name that the administrative console displays.

parent ID
Specifies the configuration ID of the parent object.

attributes
Specifies any attributes to add to the configuration ID.

Optional parameters

None.

Sample output

This command returns a string of the configuration object name, as this sample output displays:

```
ds1(cells/mycell/nodes/DefaultNode/servers/server1|resources.xml#DataSource_6)
```

Examples

- Using Jacl:

```
set jdbc1 [$AdminConfig getid /JDBCProvider:jdbc1/]
$AdminConfig create DataSource $jdbc1 {{name ds1}}
```

- Using Jython string attributes:

```
jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/')
print AdminConfig.create('DataSource', jdbc1, '[[name ds1]]')
```

- Using Jython with object attributes:

```
jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/')
print AdminConfig.create('DataSource', jdbc1, [['name', 'ds1']])
```

createClusterMember

Use the createClusterMember command to create a new server object on the node that the node id parameter specifies. This server is created as a new member of the existing cluster that is specified by the cluster id parameter, and contains attributes that are specified in the member attributes parameter. The server is created using the server template that is specified by the template id attribute, and that contains the name specified by the memberName attribute. The memberName attribute is required. The template options are available only for the first cluster member that you create. All cluster members that you create after the first member will be identical.

Target object

None.

Required parameters

cluster ID
Specifies the configuration ID of the cluster of interest.

node ID
Specifies the configuration ID of the node of interest.

template ID
Specifies the template ID to use to create the server.

member attributes
Specifies any attributes to add to the cluster member. The memberName attribute is required, and defines the name of the cluster member to create.

Optional parameters

None.

Sample output

This command returns the configuration ID of the newly created cluster member, as the following example displays:

```
myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)
```

Examples

- Using Jacl:

```
set clid [$AdminConfig getid /ServerCluster:myCluster/]
set nodeid [$AdminConfig getid /Node:mynode/]
$AdminConfig createClusterMember $clid $nodeid {{memberName newMem1} {weight 5}}
```

- Using Jython string attributes:

```
clid = AdminConfig.getid('/ServerCluster:myCluster/')
nodeid = AdminConfig.getid('/Node:mynode/')
print AdminConfig.createClusterMember(clid, nodeid, '[[memberName newMem1] [weight 5]]')
```

- Using Jython with object attributes:

```
clid = AdminConfig.getid('/ServerCluster:myCluster/')
nodeid = AdminConfig.getid('/Node:mynode/')
print AdminConfig.createClusterMember(clid, nodeid, [['memberName', 'newMem1'], ['weight', 5]])
```

createDocument

Use the createDocument command to create a new document in the configuration repository.

Target object

None.

Required parameters

document URI
Specifies the name of the document to create in the repository.

file name
Specifies a valid local file name of the document to create.

Optional parameters

None.

Examples

- Using Jacl:
  ```
  $AdminConfig createDocument cells/mycell/myfile.xml /mydir/myfile
  ```
- Using Jython with string attributes:
  ```
  AdminConfig.createDocument('cells/mycell/myfile.xml', '/mydir/myfile')
  ```

createUsingTemplate

Use the createUsingTemplate command to create a type of object with the given parent, using a template. You can only use this command for creation of a server with APPLICATION_SERVER type. If you want to create a server with a type other than APPLICATION_SERVER, use the createGenericServer or the createWebServer command.

Target object

None.

Required parameters

type
Specifies the type of object to create.

parent
Specifies the configuration ID of the parent.

template
Specifies a configuration ID of an existing object. This object can be a template object returned by using the listTemplates command, or any other existing object of the correct type.

Optional parameters

attributes
Specifies attribute values for the object. The attributes specified using this parameter override the settings in the template.

Sample output

The command returns the configuration ID of the new object, as the following example displays:

```
myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)
```

Examples

- Using Jacl:

```
set node [$AdminConfig getid /Node:mynode/]
set templ [$AdminConfig listTemplates JDBCProvider "DB2 JDBC Provider (XA)"]
$AdminConfig createUsingTemplate JDBCProvider $node {{name newdriver}} $templ
```

- Using Jython with string attributes:

```
node = AdminConfig.getid('/Node:mynode/')
templ = AdminConfig.listTemplates('JDBCProvider', "DB2 JDBC Provider (XA)")
print AdminConfig.createUsingTemplate('JDBCProvider', node, '[[name newdriver]]', templ)
```

- Using Jython with object attributes:

```
node = AdminConfig.getid('/Node:mynode/')
templ = AdminConfig.listTemplates('JDBCProvider', "DB2 JDBC Provider (XA)")
print AdminConfig.createUsingTemplate('JDBCProvider', node, [['name', 'newdriver']], templ)
```

defaults

Use the defaults command to display the default values for attributes of a given type. This method displays all of the possible attributes contained by an object of a specific type. If the attribute has a default value, this method also displays the type and default value for each attribute.

Target object

None.

Required parameters

type
Specifies the type of object to return. The name of the object type that you specify is based on the XML configuration files. This name does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

The command returns string that contains a list of attributes with its type and value, as the following example displays:

```
Attribute               Type     Default

usingMultiRowSchema     Boolean  false
maxInMemorySessionCount Integer  1000
allowOverflow           Boolean  true
scheduleInvalidation    Boolean  false
writeFrequency          ENUM
writeInterval           Integer  120
writeContents           ENUM
invalidationTimeout     Integer  30
invalidationSchedule    InvalidationSchedule
```

Examples

- Using Jacl:

```
$AdminConfig defaults TuningParams
```

- Using Jython:

```
print AdminConfig.defaults('TuningParams')
```

deleteDocument

Use the deleteDocument command to delete a document from the configuration repository.

Target object

None.

Required parameters

documentURI
Specifies the document to delete from the repository.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminConfig deleteDocument cells/mycell/myfile.xml
```

- Using Jython:

```
AdminConfig.deleteDocument('cells/mycell/myfile.xml')
```

existsDocument

Use the existsDocument command to test for the existence of a document in the configuration repository.

Target object

None.

Required parameters

documentURI
Specifies the document to test for in the repository.

Optional parameters

None.

Sample output

The command returns a true value if the document exists, as the following example displays:

```
1
```

Examples

- Using Jacl:

```
$AdminConfig existsDocument cells/mycell/myfile.xml
```

- Using Jython:

```
print AdminConfig.existsDocument('cells/mycell/myfile.xml')
```

extract

Use the extract command to extract a configuration repository file that is described by the document URI and places it in the file named by `filename`. This method only applies to deployment manager configurations.

Target object

None.

Required parameters

documentURI
Specifies the document to extract from the configuration repository. The document URI must exist in the repository. The document URI is relative to the root of the configuration repository, for example:

- `/WebSphere/AppServer/config`

filename
Specifies the filename to extract the document to. The filename must be a valid local filename where the contents of the document are written. If the file that is specified by the filename parameter exists, the extracted file replaces it.

Optional parameters

None.

Sample output

The command returns an opaque "digest" object which should be used to check the file back in using the checkin command.

Examples

- Using Jacl:

```
set obj [$AdminConfig extract cells/MyCell/nodes/MyNode/serverindex.xml /mydir/myfile]
```

- Using Jython:

```
obj = AdminConfig.extract('cells/MyCell/nodes/MyNode/serverindex.xml','/mydir/myfile')
```

getCrossDocumentValidationEnabled

Use the getCrossDocumentValidationEnabled command to return a message with the current cross-document enablement setting. This method returns true if cross-document validation is enabled.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns string that contains the message with the cross-document validation setting, as the following example displays:

```
WASX7188I: Cross-document validation enablement set to true
```

Examples

- Using Jacl:

  ```
  $AdminConfig getCrossDocumentValidationEnabled
  ```

- Using Jython:

  ```
  print AdminConfig.getCrossDocumentValidationEnabled()
  ```

getid

Use the getid command to return the configuration ID of an object.

Target object

None.

Required parameters

containment path
Specifies the containment path of interest.

Optional parameters

None.

Sample output

The command returns configuration ID for an object that is described by the containment path, as the following example displays:

```
Db2JdbcDriver(cells/testcell/nodes/testnode|resources.xml#JDBCProvider_1)
```

Examples

- Using Jacl:

  ```
  $AdminConfig getid /Cell:testcell/Node:testNode/JDBCProvider:Db2JdbcDriver/
  ```

- Using Jython:

  ```
  print AdminConfig.getid('/Cell:testcell/Node:testNode/JDBCProvider:Db2JdbcDriver/')
  ```

getObjectName

Use the getObjectName command to return a string version of the object name for the corresponding running MBean. This method returns an empty string if no corresponding running MBean exists.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object name to return.

Optional parameters

None.

Sample output

The command returns a string that contains the object name, as the following example displays:

```
WebSphere:cell=mycell,name=server1,mbeanIdentifier=cells/mycell/nodes/mynode/servers/server1/
server.xml#Server_1,type=Server,node=mynode,process=server1,processType=UnManagedProcess
```

Examples

- Using Jacl:

```
set server [$AdminConfig getid /Node:mynode/Server:server1/]
$AdminConfig getObjectName $server
```

- Using Jython:

```
server = AdminConfig.getid('/Node:mynode/Server:server1/')
print AdminConfig.getObjectName(server)
```

getObjectType

Use the getObjectType command to display the object type for the object configuration ID of interest.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object name to return.

Optional parameters

None.

Examples

- Using Jacl:

```
set server [$AdminConfig getid /Node:mynode/Server:server1/]
$AdminConfig getObjectType $server
```

- Using Jython:

```
server = AdminConfig.getid('/Node:mynode/Server:server1/')
print AdminConfig.getObjectType(server)
```

getSaveMode

Use the getSaveMode command to return the mode that is used when you invoke a save command. The command returns one of the following possible values:

- overwriteOnConflict - Saves changes even if they conflict with other configuration changes
- rollbackOnConflict - Fails a save operation if changes conflict with other configuration changes. This value is the default.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the current save mode setting, as the following example displays:

```
rollbackOnConflict
```

Examples

- Using Jacl:

  ```
  $AdminConfig getSaveMode
  ```

- Using Jython:

  ```
  print AdminConfig.getSaveMode()
  ```

getValidationLevel

Use the getValidationLevel command to return the validation used when files are extracted from the repository.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the validation level, as the following example displays:

```
WASX7189I: Validation level set to HIGH
```

Examples

- Using Jacl:

  ```
  $AdminConfig getValidationLevel
  ```

- Using Jython:

  ```
  print AdminConfig.getValidationLevel()
  ```

getValidationSeverityResult

Use the getValidationSeverityResult command to return the number of validation messages with the given severity from the most recent validation.

Target object

None.

Required parameters

severity
Specifies which severity level for which to return the number of validation messages. Specify an integer value between 0 and 9.

Optional parameters

None.

Sample output

The command returns a string that indicates the number of validation messages of the given severity, as the following example displays:

```
16
```

Examples

- Using Jacl:

  ```
  $AdminConfig getValidationSeverityResult 1
  ```

- Using Jython:

  ```
  print AdminConfig.getValidationSeverityResult(1)
  ```

hasChanges

Use the hasChanges command to determine if unsaved configuration changes exist.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns `1` if unsaved configuration changes exist or `0` if unsaved configuration changes do not exist, as the following example displays:

```
1
```

Examples

- Using Jacl:

  ```
  $AdminConfig hasChanges
  ```

- Using Jython:

  ```
  print AdminConfig.hasChanges()
  ```

help

Use the help command to display static help information for the AdminConfig object.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of options for the help command, as the following example displays:

```
WASX7053I: The AdminConfig object communicates with the configuration service in a product to manipulate
configuration data for an Application Server installation.  The AdminConfig object has commands to list,
create, remove, display, and modify configuration data, as well as commands to display information about
configuration data types.

Most of the commands supported by the AdminConfig object operate in two modes: the default mode is one in which
the AdminConfig object communicates with the Application Server to accomplish its tasks. A local mode is also
possible, in which no server communication takes place.  The local mode of operation is invoked by bringing up
the scripting client without a server connected using the command line "-conntype NONE" option or setting the
"com.ibm.ws.scripting.connectionType=NONE" property in the wsadmin.properties file.

The following commands are supported by the AdminConfig object; more detailed information about each of these commands is
available by using the help command of the AdminConfig object and by supplying the name of the command as an argument.

attributes               Shows the attributes for a given type
checkin                  Checks a file into the configuration repository.
convertToCluster         Converts a server to be the first member of a new server cluster
create                   Creates a configuration object, given a type, a parent, and a list of attributes, and
               optionally an attribute name for the new object
createClusterMember        Creates a new server that is a member of an existing cluster.
createDocument           Creates a new document in the configuration repository.
installResourceAdapter       Installs a J2C resource adapter with the given RAR file name and an option string in the node.
createUsingTemplate        Creates an object using a particular template type.
defaults                 Displays the default values for the attributes of a given type.
deleteDocument           Deletes a document from the configuration repository.
existsDocument           Tests for the existence of a document in the configuration repository.
extract                  Extracts a file from the configuration repository.
getCrossDocumentValidationEnabled  Returns true if cross-document validation is enabled.
getid                    Show the configuration ID of an object, given a string version of its containment
getObjectName            Given a configuration ID, returns a string version of the ObjectName
               for the corresponding running MBean, if any.
getSaveMode              Returns the mode used when "save" is invoked
getValidationLevel        Returns the validation that is used when files are extracted from the repository.
getValidationSeverityResult    Returns the number of messages of a given severity from the most recent validation.
hasChanges               Returns true if unsaved configuration changes exist
help                     Shows help information
list                     Lists all the configuration objects of a given type
listTemplates            Lists all the available configuration templates of a given type.
modify                   Changes the specified attributes of a given configuration object
parents                  Shows the objects which contain a given type
queryChanges             Returns a list of unsaved files
remove                   Removes the specified configuration object
required                 Displays the required attributes of a given type.
reset                    Discards the unsaved configuration changes
save                     Commits the unsaved changes to the configuration repository
setCrossDocumentValidationEnabled  Sets the cross-document validation enabled mode.
setSaveMode              Changes the mode used when "save" is invoked
setValidationLevel        Sets the validation used when files are extracted from the repository.
show                     Shows the attributes of a given configuration object
showall                  Recursively shows the attributes of a given configuration
                         object, and all the objects that are contained within each attribute.
showAttribute            Displays only the value for the single attribute that is specified.
types                    Shows the possible types for configuration
validate                 Invokes validation
```

Examples

- Using Jacl:

  ```
  $AdminConfig help
  ```

- Using Jython:

  ```
  print AdminConfig.help()
  ```

installResourceAdapter

Use the installResourceAdapter command to install a Java 2 Connector (J2C) resource adapter with the given Resource Adapter Archive (RAR) file name and an option string in the node. When you edit the installed application with the embedded RAR, only existing J2C connection factory, J2C activation specs, and J2C administrative objects will be edited. No new J2C objects will be created.

Target object

None.

Required parameters

node
Specifies the node of interest.

RAR file name
Specifies the fully qualified file name of the RAR file that resides in the node that you specify.

Optional parameters

options
Specifies additional options for installing a resource adapter. The valid options include the following options:

- rar.name
- rar.desc
- rar.archivePath
- rar.classpath
- rar.nativePath
- rar.threadPoolAlias
- rar.propertiesSet

The rar.name option is the name for the J2C resource adapter. If you do not specify this option, the display name in the RAR deployment descriptor is used. If that name is not specified, the RAR file name is used. The rar.desc option is a description of the J2CResourceAdapter.

The rar.archivePath is the name of the path where you extract the file. If you do not specify this option, the archive is extracted to the $\{CONNECTOR_INSTALL_ROOT\} directory. The rar.classpath option is the additional class path.

rar.propertiesSet is constructed with the following:

```
name String
value String
type String
*desc String
*required true/false
* means the item is optional
```

Each attribute of the property are specified in a set of {}. A property is specified in a set of {}. You can specify multiple properties in {}.

Sample output

The command returns the configuration ID of the new J2CResourceAdapter object:

```
myResourceAdapter(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
```

Examples

- Using Jacl:

```
$AdminConfig installResourceAdapter /rar/mine.rar mynode{-rar.name myResourceAdapter
-rar.desc "My rar file"}
```

- Using Jython:

```
print AdminConfig.installResourceAdapter('/rar/mine.rar', 'mynode', '[-rar.name myResourceAdapter
-rar.desc "My rar file"]')
```

To add a String

```
resourceProperties (name=myName,value=myVal)
```

into the resource adapter configuration, run the following commands:

1.
```
pSet = [['propertySet',[['resourceProperties',[[['name','myName'], ['type', 'String'],
['value','myVal']]]]]]]
```
2.
```
myRA =AdminConfig.installResourceAdapter('/query.rar','mynodeCellManager05',
['-rar.desc','mydesc'])
```
3.
```
AdminConfig.modify(myRA,pSet)
```

list

Use the list command to return a list of objects of a given type, possibly scoped by a parent. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Required parameters

object type
Specifies the name of the object type. The name of the object type is based on the XML configuration files and does not have to be the same name that the administrative console displays.

pattern
Specifies additional search query information using wildcard characters of Java regular expressions.

Optional parameters

None.

Sample output

The command returns a list of objects:

```
Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xml#JDBCProvider_1)
Db2JdbcDriver(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#JDBCProvider_1)
Db2JdbcDriver(cells/mycell/nodes/DefaultNode/servers/nodeAgent|resources.xml#JDBCProvider_1)
```

Examples

The following examples list each JDBC provider configuration object:

- Using Jacl:

```
$AdminConfig list JDBCProvider
```

- Using Jython:

```
print AdminConfig.list('JDBCProvider')
```

The following examples list each JDBC provider configuration object that begin with the `derby` string:

- Using Jacl:

```
$AdminConfig list JDBCProvider derby*
```

- Using Jython:

```
print AdminConfig.list('JDBCProvider', 'derby*')
```

You can use regular Java expression patterns and wildcard patterns to specify command name for $AdminConfig list, types and listTemplates functions.

The following examples list the configuration objects of type `server` starting from `server1`:

- Using Jacl and regular Java expression patterns:

```
$AdminConfig list Server server1.*
```

- Using Jacl and wildcard patterns:

```
$AdminConfig list Server server1*
```

- Using Jython and regular Java expression patterns::

```
print AdminConfig.list("Server", "server1.*")
```

- Using Jython and wildcard patterns:

```
print AdminConfig.list("Server", "server1*")
```

The following examples list each find configuration object of that starts with `SSLConfig`:

- Using Jacl and regular Java expression patterns:

```
$AdminConfig types SSLConfig.*
```

- Using Jacl and wildcard patterns:

```
$AdminConfig types SSLConfig*
```

- Using Jython and regular Java expression patterns:

```
print AdminConfig.types("SSLConfig.*")
```

- Using Jython and wildcard patterns:

```
print AdminConfig.types("SSLConfig*")
```

listTemplates

Use the listTemplates command to display a list of template object IDs. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Required parameters

object type
Specifies the name of the object type. The name of the object type is based on the XML configuration files and does not have to be the same name that the administrative console displays.

pattern
Specifies additional search query information using wildcard characters of Java regular expressions.

Optional parameters

None.

Sample output

The example displays a list of all the JDBCProvider templates that are available on the system:

```
"Cloudscape JDBC Provider (XA)(templates/servertypes/APPLICATION_SERVER/servers/defaultZOS_60X|resources.xml#builtin_jdbcprovider)"
"Cloudscape JDBC Provider (XA)(templates/servertypes/APPLICATION_SERVER/servers/default_60X|resources.xml#builtin_jdbcprovider)"
"Cloudscape JDBC Provider (XA)(templates/servertypes/PROXY_SERVER/servers/proxy_server_60X|resources.xml#builtin_jdbcprovider)"
"Cloudscape JDBC Provider (XA)(templates/servertypes/PROXY_SERVER/servers/proxy_server_foundation_zos|resources.xml#builtin_jdbcprovider)"
"Cloudscape JDBC Provider (XA)(templates/servertypes/PROXY_SERVER/servers/proxy_server_foundation|resources.xml#builtin_jdbcprovider)"
"Cloudscape JDBC Provider (XA)(templates/servertypes/PROXY_SERVER/servers/proxy_server_zos_60X|resources.xml#builtin_jdbcprovider)"
"Cloudscape JDBC Provider (XA)(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_4)"
"Cloudscape JDBC Provider Only (XA)(templates/system|jdbc-resource-provider-only-templates.xml#JDBCProvider_db2j_4)"
"Cloudscape JDBC Provider Only(templates/system|jdbc-resource-provider-only-templates.xml#JDBCProvider_db2j_3)"
"Cloudscape JDBC Provider(templates/servertypes/APPLICATION_SERVER/servers/defaultZOS_5X|resources.xml#JDBCProvider_1)"
"Cloudscape JDBC Provider(templates/servertypes/APPLICATION_SERVER/servers/default_5X|resources.xml#JDBCProvider_1)"
"Cloudscape JDBC Provider(templates/system|jdbc-resource-provider-templates.xml#JDBCProvider_db2j_3)"
```

Examples

The following examples return each JDBC provider template:

- Using Jacl:

  ```
  $AdminConfig listTemplates JDBCProvider
  ```

- Using Jython:

  ```
  print AdminConfig.listTemplates('JDBCProvider')
  ```

The following examples return each JDBC provider template that begins with the `sybase` string:

- Using Jacl:

  ```
  $AdminConfig listTemplates JDBCProvider sybase*
  ```

- Using Jython:

  ```
  print AdminConfig.listTemplates('JDBCProvider sybase*')
  ```

modify

Use the modify command to support the modification of object attributes.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object to modify.

attributes
Specifies the attributes to modify for the configuration ID of interest.

Optional parameters

None.

Examples

- Using Jacl:

  ```
  $AdminConfig modify ConnFactory1(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#
  GenericJMSConnectionFactory_1) {{userID newID} {password newPW}}
  ```

- Using Jython with string attributes:

  ```
  AdminConfig.modify('ConnFactory1(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#
  GenericJMSConnectionFactory_1)', '[[userID newID] [password newPW]]')
  ```

- Using Jython with object attributes:

  ```
  AdminConfig.modify('ConnFactory1(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#
  GenericJMSConnectionFactory_1)', [['userID', 'newID'], ['password', 'newPW']])
  ```

parents

Use the parents command to obtain information about object types.

Target object

None.

Required parameters

object type
Specifies the object type of interest. The name of the object type is based on the XML configuration files and does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

The example displays a list of object types:

```
Cell
Node
Server
```

Examples

- Using Jacl:

  ```
  $AdminConfig parents JDBCProvider
  ```

- Using Jython:

  ```
  print AdminConfig.parents('JDBCProvider')
  ```

queryChanges

Use the queryChanges command to return a list of unsaved configuration files.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The example displays a string that contains a list of files with unsaved changes:

```
WASX7146I: The following configuration files contain unsaved changes:
cells/mycell/nodes/mynode/servers/server1|resources.xml
```

Examples

- Using Jacl:

  ```
  $AdminConfig queryChanges
  ```

- Using Jython:

```
print AdminConfig.queryChanges()
```

remove

Use the remove command to remove a configuration object.

Target object

None.

Required parameters

configuration ID
Specifies the configuration object of interest.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminConfig remove ds1(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_6)
```

- Using Jython:

```
AdminConfig.remove('ds1(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_6)')
```

required

Use the required command to display the required attributes that are contained by an object of a certain type.

Target object

None.

Required parameters

type
Specifies the object type for which to display the required attributes. The name of the object type is based on the XML configuration files. It does not have to be the same name that the administrative console displays.

Optional parameters

None.

Sample output

The example displays a string that contains a list of the required attributes with its type:

```
Attribute                   Type
streamHandlerClassName      String
protocol                    String
```

Examples

- Using Jacl:

```
$AdminConfig required URLProvider
```

- Using Jython:

```
print AdminConfig.required('URLProvider')
```

reset

Use the reset command to reset the temporary workspace that holds updates to the configuration.

Target object

None.

Required parameters

None.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminConfig reset
```

- Using Jython:

```
AdminConfig.reset()
```

resetAttributes

Use the resetAttributes command to reset specific attributes for the configuration object of interest.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the configuration object of interest.

attributes
Specifies the attribute to reset and the value to which the attribute is reset.

Optional parameters

None.

Examples

- Using Jacl:

```
set ds [$AdminConfig getid /DataSource:myDS]
$AdminConfig resetAttributes $ds {{"classpath" "c:/temp/testing;c:/temp/test"}}
```

- Using Jython:

```
ds = AdminConfig.getid("/DataSource:myDS")
AdminConfig.resetAttributes(ds, [["classpath", "c:/temp/testing;c:/temp/test"]])
```

save

Use the save command to save changes to the configuration repository.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The save command does not return output.

Examples

- Using Jacl:

```
$AdminConfig save
```

- Using Jython:

```
AdminConfig.save()
```

setCrossDocumentValidationEnabled

Use the setCrossDocumentValidationEnabled command to set the cross-document validation enabled mode. Values include `true` or `false`.

Target object

None.

Required parameters

`flag`
Specifies whether cross-document validation is enabled or disabled. Specify `true` to enable or `false` to disable cross-document validation.

Optional parameters

None.

Sample output

The command returns a status statement for cross-document validation, as the following example displays:

```
WASX7188I: Cross-document validation enablement set to true
```

Examples

- Using Jacl:

```
$AdminConfig setCrossDocumentValidationEnabled true
```

- Using Jython:

```
print AdminConfig.setCrossDocumentValidationEnabled('true')
```

setSaveMode

Use the setSaveMode command to modify the behavior of the save command.

Target object

None.

Required parameters

save mode
Specifies the save mode to use. The default value is `rollbackOnConflict`. When the system discovers a conflict while saving, the unsaved changes are not committed. The alternative value is `overwriteOnConflict`, which saves the changes to the configuration repository even if conflicts exist. To use `overwriteOnConflict` as the value of this command, the deployment manager must be enabled for configuration overwrite.

Optional parameters

None.

Sample output

The setSaveMode command does not return output.

Examples

- Using Jacl:
  ```
  $AdminConfig setSaveMode overwriteOnConflict
  ```
- Using Jython:
  ```
  AdminConfig.setSaveMode('overwriteOnConflict')
  ```

setValidationLevel

Use the setValidationLevel command to set the validation that is used when files are extracted from the repository.

Target object

None.

Required parameters

level
Specifies the validation to use. Five validation levels are available: `none`, `low`, `medium`, `high`, or `highest`.

Optional parameters

None.

Sample output

The command returns a string that contains the validation level setting, as the following example displays:

```
WASX7189I: Validation level set to HIGH
```

Examples

- Using Jacl:

```
$AdminConfig setValidationLevel high
```

- Using Jython:

```
print AdminConfig.setValidationLevel('high')
```

show

Use the show command to return the top-level attributes of the given object.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object of interest.

Optional parameters

None.

Sample output

The command returns a string that contains the attribute value, as the following example displays:

```
[name "Sample Datasource"] [description "Data source for the Sample entity beans"]
```

Examples

- Using Jacl:

```
$AdminConfig show Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xmlJDBCProvider_1)
```

- Using Jython:

```
print AdminConfig.show('Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xmlJDBCProvider_1)')
```

showall

Use the showall command to recursively show the attributes of a given configuration object.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object of interest.

Optional parameters

None.

Sample output

The command returns a string that contains the attribute value, as the following examples show:

Using Jacl:

```
tcpNoDelay: null
SoTimeout: 0
bytesRead: 6669
```

```
{authMechanismPreference BASIC_PASSWORD}
{connectionPool {{agedTimeout 0}
{connectionTimeout 180}
{freePoolDistributionTableSize 0}
{maxConnections 10}
{minConnections 1}
{numberOfFreePoolPartitions 0}
{numberOfSharedPoolPartitions 0}
{numberOfUnsharedPoolPartitions 0}
{properties {}}
{purgePolicy EntirePool}
{reapTime 180}
{stuckThreshold 0}
{stuckTime 0}
{stuckTimerTime 0}
{surgeCreationInterval 0}
{surgeThreshold -1}
{testConnection false}
{testConnectionInterval 0}
{unusedTimeout 1800}}}
{datasourceHelperClassname com.ibm.websphere.rsadapter.DerbyDataStoreHelper}
{description "Datasource for the WebSphere Default Application"}
{diagnoseConnectionUsage false}
{jndiName DefaultDatasource}
{logMissingTransactionContext true}
{manageCachedHandles false}
{name "Default Datasource"}
{properties {}}
{propertySet {{resourceProperties {{{name databaseName}
{required false}
{type java.lang.String}
{value ${APP_INSTALL_ROOT}/${CELL}/DefaultApplication.ear/DefaultDB}} {{name shu
tdownDatabase}
{required false}
{type java.lang.String}
{value {}}} {{name dataSourceName}
{required false}
{type java.lang.String}
{value {}}} {{name description}
{required false}
{type java.lang.String}
{value {}}} {{name connectionAttributes}
{required false}
{type java.lang.String}
{value upgrade=true}} {{name createDatabase}
{required false}
{type java.lang.String}
{value {}}}}}}}
{provider "Derby JDBC Provider(cells/isthmusCell04/nodes/isthmusNode14/servers/s
erver1|resources.xml#JDBCProvider_1183122153343)"}
{providerType "Derby JDBC Provider"}
{relationalResourceAdapter "WebSphere Relational Resource Adapter(cells/isthmusC
ell04/nodes/isthmusNode14/servers/server1|resources.xml#builtin_rra)"}
{statementCacheSize 10}
```

Using Jython:

```
[datasourceHelperClassname com.ibm.websphere.rsadapter.DerbyDataStoreHelper]
[description "Datasource for the WebSphere Default Application"]
[jndiName DefaultDatasource]
[name "Default Datasource"]
[propertySet [[resourceProperties [[[description "Location of Apache Derby default database."]
[name databaseName]
[type string]
[value ${WAS_INSTALL_ROOT}/bin/DefaultDB]] [[name remoteDataSourceProtocol]
[type string]
[value []]] [[name shutdownDatabase]
[type string]
[value []]] [[name dataSourceName]
[type string]
[value []]] [[name description]
[type string]
[value []]] [[name connectionAttributes]
[type string]
[value []]] [[name createDatabase]
[type string]
[value [[[[[[]]]]]]]
[provider "Apache Derby JDBC Driver(cells/pongo/nodes/pongo/servers/server1|resources.xml#JDBCProvider_1)"]
[relationalResourceAdapter "WebSphere Relational Resource Adapter(cells/pongo/nodes/pongo/servers/server1|
resources.xml#builtin_rra)"]
[statementCacheSize 0]
```

You might have to convert the Jython output from a string to a list for further processing.

Examples

- Using Jacl:

```
$AdminConfig showall "Default Datasource(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_1)"
```

- Using Jython:

```
print AdminConfig.showall
  ("Default Datasource(cells/mycell/nodes/DefaultNode/servers/server1:resources.xml#DataSource_1)")
```

showAttribute

Use the showAttribute command to display only the value for the single attribute that you specify.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the object of interest.

attribute
Specifies the attribute to query.

Optional parameters

None.

Sample output

The output of this command is different from the output of the show command when a single attribute is specified. The showAttribute command does not display a list that contains the attribute name and value. It only displays the attribute value, as the following example displays:

```
mynode
```

Examples

- Using Jacl:

```
set ns [$AdminConfig getid /Node:mynode/]
$AdminConfig showAttribute $ns hostName
```

- Using Jython:

```
ns = AdminConfig.getid('/Node:mynode/')
print AdminConfig.showAttribute(ns, 'hostName')
```

Prior to Version 7.0.0.5, the Jython scripting language does not recognize special characters. In addition, when the comma and single space characters occur between attribute parameters, these characters are treated as deliminators and ignored when the attribute value is saved. For example, you might have the following set of Jython commands:

```
value='{"param1","param2"}'
serverId=AdminConfig.getid('/Cell:cell_name/Node:node_name/Server:server_name')
nameSpace=AdminConfig.create('StringNameSpaceBinding',serverId,[['name','TestName'],
['nameInSpace','TestNameSpace'],['stringToBind',value] ])
```

You can use the following command to print the value:

```
print AdminConfig.showAttribute(nameSpace, 'stringToBind')
```

which results in the following output:

```
{"param1" "param2"}
```

In Version 7.0.0.5 and later, the Jython scripting language recognizes the comma if you precede it with a backslash character (\). For example, in the original example set of Jython commands, change the first line to the following command:

```
value='{"param1"\,"param2"}'
```

When you print the value, the following output returns:

```
{"param1","param2"}
```

types

Use the types command to return a list of the configuration object types that you can manipulate. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of object types, as the following example displays:

```
AdminService
Agent
ApplicationConfig
ApplicationDeployment
ApplicationServer
AuthMechanism
AuthenticationTarget
AuthorizationConfig
AuthorizationProvider
AuthorizationTableImpl
BackupCluster
CMPConnectionFactory
CORBAObjectNameSpaceBinding
Cell
CellManager
Classloader
ClusterMember
ClusteredTarget
CommonSecureInteropComponent
```

Examples

The following examples return each object type in your configuration:

- Using Jacl:

  ```
  $AdminConfig types
  ```

- Using Jython:

  ```
  print AdminConfig.types()
  ```

The following examples return each object type in your configuration that contains the `security` string:

- Using Jacl:

  ```
  $AdminConfig types *security*
  ```

- Using Jython:

```
print AdminConfig.types('*security*')
```

uninstallResourceAdapter

Use the uninstallResourceAdapter command to uninstall a Java 2 Connector (J2C) resource adapter with the given J2C resource adapter configuration ID and an option list. When you remove a J2CResourceAdapter object from the configuration repository, the installed directory will be removed at the time of synchronization. A stop request will be sent to the J2CResourceAdapter MBean that was removed.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the resource adapter to remove.

Optional parameters

options list
Specifies the uninstall options for command. The valid option is `force`. This option forces the uninstallation of the resource adapter without checking whether the resource adapter is being used by an application. The application that is using it will not be uninstalled. If you do not specify the force option and the specified resource adapter is still in use, the resource adapter is not uninstalled.

Sample output

The command returns the configuration ID of the J2C resource adapter that is removed, as the following example displays:

```
WASX7397I: The following J2CResourceAdapter objects are removed:
MyJ2CRA(cells/juniarti/nodes/juniarti|resources.xml#J2CResourceAdapter_1069433028609)
```

Examples

- Using Jacl:

```
set j2cra [$AdminConfig getid /J2CResourceAdapter:MyJ2CRA/]
$AdminConfig uninstallResourceAdapter $j2cra {-force}
$AdminConfig save
```

- Using Jython:

```
j2cra = AdminConfig.getid('/J2CResourceAdapter:MyJ2CRA/')
print AdminConfig.uninstallResourceAdapter(j2cra, '[-force]')
AdminConfig.save()
```

unsetAttributes

Use the unsetAttributes command to reset specific attributes for a configuration object to the default values.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the configuration object of interest.

attributes
Specifies the attributes to reset to the default values.

Optional parameters

None.

Examples

- Using Jacl:

```
set cluster [$AdminConfig getid /ServerCluster:myCluster]
$AdminConfig unsetAttributes $cluster {"enableHA", "preferLocal"}
```

- Using Jython:

```
cluster = AdminConfig.getid("/ServerCluster:myCluster")
AdminConfig.unsetAttributes(cluster, ["enableHA", "preferLocal"])
```

validate

Use the validate command to request the configuration validation results based on the files in your workspace, the value of the cross-document validation enabled flag, and the validation level setting. Optionally, you can specify a configuration ID to set the scope. If you specify a configuration ID, the scope of this request is the object named by the configuration ID parameter.

Target object

None.

Required parameters

None.

Optional parameters

configuration ID
Specifies the configuration ID of the object of interest.

Sample output

The command returns a string that contains the results of the validation, as the following example displays:

```
WASX7193I: Validation results are logged in c:\WebSphere5\AppServer\logs\wsadmin.valout: Total number of messages: 16
WASX7194I: Number of messages of severity 1: 16
```

Examples

- Using Jacl:

```
$AdminConfig validate
```

- Using Jython:

```
print AdminConfig.validate()
```

Commands for the AdminControl object using wsadmin scripting

Use the AdminControl object to invoke operational commands that manage objects for the application server.

Many of the AdminControl commands have multiple signatures so that they can either invoke in a raw mode using parameters that are specified by Java Management Extensions (JMX), or by using strings for parameters. In addition to operational commands, the AdminControl object supports some utility commands for tracing, reconnecting with a server, and converting data types.

The following commands are available for the AdminControl object:

completeObjectName

Use the **completeObjectName** command to create a string representation of a complete ObjectName value that is based on a fragment. This command does not communicate with the server to find a matching ObjectName value. If the system finds several MBeans that match the fragment, the command returns the first one.

Target object

None.

Required parameters

object name
Specifies the name of the object to complete. (ObjectName)

template
Specifies the name of the template to use. For example, the template might be `type=Server,*`. (java.lang.String)

Optional parameters

None.

Sample output

The command does not return output.

Examples

- Using Jacl:

```
set serverON [$AdminControl completeObjectName node=mynode,type=Server,*]
```

- Using Jython:

```
serverON = AdminControl.completeObjectName('node=mynode,type=Server,*')
```

getAttribute

Use the **getAttribute** command to return the value of the attribute for the name that you provide.

If you use the **getAttribute** command to determine the state of an application, one of the following values is returned:

- 0 - which indicates that the application is starting
- 1 - which indicates that the application has started
- 2 - which indicates that the application is stopping
- 3 - which indicates that the application has stopped
- 4 - which indicates that the application failed to start

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the name of the attribute to query. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a string that contains the value of the attribute that you query, as the following example displays:

```
'DeploymentManager'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl getAttribute $objNameString processType
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.getAttribute(objNameString, 'processType')
```

getAttribute_jmx

Use the **getAttribute_jmx** command to return the value of the attribute for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the name of the attribute to query. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a string that contains the value of the attribute that you query, as the following example displays:

```
'DeploymentManager'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
set objName [java::new javax.management.ObjectName $objNameString]
$AdminControl getAttribute_jmx $objName processType
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:=type=Server,*')
import  javax.management  as  mgmt
objName =  mgmt.ObjectName(objNameString)
print AdminControl.getAttribute_jmx(objName, 'processType')
```

getAttributes

Use the **getAttributes** command to return the attribute values for the names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the names of the attributes to query. (java.lang.String[] or java.lang.Object[])

Optional parameters

None.

Sample output

The command returns a string that contains the value of the attribute that you query, as the following example displays:

```
'[ [cellName myCell01] [nodeName myCellManager01] ]'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl getAttributes $objNameString "cellName nodeName"
```

- Using Jython with string attributes:

```
objNameString = AdminControl.completeObjectname('WebSphere:type=Server,*)
print AdminControl.getAttributes(objNameString, '[cellName nodeName]')
```

- Using Jython with object attributes:

```
objNameString = AdminControl.completeObjectname('WebSphere:type=Server,*)
print AdminControl.getAttributes(objNameString, ['cellName', 'nodeName'])
```

getAttributes_jmx

Use the **getAttributes_jmx** command to return the attribute values for the names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the names of the attributes to query. (java.lang.String[] or java.lang.Object[])

Optional parameters

None.

Sample output

The command returns an attribute list.

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attrs [java::new {String[]} 2 {cellName nodeName}]
$AdminControl getAttributes_jmx $objName $attrs
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
attrs = ['cellName', 'nodeName']
print AdminControl.getAttributes_jmx(objName, attrs)
```

getCell

Use the **getCell** command to return the name of the connected cell.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the cell name that you query, as the following example displays:

```
Mycell
```

Examples

- Using Jacl:

  ```
  $AdminControl getCell
  ```

- Using Jython:

  ```
  print AdminControl.getCell()
  ```

getConfigId

Use the **getConfigId** command to create a configuration ID from an ObjectName or an ObjectName fragment. Each MBean does not have corresponding configuration objects. If several MBeans correspond to an ObjectName fragment, a warning is created and a configuration ID builds for the first MBean that the system finds.

Target object

None.

Required parameters

object name
Specifies the name of the object of interest. The object name string can be a wildcard, specified with an asterisk character (*).

Optional parameters

None.

Sample output

The command returns a string that contains the configuration ID of interest.

Examples

- Using Jacl:
- Using Jython:

getDefaultDomain

Use the **getDefaultDomain** command to return the default domain name from the server.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the default domain name of interest, as the following example displays:

```
WebSphere
```

Examples

- Using Jacl:

  ```
  $AdminControl getDefaultDomain
  ```

- Using Jython:

  ```
  print AdminControl.getDefaultDomain()
  ```

getDomainName

Use the **getDomainName** command to return the domain name from the server.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the domain name of interest, as the following example displays:

```
WebSphere
```

Examples

- Using Jacl:

```
$AdminControl getDomainName
```

- Using Jython:

```
print AdminControl.getDomainName()
```

getHost

Use the **getHost** command to return the name of your host.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the name of the host of interest, as the following example displays:

```
myHost
```

Examples

- Using Jacl:

```
$AdminControl getHost
```

- Using Jython:

```
print AdminControl.getHost()
```

getMBeanCount

Use the **getMBeanCount** command to return the number of MBeans that are registered in the server.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns an integer value that contains the number of MBeans that are registered in the server, as the following example displays:

Examples

- Using Jacl:

```
$AdminControl getMBeanCount
```

- Using Jython:

```
print AdminControl.getMBeanCount()
```

getMBeanInfo_jmx

Use the **getMBeanInfo_jmx** command to return the Java Management Extension MBeanInfo structure that corresponds to an ObjectName value. No string signature exists for this command, because the Help object displays most of the information available from the **getMBeanInfo_jmx** command.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

Optional parameters

None.

Sample output

The command returns a javax.management.MBeanInfo object, as the following example displays:

```
javax.management.modelmbean.ModelMBeanInfoSupport@10dd5f35
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]
set objName [$AdminControl makeObjectName $objectNameString]
$AdminControl getMBeanInfo_jmx $objName
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
print AdminControl.getMBeanInfo_jmx(objName)
```

getNode

Use the **getNode** command to return the name of the connected node.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string containing the name of the node, as the following example displays:

```
myNode01
```

Examples

- Using Jacl:

  ```
  $AdminControl getNode
  ```

- Using Jython:

  ```
  print AdminControl.getNode()
  ```

getObjectInstance

Use the **getObjectInstance** command to return the object instance that matches the input object name.

Target object

None.

Required parameters

object name
Specifies the name of the object of interest. (ObjectName)

Optional parameters

None.

Sample output

The command returns the object instance that matches the input object name, as the following example displays:

```
javax.management.modelmbean.RequiredModelMBean
```

Examples

- Using Jacl:

  ```
  set server [$AdminControl completeObjectName type=Server,*]
  set serverOI [$AdminControl getObjectInstance $server]
  ```

  Use the following example to manipulate the return value of the getObjectInstance command:

  ```
  puts [$serverOI getClassName]
  ```

- Using Jython:

  ```
  server = AdminControl.completeObjectName('type=Server,*')
  serverOI = AdminControl.getObjectInstance(server)
  ```

  Use the following example to manipulate the return value of the getObjectInstance command:

  ```
  print serverOI.getClassName()
  ```

getPort

Use the **getPort** command to return the name of the port used for the scripting connection.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the port number of the port that the system uses to establish the scripting connection, as the following example displays:

```
8877
```

Examples

- Using Jacl:

```
$AdminControl getPort
```

- Using Jython:

```
print AdminControl.getPort()
```

getPropertiesForDataSource (Deprecated)

The **getPropertiesForDataSource** command is deprecated, and no replacement exists. This command incorrectly assumes the availability of a configuration service when running in connected mode.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the following message:

```
WASX7389E: Operation not supported - getPropertiesForDataSource command is not supported.
```

Examples

- Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]
$AdminControl getPropertiesForDataSource $ds
```

- Using Jython:

```
ds = AdminConfig.list('DataSource')

# get line separator
import  java.lang.System  as  sys
lineSeparator = sys.getProperty('line.separator')
```

```
dsArray = ds.split(lineSeparator)
print AdminControl.getPropertiesForDataSource(dsArray[0])
```

getType

Use the **getType** command to return the connection type used for the scripting connection.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a string that contains the connection type for the scripting connection, as the following example displays:

```
SOAP
```

Examples

- Using Jacl:

  ```
  $AdminControl getType
  ```

- Using Jython:

  ```
  print AdminControl.getType()
  ```

help

Use the **help** command to return general help text for the AdminControl object.

Target object

None.

Required parameters

None.

Optional parameters

command
Specifies the command for which to return help information. The command name is not case-sensitive.

Sample output

The command returns a string that details specific options for the **help** command, as the following example displays:

```
WASX7027I: The AdminControl object enables the manipulation of MBeans that run in a
WebSphere Application Server process. The number and type of MBeans
that are available to the scripting client depend on the server to which the client is connected.
If the client is connected to a deployment manager, then all the MBeans running in the Deployment Manager
```

```
are visible, as are all the MBeans running in the node agentsthat are connected to this deployment manager,
 and all the MBeans that run in the application servers on those nodes.

The following commands are supported by the AdminControl object; more detailed information about each of these
commands is available by using the "help" command of the AdminControl object and supplying the name
of the command as an argument.

Many of these commands support two different sets of signatures: one that accepts and returns strings, and one
low-level set that works with JMX objects like ObjectName and AttributeList. In most situations, the string
signatures are likely to be more useful, but JMX-object signature versions are supplied as well.  Each of these
JMX-object signature commands has "_jmx" appended to the command name, so an "invoke" command, as well as a
"invoke_jmx" command are supported.

completeObjectName  Return a String version of an object name given atemplate name
getAttribute_jmx   Given ObjectName and name of attribute, returns value ofattribute
getAttribute       Given String version of ObjectName and name of attribute, returns value of attribute
getAttributes_jmx   Given ObjectName and array of attribute names, returns AttributeList
getAttributes      Given String version of ObjectName and attribute names, returns String of name value pairs
getCell            returns the cell name of the connected server
getConfigId        Given String version of ObjectName, return a config id for the corresponding configuration
          object, if any.
getDefaultDomain   returns "WebSphere"
getDomainName      returns "WebSphere"
getHost            returns String representation of connected host
getMBeanCount      returns number of registered beans
getMBeanInfo_jmx   Given ObjectName, returns MBeanInfo structure for MBean
getNode            returns the node name of the connected server
getPort            returns String representation of port in use
getType            returns String representation of connection type in use help  help     Show help information
invoke_jmx         Given ObjectName, name of command, array of parameters andsignature, invoke command on
          MBean specified
invoke             Invoke a command on the specified MBean
isRegistered_jmx   true if supplied ObjectName is registered
isRegistered       true if supplied String version of ObjectName is registered
makeObjectName     Return an ObjectName built with the given string
queryNames_jmx     Given ObjectName and QueryExp, retrieves set of ObjectNames that match.
queryNames         Given String version of ObjectName, retrieves String of ObjectNames that match.
reconnect          reconnects with serversetAttribute_jmx Given ObjectName and Attribute object,
          set attribute for MBean specified
setAttribute       Given String version of ObjectName, attribute name andattribute value, set attribute for
          MBean specified
setAttributes_jmx   Given ObjectName and AttributeList object, set attributes for the MBean specified
startServer        Given the name of a server, start that server.
stopServer         Given the name of a server, stop that server.
testConnection     Test the connection to a DataSource object
trace              Set the wsadmin trace specification
```

If you specify a specific command with the help command, the wsadmin tool returns detailed help about the command, as the following example displays:

```
WASX7043I: command: getAttribute
Arguments: object name, attribute
Description: Returns value of "attribute" for the MBean described by "object name."
```

Examples

- Using Jacl:

  ```
  $AdminControl help

  $AdminControl help getAttribute
  ```

- Using Jython:

  ```
  print AdminControl.help()

  print AdminControl.help('getAttribute')
  ```

invoke

Use the **invoke** command to invoke a specific MBean operation based on the number of parameters that operation requires. If this constraint is not sufficient to select a unique operation, use invoke_jmx. The

supplied parameters are converted to the object types required by the selected operation's signature before the operation is invoked. Any returned value is converted to its string representation.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest.

operation
Specifies the operation to invoke.

Optional parameters

arguments
Specifies the arguments required for the operation. If no arguments are required for the operation of interest, you can omit the arguments parameter.

The arguments parameter is a single string. Each individual argument in the string can contain spaces.

Sample output

The command returns a string that shows the result of the invocation.

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl invoke $objNameString stop
```

```
set objNameString [$AdminControl completeObjectName WebSphere:type=DynaCache,*]
$AdminControl invoke $mbean getCacheStatistics {"DiskCacheSizeInMB ObjectsReadFromDisk4000K
 RemoteObjectMisses"}
```

gotcha: Make sure the mbean variable, $mbean, is defined before issuing the preceding command

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.invoke(objNameString, 'stop')
```

```
objNameString = AdminControl.completeObjectName("WebSpheretype=DynaCache,*")
 AdminControl.invoke(dc, "getCacheStatistics", '["DiskCacheSizeInMB ObjectReadFromDisk4000K
 RemoteObjectMisses"]')
```

- Using Jython list:

```
objNameString = AdminControl.completeObjectName("WebSphere:type=DynaCache,*")
AdminControl.invoke(dc, "getCacheStatistics", [["DiskCacheSizeInMB", "ObjectReadFromDisk4000K",
"RemoteObjectMisses"]])
```

invoke_jmx

Use the **invoke_jmx** command to invoke the object operation by conforming the parameter list to the signature. The command returns the result of the invocation.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

operation
Specifies the operation to invoke. (java.lang.String)

Optional parameters

arguments
Specifies the arguments required for the operation. If no arguments are required for the operation of interest, you can omit the arguments parameter. (java.lang.String[] or java.lang.Object[])

Sample output

The command returns a string that shows the result of the invocation.

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
set objName [java::new javax.management.ObjectName $objNameString]
set parms [java::new {java.lang.Object[]} 1 com.ibm.ejs.sm.*=all=disabled]
set signature [java::new {java.lang.String[]} 1 java.lang.String]
$AdminControl invoke_jmx $objName $parms $signature
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
import  javax.management  as  mgmt
objName =  mgmt.ObjectName(objNameString)
parms = ['com.ibm.ejs.sm.*=all=disabled']
signature = ['java.lang.String']
print AdminControl.invoke_jmx(objName, parms, signature)
```

isRegistered

Use the **isRegistered** command to determine if a specific object name is registered.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a boolean value for the object of interest. If the ObjectName value is registered in the server, then the value is `1`, as the following example displays:

```
wsadmin>s = AdminControl.queryNames( 'type=Server,*' ).splitlines()[ 0 ]
wsadmin>AdminControl.isRegistered( s )
1
wsadmin>
```

If the ObjectName value is not registered in the server, then the value is `0`.

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]
$AdminControl isRegistered $objNameString
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')
print AdminControl.isRegistered(objNameString)
```

isRegistered_jmx

Use the **isRegistered_jmx** command to determine if a specific object name is registered.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a boolean value for the object of interest. If the ObjectName value is registered in the server, then the value is `true`, as the following example displays:

```
true
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]
set objName [$AdminControl makeObjectName $objNameString]
$AdminControl isRegistered_jmx $objName
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
print AdminControl.isRegistered_jmx(objName)
```

makeObjectName

Use the **makeObjectName** command to create an ObjectName value that is based on the strings input. This command does not communicate with the server, so the ObjectName value that results might not exist. If the string you supply contains an extra set of double quotes, they are removed. If the string does not begin with a Java Management Extensions (JMX) domain, or a string followed by a colon, then the WebSphere Application Server string appends to the name.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns an Objectname object constructed from the object name string.

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,node=mynode,*]
set objName [$AdminControl makeObjectName $objNameString]
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,node=mynode,*')
objName = AdminControl.makeObjectName(objectNameString)
```

queryMBeans

Use the **queryMBeans** command to query for a list of object instances that match the object name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of interest. (ObjectName)

Optional parameters

query
Specifies the query expression. (QueryExp)

Sample output

The command returns a list of object instances for the object name specified, as the following example displays:

```
WebSphere:name=PlantsByWebSphere,process=server1,platform=dynamicproxy,node=Goodog,
J2EEName=PlantsByWebSphere,Server=server1,version=6.1.0.0,type=Application,
mbeanIdentifier=cells/GooddogNode02Cell/applications/PlantsByWebSphere.ear/
deployments/PlantsByWebSphere/deployment.xml#ApplicationDeployment_1126623343902,
cell=GooddogNode02Cell
```

Examples

- Using Jacl:

```
set apps [$AdminControl queryMBeans type=Application,*]
```

Use the following example to manipulate the return value of the queryMBeans command:

```
set appArray [$apps toArray]
set app1 [java::cast javax.management.ObjectInstance [$appArray get 0]]
puts [[$app1 getObjectName] toString]
```

The following example specifies the object name and the query expression:

```
set apps [$AdminControl queryMBeans type=Application,* [java::null]]
```

Use the following example to manipulate the return value of the queryMBeans command:

```
set appArray [$apps toArray]
set app1 [java::cast javax.management.ObjectInstance [$appArray get 0]]
puts [[$app1 getObjectName] toString]
```

- Using Jython:

```
apps = AdminControl.queryMBeans('type=Application,*')
```

Use the following example to manipulate the return value of the queryMBeans command:

```
appArray = apps.toArray()
app1 = appArray[0]
print app1.getObjectName().toString()
```

The following example specifies the object name and the query expression:

```
apps = AdminControl.queryMBeans('type=Application,*',None)
```

Use the following example to manipulate the return value of the queryMBeans command:

```
appArray = apps.toArray()
app1 = appArray[0]
print app1.getObjectName().toString()
```

queryNames

Use the **queryNames** command to query for a list of each of the ObjectName objects based on the name template.

Target object

None.

Required parameters

object name
Specifies the object name of interest. You can specify a wildcard for the object name parameter with the asterisk character (*). (java.lang.String)

Optional parameters

None.

Sample output

The command returns a string that contains the ObjectNames that match the input object name, as the following example displays:

```
WebSphere:cell=BaseApplicationServerCell,
name=server1,mbeanIdentifier=server1,
type=Server,node=mynode,process=server1
```

Examples

- Using Jacl:

```
$AdminControl queryNames WebSphere:type=Server,*
```

- Using Jython:

```
print AdminControl.queryNames('WebSphere:type=Server,*')
```

queryNames_jmx

Use the **queryNames_jmx** command to query for a list of each of the ObjectName objects based on the name template and the query conditions that you specify.

Target object

None.

Required parameters

object name
Specifies the object name of interest. You can specify a wildcard for the object name parameter with the asterisk character (*). (ObjectName)

query
Specifies the query expression to use. (javax.management.QueryExp)

Optional parameters

None.

Sample output

The command returns a string that contains the ObjectNames that match the input object name, as the following example displays:

```
[WebSphere:cell=BaseApplicationServerCell,name=server1,mbeanIdentifier=
server1,type=Server,node=mynode,process=server1]
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName type=Server,*]
set objName [$AdminControl makeObjectName $objNameString]
set null [java::null]
$AdminControl queryNames_jmx $objName $null
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('type=Server,*')
objName = AdminControl.makeObjectName(objectNameString)
print AdminControl.queryNames_jmx(objName, None)
```

reconnect

Use the **reconnect** command to reconnect to the server, and to clear information out of the local cache.

Target object

None.

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a message that displays the status of the operation, as the following example displays:

```
WASX7074I: Reconnect of SOAP connector to host myhost completed.
```

Examples

- Using Jacl:

```
$AdminControl reconnect
```

- Using Jython:

```
print AdminControl.reconnect()
```

setAttribute

Use the **setAttribute** command to set the attribute value for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (java.lang.String)

attribute name
Specifies the name of the attribute to set. (java.lang.String)

attribute value
Specifies the value of the attribute of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns does not return output.

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
$AdminControl setAttribute $objNameString traceSpecification com.ibm.*=all=disabled
```

- Using Jython:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
print AdminControl.setAttribute(objNameString, 'traceSpecification',  'com.ibm.*=all=disabled')
```

setAttribute_jmx

Use the **setAttribute_jmx** command to set the attribute value for the name that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (ObjectName)

attribute
Specifies the name of the attribute to set. (Attribute)

Optional parameters

None.

Sample output

The command returns does not return output.

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attr [java::new javax.management.Attribute traceSpecification com.ibm.*=all=disabled]
$AdminControl setAttribute_jmx $objName $attr
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
import  javax.management  as  mgmt
objName = AdminControl.makeObjectName(objectNameString)
attr = mgmt.Attribute('traceSpecification', 'com.ibm.*=all=disabled')
print AdminControl.setAttribute_jmx(objName, attr)
```

setAttributes

Use the **setAttributes** command to set the attribute values for the object names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (String)

attributes
Specifies the names of the attributes to set. (java.lang.String[] or java.lang.Object[])

Optional parameters

None.

Sample output

The command returns a list of object names that are successfully set by the command invocation, as the following example displays:

```
'[traceSpecification com.ibm.ws.*=all=enabled]'
```

Examples

- Using Jacl:

```
set objNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
$AdminControl setAttributes $objNameString {{traceSpecification com.ibm.ws.*=all=enabled}}
```

- Using Jython with string attributes:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
AdminControl.setAttributes(objNameString, '[[traceSpecification "com.ibm.ws.*=all=enabled"]]')
```

- Using Jython with object attributes:

```
objNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
print AdminControl.setAttributes(objNameString, [['traceSpecification', 'com.ibm.ws.*=all=enabled']])
```

setAttributes_jmx

Use the **setAttributes_jmx** command to set the attribute values for the object names that you provide.

Target object

None.

Required parameters

object name
Specifies the object name of the MBean of interest. (String)

attributes
Specifies the names of the attributes to set. (javax.management.AttributeList)

Optional parameters

None.

Sample output

The command returns an attribute list of object names that are successfully set by the command invocation, as the following example displays:

```
'[traceSpecification com.ibm.ws.*=all=enabled]'
```

Examples

- Using Jacl:

```
set objectNameString [$AdminControl completeObjectName WebSphere:type=TraceService,*]
set objName [$AdminControl makeObjectName $objectNameString]
set attr [java::new javax.management.Attribute traceSpecification com.ibm.ws.*=all=enabled]
set alist [java::new javax.management.AttributeList]
$alist add $attr
$AdminControl setAttributes_jmx $objName $alist
```

- Using Jython:

```
objectNameString = AdminControl.completeObjectName('WebSphere:type=TraceService,*')
import  javax.management  as  mgmt
objName = AdminControl.makeObjectName(objectNameString)
attr = mgmt.Attribute('traceSpecification', 'com.ibm.ws.*=all=enabled')
alist = mgmt.AttributeList()
alist.add(attr)
print AdminControl.setAttributes_jmx(objName, alist)
```

startServer

Use the **startServer** command to start the specified application server by locating it in the configuration. This command uses the default wait time. Use the following guidelines to determine which parameters to use:

- If the scripting process is attached to a node agent server, you must specify the server name. You can also specify the optional wait time and node name parameters.
- If the scripting process is attached to a deployment manager process, you must specify the server name and node name. You can also specify the optional wait time parameter.

Target object

None.

Required parameters

server name
Specifies the name of the server to start. (java.lang.String)

Optional parameters

node name
Specifies the name of the node of interest. (java.lang.String)

wait time
Specifies the number of seconds that the start process waits for the server to start. The default wait time is 1200 seconds. (java.lang.String)

Sample output

The command returns a message to indicate if the server starts successfully, as the following example displays:

```
'[traceSpecification com.ibm.ws.*=all=enabled]'
```

Examples

Using Jacl:

- The following example specifies only the name of the server to start:
  ```
  $AdminControl startServer server1
  ```
- The following example specifies the name of the server to start and the wait time:
  ```
  $AdminControl startServer server1 100
  ```
- The following example specifies the name of the server to start and the name of the node:
  ```
  $AdminControl startServer server1 myNode
  ```
- The following example specifies the name of the server, the name of the node, and the wait time:
  ```
  $AdminControl startServer server1 myNode 100
  ```

Using Jython:

- The following example specifies only the name of the server to start:
  ```
  AdminControl.startServer('server1')
  ```
- The following example specifies the name of the server to start and the wait time:
  ```
  AdminControl.startServer('server1', 100)
  ```
- The following example specifies the name of the server to start and the name of the node:
  ```
  AdminControl.startServer('server1', 'myNode')
  ```
- The following example specifies the name of the server, the name of the node, and the wait time:
  ```
  AdminControl.startServer('server1', 'myNode', 100)
  ```

stopServer

Use the **stopServer** command to stop the specified application server. When the **stopServer** command runs without the immediate or terminate flags, the server finishes any work in progress, but does not accept any new work once it begins the stop process. Use the following options to determine which parameters to use:

- Use the server name and the node name parameters to stop a server in a specific node.
- Use the server name and immediate flag parameters to stop the server immediately. If this parameter is not specified, the system stops the server normally.
- Use the server name, node name, and immediate flag parameters to immediately stop a server for a specific node.

gotcha: An error message is issued if only specify the server name when you attempt to stop that server.

Target object

None.

Required parameters

server name
Specifies the name of the server to start. (java.lang.String)

Optional parameters

node name
Specifies the name of the node of interest. (java.lang.String)

immediate flag
Specifies to stop the server immediately if the value is set to `immediate`. If you specify the immediate flag, the server does not finish processing any work in progress, does not accept any new work, and ends the server process. (java.lang.String)

terminate flag
Specifies that the server process should be terminated by the operating system. (String)

Sample output

The command returns a message to indicate if the server stops successfully, as the following example displays:

```
WASX7337I: Invoked stop for server "server1" Waiting for stop completion.
'WASX7264I: Stop completed for server "server1" on node "myNode"'
```

Examples

Using Jacl:

- The following example specifies only the name of the server to stop:
  ```
  $AdminControl stopServer server1
  ```
- The following example specifies the name of the server to stop and indicates that the server should stop immediately:
  ```
  $AdminControl stopServer server1 immediate
  ```
- The following example specifies the name of the server to stop and the name of the node:
  ```
  $AdminControl stopServer server1 myNode
  ```
- The following example specifies the name of the server, the name of the node, and indicates that the server should stop immediately:
  ```
  $AdminControl stopServer server1 myNode immediate
  ```

Using Jython:

- The following example specifies the name of the server to stop and indicates that the server should stop immediately:
  ```
  AdminControl.stopServer('server1','immediate')
  ```
- The following example specifies the name of the server to stop and the name of the node:
  ```
  AdminControl.stopServer('server1','myNode')
  ```
- The following example specifies the name of the server, the name of the node, and indicates that the server should stop immediately:
  ```
  AdminControl.stopServer('server1','myNode','immediate')
  ```

testConnection

Use the **testConnection** command to test a data source connection. This command works with the data source that resides in the configuration repository. If the data source to be tested is in the temporary workspace that holds the update to the repository, you must save the update to the configuration repository before running this command. Use this command with the configuration ID that corresponds to the data source and the WAS40DataSource object types.

Target object

None.

Required parameters

configuration ID
Specifies the configuration ID of the data source object of interest. (java.lang.String)

Optional parameters

None.

Sample output

The command returns a message that indicates a successful connection or a connection with a warning. If the connection fails, an exception is created from the server indicating the error. For example:

```
WASX7217I: Connection to provided datasource was successful.
```

Examples

- Using Jacl:

```
set ds [lindex [$AdminConfig list DataSource] 0]
$AdminControl testConnection $ds
```

- Using Jython:

```
# get line separator
import java.lang.System as sys
lineSeparator = sys.getProperty('line.separator')
ds = AdminConfig.list('DataSource').split(lineSeparator)[0]
print AdminControl.testConnection(ds)
```

trace

Use the **trace** command to set the trace specification for the scripting process to the value that you specify.

Target object

None.

Required parameters

trace specification
Specifies the trace to enable for the scripting process. (java.lang.String)

Optional parameters

None.

Sample output

The command does not return output.

Examples

- Using Jacl:

```
$AdminControl trace com.ibm.ws.scripting.*=all=enabled
```

- Using Jython:

```
print AdminControl.trace('com.ibm.ws.scripting.*=all=enabled')
```

Commands for the AdminApp object using wsadmin scripting

Use the AdminApp object to install, modify, and administer applications.

The AdminApp object interacts with the WebSphere Application Server management and configuration services to make application inquiries and changes. This interaction includes installing and uninstalling applications, listing modules, exporting, and so on.

You can start the scripting client when no server is running, if you want to use only local operations. To run in local mode, use the `-conntype NONE` option to start the scripting client. You receive a message that you are running in the local mode. Running the AdminApp object in local mode when a server is currently running is not recommended. This is because any configuration changes made in local mode will not be reflected in the running server configuration and vice versa. If you save a conflicting configuration, you could corrupt the configuration.

The following commands are available for the AdminApp object:

The following note applies to the xmi file references in this topic:

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

deleteUserAndGroupEntries

Use the deleteUserAndGroupEntries command to delete users or groups for all roles, and to delete user IDs and passwords for all of the RunAs roles that are defined in the application.

Target object

None.

Required parameters

application name
Specifies the application of interest.

Optional parameters

None.

Examples

- Using Jacl:
  ```
  $AdminApp deleteUserAndGroupEntries myapp
  ```
- Using Jython string:
  ```
  print AdminApp.deleteUserAndGroupEntries('myapp')
  ```
- Using Jython list:
  ```
  print AdminApp.deleteUserAndGroupEntries(['myapp'])
  ```

edit

Use the edit command to edit an application or module in batch mode. The edit command changes the application specified by the application name argument using the options specified by the options argument. No options are required for the edit command.

Target object

None.

Required parameters

application name
Specifies the application of interest.

options
Specifies the options to apply to the application or module configuration.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp edit "JavaMail Sample" {-MapWebModToVH {{"JavaMail Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH}}}
```

- Using Jython string:

```
print AdminApp.edit("JavaMail Sample", '[-MapWebModToVH [["JavaMail 32 Sample WebApp"
mtcomps.war,WEB-INF/web.xml newVH]]]')
```

- Using Jython list:

```
option = [["JavaMail 32 Sample WebApp", "mtcomps.war,WEB-INF/web.xml", "newVH"]]
mapVHOption = ["-MapWebModToVH", option]
print AdminApp.edit("JavaMail Sample", mapVHOption)
```

editInteractive

Use the editInteractive command to edit an application or module in interactive mode. The editInteractive command changes the application deployment. Specify these changes in the options parameter. No options are required for the editInteractive command.

Target object

None.

Required parameters

application name
Specifies the application of interest.

options
Specifies the options to apply to the application or module configuration.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp editInteractive ivtApp
```

- Using Jython string:

```
AdminApp.editInteractive('ivtApp')
```

export

Use the export command to export the application name parameter to a file that you specify by the file name.

Target object

None.

Required parameters

application name
Specifies the application of interest.

file name
Specifies the file name to export the application name to.

Optional parameters

exportToLocal
Specifies that the system should export the application of interest to the file name specified on the local client machine.

Examples

- Using Jacl:

```
$AdminApp export DefaultApplication c:/temp/export.ear {-exportToLocal}
```

- Using Jython:

```
AdminApp.export('DefaultApplication', 'c:/temp/export.ear', '[-exportToLocal]')
```

exportDDL

Use the exportDDL command to extract the data definition language (DDL) from the application name parameter to the directory name parameter that a directory specifies. The options parameter is optional.

Target object

None.

Required parameters

application name
Specifies the application of interest.

directory name
Specifies the name of the directory to export the application name to.

Optional parameters

options
Specifies the options to pass to the application name specified.

Examples

- Using Jacl:

```
$AdminApp exportDDL "My App" /usr/me/DDL {-ddlprefix myApp}
```

- Using Jython string:

```
print AdminApp.exportDDL("My App", '/usr/me/DDL', '[-ddlprefix myApp]')
```

exportFile

Use the exportFile command to export the contents of a single file specified by the uniform resource identifier (URI) from the application of interest.

Target object

None.

Required parameters

application name
Specifies the application of interest.

URI
Specifies the single file to export. Specify the URI within the context of an application, as the following example displays: `META-INF/application.xml`. To specify files within a module, the URI begins with a module URI, as the following example displays: `foo.war/WEB-INF/web.xml`.

filename
Specifies the fully qualified path and file name of the file to export to.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp exportFile "My App" myapp/components.jar/META-INF/ibm-ejb-jar-bnd.xml
 META-INF/ibm-ejb-jar-bnd.xml
```

- Using Jython string:

```
AdminApp.exportFile('My App', 'myapp/components.jar/META-INF/ibm-ejb-jar-bnd.xml',
'META-INF/ibm-ejb-jar-bnd.xml')
```

getDeployStatus

Use the getDeployStatus command to display the deployment status of the application. After installing or updating a large application, use this command to display detailed status information for application binary file expansion. You cannot start the application until the system extracts the application binaries.

Target object

None.

Required parameters

application name
Specifies the name of the application of interest.

Optional parameters

None.

Examples

- Using Jacl:

```
$AdminApp getDeployStatus myApplication
```

- Using Jython:

```
print AdminApp.getDeployStatus('myApplication')
```

Running the getDeployStatus command where *myApplication* is `DefaultApplication` results in status information about DefaultApplication resembling the following:

```
ADMA5071I: Distribution status check started for application DefaultApplication.
WebSphere:cell=myCell01,node=myNode01,distribution=unknown,expansion=unknown
ADMA5011I: The cleanup of the temp directory for application DefaultApplication is complete.
ADMA5072I: Distribution status check completed for application DefaultApplication.
WebSphere:cell=myCell01,node=myNode01,distribution=unknown,expansion=unknown
```

help

Use the help command to display general help information about the AdminApp object.

Target object

None.

Required parameters

None.

Optional parameters

operation name
Specify this option to display help for an AdminApp command or installation option.

Sample output

The following output is returned if you do not specify an argument:

```
WASX7095I: The AdminApp object allows application objects to be manipulated including installing, uninstalling,
editing, and listing.  Most of the commands supported by AdminApp operate in two modes: the default mode is one
in which AdminApp communicates with the WebSphere Application Server to accomplish its tasks.  A local mode is also
possible, in which no server communication takes place.  The local mode of operation is invoked by including the
"-conntype NONE" flag in the option string supplied to the command.

The following commands are supported by AdminApp; more detailed information about each of these commands
is available by using the "help" command of AdminApp and supplying the name of the command as an argument.

edit               Edit the properties of an application
editInteractive    Edit the properties of an application interactively
export             Export application to a file
exportDDL          Extract DDL from application to a directory
help                Show help information
install            Installs an application, given a file name and an option string.
installInteractive  Installs an application in interactive mode, given a file name and an option string.
list               List all installed applications
listModules        List the modules in a specified
application options  Shows the options available, either for a given file, or in general.
taskInfo           Shows detailed information pertaining to a given installation task for a given file
uninstall          Uninstalls an application, given an application name and an option string
```

The following output is returned if you specify `uninstall` as the operation name argument:

```
WASX7102I: Method: uninstall
Arguments: application name, options
Description: Uninstalls application named by "application name" using the options supplied by String 2.
Method: uninstall
Arguments: application name
Description: Uninstalls the application specified by "application name" using default options.
```

Examples

Using Jacl:

- The following example does not specify any arguments:
  ```
  $AdminApp help
  ```
- The following example specifies the operation name argument:
  ```
  $AdminApp help uninstall
  ```

Using Jython:

- The following example does not specify any arguments:
  ```
  print AdminApp.help()
  ```
- The following example specifies the operation name argument:
  ```
  print AdminApp.help('uninstall')
  ```

install

Use the install command to install an application in non-interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.

Target object

None.

Required parameters

ear file
Specify the path of the .ear file to install.

Optional parameters

options
Specify the installation options for the command.

Examples

- Using Jacl:

```
$AdminApp install c:/apps/myapp.ear
```

- Using Jython:

```
print AdminApp.install('c:/apps/myapp.ear')
```

Many options are available for this command. You can obtain a list of valid options for an Enterprise Archive (EAR) file with the following command:

Using Jacl:

```
$AdminApp options myApp.ear
```

Using Jython:

```
print AdminApp.options('myApp.ear')
```

You can also obtain help for each object with the following command:

Using Jacl:

```
$AdminApp help MapModulesToServers
```

Using Jython:

```
print AdminApp.help('MapModulesToServers')
```

installInteractive

Use the installInteractive command to install an application in interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.

Target object

None.

Required parameters

ear file
Specify the path of the .ear file to install.

Optional parameters

options
Specify the installation options for the command.

Examples

- Using Jacl:

```
$AdminApp installInteractive c:/websphere/appserver/installableApps/jmsample.ear
```

- Using Jython:

```
print AdminApp.installInteractive('c:/websphere/appserver/installableApps/jmsample.ear')
```

isAppReady

Use the isAppReady command to determine if the specified application has been distributed and is ready to be run. Returns a value of `true` if the application is ready, or a value of `false` if the application is not ready. This command is not supported when the wsadmin tool is not connected to a server.

Target object

None.

Required parameters

application name
Specify the name of the application of interest.

Optional parameters

ignoreUnknownState
Tests to see if the specified application has been distributed and is ready to be run. Valid values for the ignoreUnknownState parameter include `true` and `false`. If you specify a value of `true`, nodes and servers with an unknown state will not be included in the final ready return. The command returns a value of `true` if the application is ready or a value of `false` if the application is not ready. This command is not supported when the wsadmin tool is not connected to a server.

Sample output

The following sample output is returned if you specify the application name parameter:

```
ADMA5071I: Distribution status check started for application DefaultApplication.WebSphere:cell=Node03Cell,
node=myNode,distribution=true
ADMA5011I: The cleanup of the temp directory for application DefaultApplication is complete.
ADMA5072I: Distribution status check completed for application DefaultApplication.true
```

The following sample output is returned if you specify the application name and ignoreUnknownState parameters:

```
ADMA5071I: Distribution status check started for application TEST.WebSphere:cell=myCell,node=myNode,
     distribution=unknown
ADMA5011I: The cleanup of the temp directory for application TEST is complete.
ADMA5072I: Distribution status check completed for application TEST.false
```

Examples

The following examples only specify the application name parameter:

- Using Jacl:

```
$AdminApp isAppReady DefaultApplication
```

- Using Jython:

```
print AdminApp.isAppReady('DefaultApplication')
```

The following examples specify the application name and ignoreUnknownState parameters:

- Using Jacl:

```
$AdminApp isAppReady TEST true
```

- Using Jython:

```
print AdminApp.isAppReady('TEST', 'true')
```

list

Use the list command to list the applications that are installed in the configuration.

Target object

None.

Required parameters

None.

Optional parameters

target
Lists the applications that are installed on a given target scope in the configuration.

Sample output

```
adminconsole
DefaultApplication
ivtApp
```

Examples

- Using Jacl:

```
$AdminApp list
```

- Using Jython:

```
print AdminApp.list()
```

The following examples specify a value for the target parameter:

- Using Jacl:

```
$AdminApp list WebSphere:cell=myCell,node=myNode,server=myServer
```

- Using Jython:

```
print AdminApp.list("WebSphere:cell=myCell,node=myNode,server=myServer")
```

listModules

Use the listModules command to list the modules in an application.

Target object

None.

Required parameters

application name
Specifies the application of interest.

Optional parameters

options
Specifies the list of application servers on which the modules are installed. The options parameter is optional. The valid option is -server.

Sample output

The following example is the concatenation of appname, #, module URI, +, and DD URI. You can pass this string to the edit and editInteractive AdminApp commands.

```
ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml
ivtApp#ivt_app.war+WEB-INF/web.xml
```

Examples

- Using Jacl:

  ```
  $AdminApp listModules ivtApp
  ```

- Using Jython:

  ```
  print AdminApp.listModules('ivtApp')
  ```

options

Use the options command to display a list of options for installing an Enterprise Archive (EAR) file.

Target object

None.

Required parameters

None.

Optional parameters

EAR file
Specifies the EAR file of interest.

application name
Specifies the application for which to display a list of options for editing an existing application.

application module name
Specifies the module name for which to display a list of options for editing a module in an existing application. This parameter requires the same module name format as the output that is returned by the listModules command.

file, operations
Displays a list of options for installing or updating an application or application module file. Specify one of the following valid values:

- installapp - Use this option to install the file that is specified.
- updateapp - Use this option to update an existing application with the file that is specified.
- addmodule - Use this option to add the module file that is specified to an existing application.
- updatemodule - Use this option to update an existing module in an application with the module file that is specified.

Sample output

```
WASX7112I: The following options are valid for "ivtApp"
MapRolesToUsers
BindJndiForEJBNonMessageBinding
MapEJBRefToEJB
MapWebModToVH
```

```
MapModulesToServers
distributeApp
nodistributeApp
useMetaDataFromBinary
nouseMetaDataFromBinary
createMBeansForResources
nocreateMBeansForResources
reloadEnabled
noreloadEnabled
verbose
installed.ear.destination
reloadInterval
```

Examples

The following example options command returns the valid options for an EAR file:

- Using Jacl:

  ```
  $AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear
  ```

- Using Jython:

  ```
  print AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear')
  ```

The following example options command returns the valid options for an application:

- Using Jacl:

  ```
  $AdminApp options ivtApp
  ```

- Using Jython:

  ```
  print AdminApp.options('ivtApp')
  ```

The following example options command returns the valid options for an application module:

- Using Jacl:

  ```
  $AdminApp options ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml
  ```

- Using Jython:

  ```
  print AdminApp.options('ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml')
  ```

The following example options command returns the valid options for the operation that is requested with the input file:

- Using Jacl:

  ```
  $AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear updateapp
  ```

- Using Jython:

  ```
  print AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear', 'updateapp')
  ```

publishWSDL

Use the publishWSDL command to publish Web Services Description Language (WSDL) files for the application that is specified in the application name parameter to the file that is specified in the file name parameter.

Target object

None.

Required parameters

file name
Specifies the file of interest.

application name
Specifies the application of interest

Optional parameters

SOAP address prefixes
Specifies the SOAP address prefixes to use.

Sample output

The publishWSDL command does not return output.

Examples

The following example publishWSDL command specifies the application name and the file name:

- Using Jacl:

```
$AdminApp publishWSDL JAXRPCHandlerServer c:/temp/a.zip
```

- Using Jython:

```
print AdminApp.publishWSDL('JAXRPCHandlerServer', 'c:/temp/a.zip')
```

The following example publishWSDL command specifies the application name, file name, and SOAP address prefixes parameter values:

- Using Jacl:

```
$AdminApp publishWSDL JAXRPCHandlersServer c:/temp/a.zip {{JAXRPCHandlersServerApp.war
{{http http://localhost:9080}}}}
```

- Using Jython:

```
print AdminApp.publishWSDL('JAXRPCHandlersServer', 'c:/temp/a.zip', '[[JAXRPCHandlersServerApp.war
 [[http http://localhost:9080]]]]')
```

searchJNDIReferences

Use the searchJNDIReferences command to list applications that refer to the Java Naming and Directory Interface (JNDI) name on a specific node.

Target object

None.

Required parameters

node configuration ID
Specifies the configuration ID for the node of interest.

Optional parameters

options
Specifies the options to use.

Sample output

```
WASX7410W: This operation may take a while depending on the number of applications installed in your system.
MyApp
MapResRefToEJB :ejb-jar-ic.jar : [eis/J2CCF1]
```

Examples

The following example assumes that an installed application named MyApp has a JNDI name of eis/J2CCF1:

- Using Jacl:

```
$AdminApp searchJNDIReferences $node {-JNDIName eis/J2CCF1 -verbose}
```

- Using Jython:

```
print AdminApp.searchJNDIReferences(node, '[-JNDIName eis/J2CCF1 -verbose]')
```

taskInfo

Use the taskInfo command to provide information about a particular task option for an application file. Many task names have changed between V5.x and V6.x for a similar or the exact same operation. You might need to update existing scripts if you are migrating from V5.x to V6.x.

Target object

None.

Required parameters

EAR file
Specifies the EAR file of interest.

task name
Specifies the task for which to request the information.

Optional parameters

None.

Sample output

```
MapWebModToVH: Selecting virtual hosts for web modules
Specify the virtual host where you want to install the web modules that are contained in
your application. Web modules can be installed on the same virtual host or dispersed among several hosts.
Each element of the MapWebModToVH task consists of the following three fields: "webModule," "uri," "virtualHost."
Of these fields, the following fields might be assigned new values: "virtualHost"and the following are
required: "virtualHost"

The current contents of the task after running default bindings are:
webModule: JavaMail Sample WebApp
uri: mtcomps.war,WEB-INF/web.xml
virtualHost: default_host
```

Examples

- Using Jacl:

```
$AdminApp taskInfo c:/websphere/appserver/installableApps/jmsample.ear MapWebModToVH
```

- Using Jython:

```
print AdminApp.taskInfo('c:/websphere/appserver/installableApps/jmsample.ear', 'MapWebModToVH')
```

uninstall

Use the uninstall command uninstall an existing application.

Target object

None.

Required parameters

application name
Specifies the name of the application to uninstall.

Optional parameters

options
Specifies the options for uninstall.

Sample output

```
ADMA5017I: Uninstallation of myapp started.
ADMA5104I: Server index entry for myCellManager was updated successfully.
ADMA5102I: Deletion of config data for myapp from config
repository completed successfully.
ADMA5011I: Cleanup of temp dir for app myapp done.
ADMA5106I: Application myapp uninstalled successfully.
```

Examples

- Using Jacl:

  ```
  $AdminApp uninstall myApp
  ```

- Using Jython:

  ```
  print AdminApp.uninstall('myApp')
  ```

update

Use the update command to update an application in non-interactive mode. This command supports the addition, removal, and update of application subcomponents or the entire application. Provide the application name, content type, and update options.

Target object

None.

Required parameters

application name
Specifies the name of the application to update.

content type
Use the content type parameter to indicate if you want to update part of the application or the entire application. The following list includes the valid content type values for the update command:

- `app` - Indicates that you want to update the entire application. This option is the same as indicating the update option with the install command. With the `app` value as the content type, you must specify the operation option with update as the value. Provide the new enterprise archive file (EAR) file using the contents option. You can also specify binding information and application options. By default, binding information for installed modules is merged with the binding information for updated modules. To change this default behavior, specify the update.ignore.old or the update.ignore.new options.
- `file` - Indicates that you want to update a single file. You can add, remove, or update individual files at any scope within the deployed application. With the `file` value as the content type, you must perform operations on the file using the `operation` option. Depending on the type of operation, additional options are required. For file additions and updates, you must provide file content and the file URI relative to the root of the EAR file using the contents and contenturi options. For file deletion, you must provide the file URI relative to the root of the EAR file using the contenturi option which is the only required input. Any other options that you provide are ignored.
- `modulefile` - Indicates that you want to update a module. You can add, remove, or update an individual application module. If you specify the `modulefile` value as the content type, you must indicate the operation that you want to perform on the module using the operation option. Depending on the type of operation, further options are required. For installing new modules or updating existing modules in an application, you must indicate the file content and the file URI relative to the root of the EAR file using the contents and contenturi options. You can also specify binding information and application options that pertain to the new or updated modules. For module

updates, the binding information for the installed module is merged with the binding information for the input module by default. To change the default behavior, specify the update.ignore.old or the update.ignore.new options. To delete a module, indicate the file URI relative to the root of the EAR file.

Restriction: Generally, you cannot add or update a module if it depends on other classes outside the module, such as classes within an optional package or library. Client-side processing involves introspecting input module classes and those classes might not load successfully unless class dependencies also are resolved. You can add or update a module if you make the required classes available on a file system that is accessible through wsadmin scripting. Through wsadmin scripting, you can modify the com.ibm.ws.scripting.classpath property in the *profile_root*/`properties/wsadmin.properties` file to include those classes so that the class loader can resolve them.

- `partialapp` - Indicates that you want to update a partial application. Using a subset of application components provided in a compressed `.zip` file format you can update, add, and delete files and modules. The compressed file is not a valid Java 2 platform, Enterprise Edition (J2EE) archive. Instead, it contains application artifacts in the same hierarchical structure as they display in an EAR file. For more information on how to construct the partial application compressed file, see the Java API section. If you indicate the `partialapp` value as the content type, use the contents option to specify the location of the compressed file. When a partial application is provided as an update input, binding information and application options cannot be specified and are ignored, if provided.

Optional parameters

`options`
There are many options available for the update command. For a list of each valid option for the update command, see “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands using wsadmin scripting” on page 898.

Sample output

```
Update of singleFile has started.
ADMA5009I: Application archive extracted at C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\ext
Added files from partial ear: []
performFileOperation: source=C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\ext,
dest=C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\mrg, uri= META-INF/web.xml, op= add
Copying file from C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\ext/META-INF/web.xml to
C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0\mrg\META-INF\web.xml
Collapse list is: []
FileMergeTask completed successfully
ADMA5005I: Application singleFile configured in WebSphere repository
delFiles: []
delM: null
addM: null
Pattern for remove loose and mod:
Loose add pattern: META-INF/[^/]*|WEB-INF/[^/]*|.*wsdl
root file to be copied: META-INF/web.xml to
C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a191b4e\workspace\cells\BAMBIE\applications\
singleFile.ear\deployments\singleFile/META-INF/web.xml
ADMA5005I: Application singleFile configured in WebSphere repository xmlDoc: [#document: null]
root element: [app-delta: null]
****** delta file name: C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a191b4e\workspace\cells\BAMBIE\applications\
singleFile.ear/deltas/delta-1079548405564
ADMA5005I: Application singleFile configured in WebSphere repository
ADMA6011I: Deleting directory tree C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a1960f0
ADMA5011I: Cleanup of temp dir for app singleFile done.
Update of singleFile has ended.
```

Examples

- Using Jacl:
- Using Jython:
- Using Jython list:

updateAccessIDs

Use the updateAccessIDs command to update the access ID information for users and groups that are assigned to various roles that are defined in the application. The system reads the access IDs from the user registry and saves the IDs in the application bindings. This operation improves runtime performance of the application. Use this command after installing an application or after editing security role-specific information for an installed application. This method cannot be invoked when the -conntype option for the wsadmin tool is set to `NONE`. You must be connected to a server to invoke this command.

Target object

None.

Required parameters

application name
Specifies the name of the application of interest.

bALL
The bALL boolean parameter retrieves and saves all access IDs for users and groups in the application bindings. Specify `false` to retrieve access IDs for users or groups that do not have an access ID in the application bindings.

Examples

- Using Jacl or true:

```
$AdminApp updateAccessIDs myapp true
```

- Using Jacl for false:

```
$AdminApp updateAccessIDs myapp false
```

- Using Jython for true:

```
print AdminApp.updateAccessIDs('myapp', 1)
```

- Using Jython for false:

```
print AdminApp.updateAccessIDs('myapp', 0)
```

updateInteractive

Use the updateInteractive command to add, remove, and update application subcomponents or an entire application. When you update an application module or an entire application using interactive mode, the steps that you use to configure binding information are similar to those that apply to the installInteractive command. If you update a file or a partial application, the steps that you use to configure the binding information are not available. In this case, the steps are the same as the ones you use with the update command.

Target object

None.

Required parameters

application name
Specifies the name of the application to update.

content type
Use the content type parameter to indicate if you want to update part of the application or the entire application. The following list includes the valid content type values for the updateInteractive command:

- `app` - Indicates that you want to update the entire application. This option is the same as indicating the update option with the install command. With the `app` value as the content type, you must specify the operation option with update as the value. Provide the new enterprise archive file (EAR) file using the contents option. You can also specify binding information and application options. By default, binding information for installed modules is merged with the binding information for updated modules. To change this default behavior, specify the update.ignore.old or the update.ignore.new options.
- `file` - Indicates that you want to update a single file. You can add, remove, or update individual files at any scope within the deployed application. With the `file` value as the content type, you must perform operations on the file using the `operation` option. Depending on the type of operation, additional options are required. For file additions and updates, you must provide file content and the file URI relative to the root of the EAR file using the contents and contenturi options. For file deletion, you must provide the file URI relative to the root of the EAR file using the contenturi option which is the only required input. Any other options that you provide are ignored.
- `modulefile` - Indicates that you want to update a module. You can add, remove, or update an individual application module. If you specify the `modulefile` value as the content type, you must indicate the operation that you want to perform on the module using the operation option. Depending on the type of operation, further options are required. For installing new modules or updating existing modules in an application, you must indicate the file content and the file URI relative to the root of the EAR file using the contents and contenturi options. You can also specify binding information and application options that pertain to the new or updated modules. For module updates, the binding information for the installed module is merged with the binding information for the input module by default. To change the default behavior, specify the update.ignore.old or the update.ignore.new options. To delete a module, indicate the file URI relative to the root of the EAR file.

  Restriction: Generally, you cannot add or update a module if it depends on other classes outside the module, such as classes within an optional package or library. Client-side processing involves introspecting input module classes and those classes might not load successfully unless class dependencies also are resolved. You can add or update a module if you make the required classes available on a file system that is accessible through wsadmin scripting. Through wsadmin scripting, you can modify the com.ibm.ws.scripting.classpath property in the *profile_root*/`properties`/`wsadmin.properties` file to include those classes so that the class loader can resolve them.
- `partialapp` - Indicates that you want to update a partial application. Using a subset of application components provided in a compressed `.zip` file format you can update, add, and delete files and modules. The compressed file is not a valid Java 2 platform, Enterprise Edition (J2EE) archive. Instead, it contains application artifacts in the same hierarchical structure as they display in an EAR file. For more information on how to construct the partial application compressed file, see the Java API section. If you indicate the `partialapp` value as the content type, use the contents option to specify the location of the compressed file. When a partial application is provided as an update input, binding information and application options cannot be specified and are ignored, if provided.

Optional parameters

options
There are many options available for the updateInteractive command. For a list of each valid option for the updateInteractive command, see “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands using wsadmin scripting” on page 898.

Sample output

```
Getting tasks for: myApp
WASX7266I: A was.policy file exists for this application; would you like to display it? [No]

Task[4]: Binding enterprise beans to JNDI names
Each non message driven enterprise bean in your application or module must be bound to a JNDI name.
EJB Module:  Increment EJB module
```

```
EJB:  Increment
URI:  Increment.jar,META-INF/ejb-jar.xml
JNDI Name:  [Inc]:

Task[10]: Specifying the default data source for
EJB 2.x modules
Specify the default data source for
the EJB 2.x Module containing 2.x CMP beans.

WASX7349I: Possible value for resource authorization is container or per connection factory
EJB Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
JNDI Name:  [DefaultDatasource]:
Resource Authorization:  [Per connection factory]:

Task[12]: Specifying data sources for individual 2.x CMP beans
Specify an optional data source for each 2.x CMP bean. Mapping a specific data source to a CMP bean overrides
the default data source for the module containing the enterprise bean.

WASX7349I: Possible value for resource authorization is container or per connection factory
EJB Module:  Increment EJB module
EJB:  Increment
URI:  Increment.jar,META-INF/ejb-jar.xml
JNDI Name:  [DefaultDatasource]:
Resource Authorization:  [Per connection factory]:container
Setting "Resource Authorization" to "cmpBinding.container"

Task[14]: Selecting Application Servers
Specify the application server where you want to install modules that are contained in your application.
Modules can be installed on the same server or dispersed among several servers.
Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
Server:  [WebSphere:cell=myCell,node=myNode,server=server1]:

Task[16]: Selecting method protections for unprotected methods for 2.x EJB
Specify whether you want to assign security role to the unprotected method, add the method to the exclude list, or mark
the method as unchecked.
EJB Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
Protection Type:  [methodProtection.uncheck]:
Task[18]: Selecting backend ID
Specify the selection for the BackendID
EJB Module:  Increment EJB module
URI:  Increment.jar,META-INF/ejb-jar.xml
BackendId list:  CLOUDSCAPE_V50_1
CurrentBackendId:  [CLOUDSCAPE_V50_1]:

Task[21]: Specifying application options
Specify the various options available to prepare and install your application.
Pre-compile JSP:  [No]:
Deploy EJBs:  [No]:
Deploy WebServices:  [No]:

Task[22]: Specifying EJB deploy options
Specify the options to deploy EJB.
....EJB Deploy option is not enabled.

Task[24]: Copy WSDL files
Copy WSDL files
....This task does not require any user input

Task[25]: Specify options to deploy web services
Specify options to deploy web services
....Web services deploy option is not enabled.
Update of myApp has started.

ADMA5009I: Application archive extracted at C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a48e969\ext/Increment.jar
FileMergeTask completed successfully
ADMA5005I: Application myApp configured in WebSphere repository
delFiles: []
delM: null
addM: [Increment.jar, ]
Pattern for remove loose and mod:
Loose add pattern: META-INF/[^/]*|WEB-INF/[^/]*|.*wsdl
root file to be copied:
META-INF/application.xml to C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/META-INF/application.xml
del files for full module add/update: []
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/ejb-jar.xml
ADMA6016I: Add to workspace Increment.jar/META-INF/ejb-jar.xml
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/MANIFEST.MF
ADMA6016I: Add to workspace Increment.jar/META-INF/MANIFEST.MF
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/ibm-ejb-jar-bnd.xmi
ADMA6016I: Add to workspace Increment.jar/META-INF/ibm-ejb-jar-bnd.xmi
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/Table.ddl
ADMA6016I: Add to workspace Increment.jar/META-INF/Table.ddl
```

```
ADMA6017I: Saved document C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear\deployments\testSM/Increment.jar\META-INF/ibm-ejb-jar-ext.xmi
ADMA6016I: Add to workspace Increment.jar/META-INF/ibm-ejb-jar-ext.xmi
add files for full module add/update: [Increment.jar/META-INF/ejb-jar.xml, Increment.jar/META-INF/MANIFEST.MF,
Increment.jar/META-INF/ibm-ejb-jar-bnd.xmi,
Increment.jar/META-INF/Table.ddl, Increment.jar/META-INF/ibm-ejb-jar-ext.xmi]
ADMA5005I: Application myApp configured in WebSphere repository
xmlDoc: [#document: null]
root element: [app-delta: null]
****** delta file name: C:\asv\b0403.04\WebSphere\AppServer\wstemp\Scriptfb5a487089\workspace\cells\BAMBIE\
applications\testSM.ear/deltas/delta-1079551520393
ADMA5005I: Application myApp configured in WebSphere repository
ADMA6011I: Deleting directory tree C:\DOCUME~1\lavena\LOCALS~1\Temp\app_fb5a48e969
ADMA5011I: Cleanup of temp dir for app myApp done.
Update of myApp has ended.
```

Examples

- Using Jacl:
- Using Jython:
- Using Jython list:

view

Use the view command to view the task that is specified by the task name parameter for the application or module that is specified by the application name parameter. Use -tasknames as the option to get a list of valid task names for the application. Otherwise, specify one or more task names as the option.

Target object

None.

Required parameters

name
Specifies the name of the application or module to view.

Optional parameters

bALL
The bALL boolean parameter retrieves and saves all access IDs for users and groups in the application bindings. Specify `false` to retrieve access IDs for users or groups that do not have an access ID in the application bindings.

-buildVersion
Specifies whether to display the build version of the application of interest.

Sample output

The command returns the following information if you specify the `taskoptions` value for the task name parameter:

```
MapModulesToServers
MapWebModToVH
MapRolesToUsers
```

The command returns the following information if you specify the `mapModulesToServers` task for the task name parameter:

```
MapModulesToServers: Selecting Application Servers

Specify the application server where you want to install the modules that are contained in your
application. Modules can be installed on the same server or dispersed among several servers:

Module:  adminconsole
URI:  adminconsole.war,WEB-INF/web.xml
Server:  WebSphere:cell=juniartiNetwork,
node=juniartiManager,server=dmgr
```

Examples

The following view command example lists each available task name:

- Using Jacl:

```
$AdminApp view DefaultApplication {-tasknames}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', ['-tasknames'])
```

The following view command example returns information for the mapModulesToServer task:

- Using Jacl:

```
$AdminApp view DefaultApplication {-MapModulesToServers}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', ['-MapModulesToServers'])
```

The following view command example returns information for the AppDeploymentOptions task:

- Using Jacl:

```
$AdminApp view DefaultApplication {-AppDeploymentOptions}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', '-AppDeploymentOptions')
```

The following view command example returns the build version for the DefaultApplication application:

- Using Jacl:

```
$AdminApp view DefaultApplication {-buildVersion}
```

- Using Jython:

```
print AdminApp.view('DefaultApplication', '-buildVersion')
```

Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands using wsadmin scripting

This topic lists the available options for the install, installInteractive, edit, editInteractive, update, and updateInteractive commands of the AdminApp object.

You can use the commands for the AdminApp object to install, edit, update, and manage your application configurations. This topic provides additional options to use with the install, installInteractive, edit, editInteractive, update, and updateInteractive commands to administer your applications. The options listed in this topic apply to all of these commands except where noted.

You can set or update a configuration value using options in batch mode. To identify which configuration object is to be set or updated, the values of read only fields are used to find the corresponding configuration object. All the values of read only fields have to match with an existing configuration object, otherwise the command fails.

You can use pattern matching to simplify the task of supplying required values for certain complex options. Pattern matching only applies to fields that are required or read only.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

The following options are available for the install, installInteractive, edit, editInteractive, update, and updateInteractive commands:

ActSpecJNDI

The ActSpecJNDI option binds Java 2 Connector (J2C) activation specifications to destination Java Naming and Directory Interface (JNDI) names. You can optionally bind J2C activation specifications in your application or module to a destination JNDI name. You can assign a value to each of the following elements of the ActSpecJNDI option: RARModule, uri, j2cid, and j2c.jndiName fields. The contents of the option after running default bindings include:

- RARModule: <rar module name>
- uri: <rar name>,META-INF/ra.xml
- Object identifier: <messageListenerType>
- JNDI name: null

To use this option, you must specify the `Destination` property in the `ra.xml` file and set the introspected type of the `Destination` property as `javax.jms.Destination`

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $embeddedEar {-ActSpecJNDI
 {{"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml javax.jms.MessageListener jndi5}
 {"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml javax.jms.MessageListener2 jndi6}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $embeddedEar {-ActSpecJNDI
 {{.* .*.rar,.* javax.jms.MessageListener jndi5}
 {.*  .*.rar,.*  javax.jms.MessageListener2 jndi6}}}
```

Using Jython:

```
AdminApp.install(embeddedEar, ['-ActSpecJNDI',
 [["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageListener', 'jndi5'],
 ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageListener2', 'jndi6']]])
```

Using Jython with pattern matching:

```
AdminApp.install(embeddedEar, ['-ActSpecJNDI',
 [['.*', '.*.rar,.*', 'javax.jms.MessageListener', 'jndi5'],
 ['.*', '.*.rar,.*', 'javax.jms.MessageListener2', 'jndi6']]])
```

allowDispatchRemoteInclude

The allowDispatchRemoteInclude option enables an enterprise application to dispatch includes to resources across web modules that are in different Java virtual machines in a managed node environment through the standard request dispatcher mechanism.

Batch mode example usage

Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:RRDEnabledAppname/]
set deploymentObject [$AdminConfig showAttribute $deployments deployedObject]
set rrdAttr [list allowDispatchRemoteInclude true]
set attrs [list $rrdLocalAttr]
$AdminConfig modify $deploymentObject $attrs
```

Using Jython:

```
deployments = AdminConfig.getid ('/Deployment:RRDEnabledAppname/')
deploymentObject = AdminConfig.showAttribute (deployments, 'deployedObject')
rrdAttr = ['allowDispatchRemoteInclude', 'true']
attrs = [rrdLocalAttr]
AdminConfig.modify (deploymentObject, attrs)
```

allowPermInFilterPolicy

The allowPermInFilterPolicy option specifies that the application server should continue with the application deployment process even when the application contains policy permissions that are in the filter policy. This option does not require a value.

allowServiceRemoteInclude

The allowServiceRemoteInclude option enables an enterprise application to service an include request from an enterprise application with the `allowDispatchRemoteInclude` option set to `true`.

Batch mode example usage

Using Jacl:

```
set deployments [$AdminConfig getid /Deployment:RRDEnabledAppname/]
set deploymentObject [$AdminConfig showAttribute $deployments deployedObject]
set rrdAttr [list allowServiceRemoteInclude true]
set attrs [list $rrdAttr]
$AdminConfig modify $deploymentObject $attrs
```

Using Jython:

```
deployments = AdminConfig.getid ('/Deployment:RRDEnabledAppname/')
deploymentObject = AdminConfig.showAttribute ('deployments', 'deployedObject')
rrdAttr = ['allowServiceRemoteInclude', 'true']
attrs = [rrdAttr]
AdminConfig.modify (deploymentObject, attrs)
```

appname

The appname option specifies the name of the application. The default value is the display name of the application.

BackendIdSelection

The BackendIdSelection option specifies the backend ID for the enterprise bean Java archive (JAR) modules that have container-managed persistence (CMP) beans. An enterprise bean JAR module can support multiple backend configurations as specified using an application assembly tool. Use this option to change the backend ID during installation.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-BackendIdSelection
 {{Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml DB2UDBNT_V72_1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-BackendIdSelection
 {{.* Annuity20EJB.jar,.* DB2UDBNT_V72_1}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-BackendIdSelection
 [[Annuity20EJB Annuity20EJB.jar,META-INF/ejb-jar.xml DB2UDBNT_V72_1]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-BackendIdSelection',
 [['.*', 'Annuity20EJB.jar,.*', 'DB2UDBNT_V72_1']]])
```

BindJndiForEJBBusiness

The BindJndiForEJBBusiness option binds EJB modules with business interfaces to JNDI names. Ensure that each EJB module with business interfaces is bound to a JNDI name.

The current contents of the option after running default bindings include:

- EJBModule: SampleModule
- EJB: SampleEJB
- URI: sample.jar,META-INF/ejb-jar.xml
- Business interface: com.ibm.sample.business.bnd.LocalTargetOne
- JNDI name: []: ejblocal:ejb/LocalTargetOne

If you specify the target resource JNDI name using the BindJndiForEJBNonMessageBinding option, do not specify a business interface JNDI name in the BindJndiForEJBBusiness option. If you do not specify the target resource JNDI name, specify a business interface JNDI name. If you do not specify a business interface JNDI name, the run time provides a container default.

For a no-interface view, the business interface value is an empty string ("").

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $ear {-BindJndiForEJBBusiness {{ ejb_1.jar ejb1
  ejb_1.jar,META-INF/ejb-jar.xml test.ejb1 jndi1 }}}
```

Using Jacl with pattern matching:

```
$AdminApp install $ear {-BindJndiForEJBBusiness {{ .* .* .*.jar,.* test.ejb1 jndi1 }}}
```

Using Jython:

```
AdminApp.install(ear, '[-BindJndiForEJBBusiness [[ejb_1.jar ejb1
  ejb_1.jar,META-INF/ejb-jar.xml test.ejb1 jndi1 ]]]')
```

Using Jython with pattern matching:

```
AdminApp.install(ear, '[-BindJndiForEJBBusiness [[.* .* .*.jar,.* test.ejb1 jndi1]]]')
```

BindJndiForEJBMessageBinding

The BindJndiForEJBMessageBinding option binds message-driven enterprise beans to listener port names or to activation specification Java Naming and Directory Interface (JNDI) names. Use this option to provide missing data or update a task. When a message-driven enterprise bean is bound to an activation specification JNDI name, you can also specify the destination JNDI name and authentication alias.

Each element of the BindJndiForEJBMessageBinding option consists of the following fields: EJBModule, EJB, uri, listenerPort, JNDI, jndi.dest, and actspec.auth. Some of these fields can be assigned values: listenerPort, JNDI, jndi.dest, and actspec.auth.

The current contents of the option after running default bindings include:

- EJBModule: Ejb1
- EJB: MessageBean
- URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml
- Listener port: [null]:
- JNDI name: [eis/MessageBean]:
- Destination JNDI Name: [jms/TopicName]:
- ActivationSpec Authentication Alias: [null]:

The default Destination JNDI Name is collected from the corresponding message reference.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding {{Ejb1 MessageBean
 ejb-jar-ic.jar,META-INF/ejb-jar.xml myListenerPort jndi1 jndiDest1 actSpecAuth1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $ear {-BindJndiForEJBMessageBinding {{.*  .*
  .*.jar,.* myListenerPort jndi1 jndiDest1 actSpecAuth1}}}
```

Using Jython:

```
AdminApp.install( ear, ['-BindJndiForEJBMessageBinding',
 [['Ejb1', 'MessageBean', 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'myListenerPort',
 'jndi1', 'jndiDest1', 'actSpecAuth1']]])
```

Using Jython with pattern matching:

```
AdminApp.install( ear, ['-BindJndiForEJBMessageBinding',
 [['.*', '.*', '.*.jar,.*', 'myListenerPort', 'jndi1', 'jndiDest1', 'actSpecAuth1']]])
```

BindJndiForEJBNonMessageBinding

The BindJndiForEJBNonMessageBinding option binds enterprise beans to Java Naming and Directory Interface (JNDI) names. Ensure each non message-driven enterprise bean in your application or module is bound to a JNDI name. Use this option to provide missing data or update a task.

The current contents of the option after running default bindings include:

- EJBModule: Ejb1
- EJB: MessageBean
- URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml
- Target Resource JNDI Name: [com.acme.ejbws.AnnuityMgmtSvcEJB2xJAXRPC]
- Local Home JNDI Name: [null]
- Remote Home JNDI Name: [null]

Special constraints exist for Enterprise JavaBeans (EJB) 3.0 and later modules. If you specify the target resource JNDI name, do not specify the local home or remote home JNDI names. You also cannot specify the JNDI for business interfaces field in the BindJndiForEJBBusiness option. If you do not specify the target resource JNDI name, then the local and remote home JNDI name fields are optional. If you do not specify local and remote JNDI names, the run time provides a container default.

If you do not use EJB 3.0 modules, you must specify the target resource JNDI name.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install /myapp.ear {-BindJndiForEJBNonMessageBinding {{Ejb1 MessageBean
  ejb-jar-ic.jar,META-INF/ejb-jar.xml com.acme.ejbws.AnnuityMgmtSvcEJB2xJAXRPC ""  ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/samples/SCA/installableApps/JobbankTargetEJBApp.ear
 {-BindJndiForEJBNonMessageBinding {{.* .* ResumeBankAppEJB.jar,META-INF/ejb-jar.xml
 ejb/com/app/ResumeBank/ResumeBank "" "" }}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-BindJndiForEJBNonMessageBinding [[Ejb1 MessageBean
  ejb-jar-ic.jar,META-INF/ejb-jar.xml com.acme.ejbws.AnnuityMgmtSvcEJB2xJAXRPC "" ""]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/samples/SCA/installableApps/JobbankTargetEJBApp.ear',
 '[ -BindJndiForEJBNonMessageBinding [[ .* .* ResumeBankAppEJB.jar,META-INF/ejb-jar.xml
 ejb/com/app/ResumeBank/ResumeBank "" "" ]] ]' )
```

blaname

Use the blaname option to specify the name of business level application under which the system creates the Java EE application. This option is optional. If you do not specify a value, the system sets the name as the Java EE application name. This option is available only with the install command.

buildVersion

The buildVersion option displays the build version of an application EAR file. You cannot modify this option because it is read-only. This option returns the build version information for an application EAR if you have specified the build version in the `MANIFEST.MF` application EAR file.

cell

The cell option specifies the cell name to install or update an entire application, or to update an application in order to add a new module. If you want to update an entire application, this option only applies if the application contains a new module that does not exist in the installed application.

Batch mode example usage

Using Jacl:

```
$AdminApp install "/samples/SCA/installableApps/JobbankTargetEJBApp.ear" {-cell cellName}
```

Using Jython:

```
AdminApp.install('/samples/SCA/installableApps/JobbankTargetEJBApp.ear', '[-cell cellName]')
```

clientMode

Note: The clientMode option specifies whether to deploy client modules to an isolated deployment target (`isolated`), a federated node of a deployment manager (`federated`), or an application server (`server_deployed`). If you specify this option, install the client modules only onto a Version 8.0 deployment target.

The default value is `isolated`.

The choice of client deployment mode affects how `java:` lookups are handled. All Java URL name spaces (global, application, module, and component) are local in isolated client processes. The name spaces reside on a server in federated and server deployed client processes. The server chosen as a target for a client module determines where those name spaces are created. All `java:` lookups for federated or server deployed client modules are directed to the target server. The client module does not actually run in the target server. Multiple instances of the same client module will all share the component name space in the federated and server deployed modes. Choosing the federated mode is simply a declaration of intent to launch the client module using Java Network Launching Protocol (JNLP), but the Java Naming and Directory Interface (JNDI) mechanics of federated and server deployed modes are the same.

contents

The contents option specifies the file that contains the content that you want to update. For example, depending on the content type, the file could be an EAR file, a module, a partial zip, or a single file. The path to the file must be local to the scripting client. The `contents` option is required unless you have specified the `delete` option.

contenturi

The contenturi option specifies the URI of the file that you are adding, updating, or removing from an application. This option only applies to the update command. The `contenturi` option is required if the content type is `file` or `modulefile`. This option is ignored for other content types.

contextroot

The contextroot option specifies the context root that you use when installing a stand-alone web application archive (WAR) file.

CorrectOracleIsolationLevel

The CorrectOracleIsolationLevel option specifies the isolation level for the Oracle type provider. Use this option to provide missing data or to update a task. The last field of each entry specifies the isolation level. Valid isolation level values are `2` or `4`.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You only need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-CorrectOracleIsolationLevel
 {{AsyncSender jms/MyQueueConnectionFactory jms/Resource1 2}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-CorrectOracleIsolationLevel
  {{.*  jms/MyQueueConnectionFactory jms/Resource1 2}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-CorrectOracleIsolationLevel
 [[AsyncSender jms/MyQueueConnectionFactory jms/Resource1 2]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-CorrectOracleIsolationLevel',
 [['.*', 'jms/MyQueueConnectionFactory', 'jms/Resource1', 2]]])
```

CorrectUseSystemIdentity

The CorrectUseSystemIdentity option replaces RunAs System to RunAs Roles. The enterprise beans that you install contain a RunAs system identity. You can optionally change this identity to a RunAs role. Use this option to provide missing data or update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-CorrectUseSystemIdentity {{Inc "Increment Bean Jar"
 Increment.jar,META-INF/ejb-jar.xml getValue() RunAsUser2 user2 password2}
 {Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml Increment()
 RunAsUser2 user2 password2}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-CorrectUseSystemIdentity
 {{.*  .*  .*  getValue() RunAsUser2 user2 password2}
 {.*  .*  .*  Increment() RunAsUser2 user2 password2}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-CorrectUseSystemIdentity
 [[Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml getValue()
 RunAsUser2 user2 password2]
 [Inc "Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml Increment()
 RunAsUser2 user2 password2]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-CorrectUseSystemIdentity',
 [['.*', '.*', '.*', 'getValue()', 'RunAsUser2', 'user2', 'password2'],
 ['.*', '.*', '.*', 'Increment()', 'RunAsUser2', 'user2', 'password2']]])
```

createMBeansForResources

The createMBeansForResources option specifies that MBeans are created for all resources, such as servlets, JavaServer Pages (JSP) files, and enterprise beans, that are defined in an application when the application starts on a deployment target. This option does not require a value. The default setting is the nocreateMBeansForResources option.

CtxRootForWebMod

The CtxRootForWebMod option edits the context root of the web module. You can edit a context root that is defined in the `application.xml` file using this option. The current contents of this option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- ContextRoot: <context root>

If the web module is a Servlet 2.5, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname ivtApp -CtxRootForWebMod
 {{"IVT Application" ivt_app.war,WEB-INF/web.xml /mycontextroot}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -CtxRootForWebMod {{.* .* /mycontextroot}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'ivtApp ', '-CtxRootForWebMod',
 [["IVT Application", 'ivt_app.war,WEB-INF/web.xml', '/mycontextroot']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-CtxRootForWebMod',
 [['.*', '.*', '/mycontextroot']]])
```

custom

The custom option specifies a name-value pair using the format `name=value`. Use the `custom` option to pass options to application deployment extensions. See the application deployment extension documentation for available custom options.

CustomActivationPlan

The CustomActivationPlan option specifies runtime components to add or remove from the default runtime components that are used to run the application. Only use this option when the application server can not obtain all necessary runtime components by inspecting the application.

Batch mode example usage

Using Jython:

```
AdminApp.install('Increment.jar','[-CustomActivationPlan [["Increment Enterprise Java Bean"
 Increment.jar,META-INF/ejb-jar.xml WebSphere:specname=WS_ComponentToAdd ""]]')
```

Using Jython with pattern matching:

```
AdminApp.install('Increment.jar','[-CustomActivationPlan [[.*
 Increment.jar,META-INF/ejb-jar.xml WebSphere:specname=WS_ComponentToAdd ""]]')
```

Table 576. CustomActivationPlan components to add. Specify a WS_ComponentToAdd value in the command.

Component	WS_ComponentToAdd value
EJB container	WS_EJBContainer
Portlet container	WS_PortletContainer
JavaServer Faces (JSF)	WS_JSF
SIP container	WS_SipContainer
Compensation scope service	WS_Compensation
Application profile	WS_AppProfile
Activity session	WS_ActivitySession
Internationalization	WS_I18N
Startup beans	WS_StartupService

DataSourceFor10CMPBeans

The DataSourceFor10CMPBeans option specifies optional data sources for individual 1.x container-managed persistence (CMP) beans. Use this option to provide missing data or to update a task.

Mapping a specific data source to a CMP bean overrides the default data source for the module that contains the enterprise bean. Each element of the DataSourceFor10CMPBeans option consists of the following fields: EJBModule, EJB, uri, JNDI, userName, password, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, userName, password, login.config.name, and auth.props.

The current contents of the option after running default bindings include:

- EJBModule: Increment CMP 1.1 EJB
- EJB: IncCMP11
- URI: IncCMP11.jar,META-INF/ejb-jar.xml
- JNDI name: [DefaultDatasource]:
- User name: [null]:

- Password: [null]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias. The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+) .

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-DataSourceFor10CMPBeans {{"Increment CMP 1.1 EJB" IncCMP11
 IncCMP11.jar,META-INF/ejb-jar.xml myJNDI user1 password1 loginName1 authProps1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-DataSourceFor10CMPBeans {{.*  .*  IncCMP11.jar,.*
 myJNDI user1 password1 loginName1 authProps1}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-DataSourceFor10CMPBeans', [["Increment CMP 1.1 EJB",
 'IncCMP11', 'IncCMP11.jar,META-INF/ejb-jar.xml', 'myJNDI', 'user1',
 'password1', 'loginName1', 'authProps1']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-DataSourceFor10CMPBeans', [['.*', '.*',
 'IncCMP11.jar,.*', 'myJNDI', 'user1', 'password1', 'loginName1', 'authProps1']]])
```

DataSourceFor20CMPBeans

The DataSourceFor20CMPBeans option specifies optional data sources for individual 2.x container-managed persistence (CMP) beans. Use this option to provide missing data or to update a task.

Mapping a specific data source to a CMP bean overrides the default data source for the module that contains the enterprise bean. Each element of the DataSourceFor20CMPBeans option consists of the following fields: EJBModule, EJB, uri, JNDI, resAuth, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, resAuth, login.config.name, and auth.props.

The current contents of the option after running default bindings includes the following:

- EJBModule: Increment enterprise bean
- EJB: Increment
- URI: Increment.jar,META-INF/ejb-jar.xml
- JNDI name: [null]:
- Resource authorization: [Per application]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

- Properties: []: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias. The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+) .

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application. You only need to provide data for rows or entries that are missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install Increment.jar {-DataSourceFor20CMPBeans
 {{"Increment Enterprise Java Bean" Increment
 Increment.jar,META-INF/ejb-jar.xml jndi1 container "" ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install Increment.jar {-DataSourceFor20CMPBeans {{.*  .*
 Increment.jar,.*  jndi1 container "" ""}}}
```

Using Jython:

```
AdminApp.install('Increment.jar', ['-DataSourceFor20CMPBeans',
 [["Increment Enterprise Java Bean", 'Increment',
 'Increment.jar,META-INF/ejb-jar.xml', 'jndi1', 'container', '', '']]])
```

Using Jython with pattern matching:

```
AdminApp.install('Increment.jar', ['-DataSourceFor20CMPBeans', [['.*', '.*',
 'Increment', 'Increment.jar,.*', 'jndi1', 'container', '', '']]])
```

DataSourceFor10EJBModules

The DataSourceFor10EJBModules option specifies the default data source for the enterprise bean module that contains 1.x container-managed persistence (CMP) beans. Use this option to provide missing data or update a task.

Each element of the DataSourceFor10EJBModules option consists of the following fields: EJBModule, uri, JNDI, userName, password, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, userName, password, login.config.name, and auth.props.

The current contents of the option after running default bindings include:

- EJBModule: Increment CMP 1.1 enterprise bean
- uri: IncCMP11.jar,META-INF/ejb-jar.xml
- JNDI name: [DefaultDatasource]:
- User name: [null]:
- Password: [null]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias. The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+).

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-DataSourceFor10EJBModules {{"Increment CMP 1.1 EJB"
 IncCMP11.jar,META-INF/ejb-jar.xml yourJNDI user2 password2 loginName authProps}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-DataSourceFor10EJBModules
 {{.* IncCMP11.jar,.* yourJNDI user2 password2 loginName authProps}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-DataSourceFor10EJBModules', [["Increment CMP 1.1 EJB",
 'IncCMP11.jar,META-INF/ejb-jar.xml', 'yourJNDI', 'user2', 'password2',
 'loginName', 'authProps']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-DataSourceFor10EJBModules',
 [['.*', 'IncCMP11.jar,.*', 'yourJNDI', 'user2', 'password2', 'loginName', 'authProps']]])
```

DataSourceFor20EJBModules

The DataSourceFor20EJBModules option specifies the default data source for the enterprise bean 2.x module that contains 2.x container managed persistence (CMP) beans. Use this option to provide missing data or update a task.

Each element of the DataSourceFor20EJBModules option consists of the following fields: EJBModule, uri, JNDI, resAuth, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, resAuth, login.config.name, auth.props, and extended datasource properties.

The current contents of the option after running default bindings include:

- EJBModule: Increment enterprise bean
- URI: Increment.jar,META-INF/ejb-jar.xml
- JNDI name: [DefaultDatasource]:
- Resource authorization: [Per application]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: []: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Extended Data source properties: []: Use this option so that a data source that uses heterogeneous pooling can connect to a DB2 database. The pattern for the property is `property1=value1+property2=value2`.

The last field in each entry of this task specifies the value for resource authorization. Valid values for resource authorization are `per connection factory` or `container`.

If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias. The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is websphere:name= <name1>,value=<value1>,description=<desc1>. Specify multiple properties using the plus sign (+) .

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or require update.

Batch mode example usage

Using Jacl:

```
$AdminApp install Increment.jar {-DataSourceFor20EJBModules
 {{"Increment Enterprise Java Bean" Increment.jar,META-INF/ejb-jar.xml
 jndi2 container "" "" ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install Increment.jar {-DataSourceFor20EJBModules {{.*
 Increment.jar,.* jndi2 container "" "" ""}}}
```

Note: If you use multiple values for extended data source properties, you must assign the values to a variable and use that variable for the property value. For example:

```
set s \"value1,value2\"
```

In the command substitution, reference the variable as shown in the following example:

```
"clientApplicationInformation=value1,value2" with "clientapplication=$s"
```

Using Jython:

```
AdminApp.install('Increment.jar', ['-DataSourceFor20EJBModules',
 [["Increment Enterprise Java Bean", 'Increment.jar,META-INF/ejb-jar.xml',
 'jndi2', 'container', '', '', '']]])
```

Using Jython with pattern matching:

```
AdminApp.install('Increment.jar', ['-DataSourceFor20EJBModules',
 [['.*', 'Increment.jar,.*', 'jndi2', 'container', '', '', '']]])
```

Note: If you use multiple values for extended data source properties, surround the two values with double quote characters. For example:

```
'property1="value1,value2"+property2="value3,value4"'
```

defaultbinding.cf.jndi

The defaultbinding.cf.jndi option specifies the Java Naming and Directory Interface (JNDI) name for the default connection factory.

defaultbinding.cf.resauth

The defaultbinding.cf.resauth option specifies the RESAUTH for the connection factory.

defaultbinding.datasource.jndi

The defaultbinding.datasource.jndi option specifies the Java Naming and Directory Interface (JNDI) name for the default data source.

defaultbinding.datasource.password

The defaultbinding.datasource.password option specifies the password for the default data source.

defaultbinding.datasource.username

The defaultbinding.datasource.username option specifies the user name for the default data source.

defaultbinding.ejbjndi.prefix

The defaultbinding.ejbjndi.prefix option specifies the prefix for the enterprise bean Java Naming and Directory Interface (JNDI) name.

defaultbinding.force

The defaultbinding.force option specifies that the default bindings override the current bindings.

defaultbinding.strategy.file

The defaultbinding.strategy.file option specifies a custom default bindings strategy file.

defaultbinding.virtual.host

The defaultbinding.virtual.host option specifies the default name for a virtual host.

depl.extension.reg (deprecated)

Note: The depl.extension.reg option is deprecated. No replication option is available.

deployejb

The deployejb option specifies to run the EJBDeploy tool during installation. This option does not require a value. The EJBDeploy runs during installation of EJB 1.x or 2.x modules. The EJB deployment tool does not run during installation of EJB 3.x modules.

If you pre-deploy the application enterprise archive (EAR) file using the EJBDeploy tool then the default value is `nodeployejb`. If not, the default value is `deployejb`.

You must use this deployejb option in the following situations:
- The EAR file was assembled using an assembly tool such as Rational® Application Developer and the EJBDeploy tool was not run during assembly.
- The EAR file was not assembled using an assembly tool such as Rational Application Developer.
- The EAR file was assembled using versions of the Application Assembly Tool (AAT) previous to Version 5.0.

If an EJB module is packaged in a web archive (WAR), you do not need to use this deployejb option.

For this option, install only onto a Version 6.1 or later deployment target.

deployejb.classpath

The deployejb.classpath option specifies an extra class path for the EJBDeploy tool.

deployejb.complianceLevel

The deployejb.complianceLevel option specifies the JDK compliance level for the EJBDeploy tool.

Possible values include:

`1.4 (default) 5.0 6.0`

For a list of currently supported JDK compliance levels, run the `ejbdeploy -?` command.

deployejb.dbschema

The deployejb.dbschema option specifies the database schema for the EJBDeploy tool.

deployejb.dbtype

The deployejb.dbtype option specifies the database type for the EJBDeploy tool.

Possible values include:

```
DB2UDB_V81 DB2UDB_V82 DB2UDB_V91 DB2UDB_V95 DB2UDBOS390_V8 DB2UDBOS390_NEWFN_V8 DB2UDBOS390_V9
 DB2UDBISERIES_V53 DB2UDBISERIES_V54 DB2UDBISERIES_V61  DERBY_V10 DERBY_V101 INFORMIX_V100 INFORMIX_V111 INFORMIX_V115
 MSSQLSERVER_2005 ORACLE_V10G ORACLE_V11G SYBASE_V15 SYBASE_V125
```

The following databases support Structured Query Language in Java (SQLJ): DB2UDB_V82, DB2UDB_V81, DB2UDBOS390_V7, and DB2UDBOS390_V8.

For a list of supported database vendor types, run the `ejbdeploy -?` command.

deployejb.dbaccesstype

The deployejb.dbaccesstype option specifies the type of database access for the EJBDeploy tool. Valid values are `SQLj` and `JDBC`. The default is `JDBC`.

deployejb.rmic

The deployejb.rmic option specifies extra RMIC options to use for the EJBDeploy tool.

deployejb.sqljclasspath

The deployejb.sqljclasspath option specifies the location of the SQLJ translator classes.

deployws

The deployws option specifies to deploy web services during installation. This option does not require a value.

The default value is: `nodeployws`.

deployws.classpath

The deployws.classpath option specifies the extra class path to use when you deploy web services.

To specify the class paths of multiple entries, separate the entries with the same separator that is used with the CLASSPATH environment variable.

deployws.jardirs

The deployws.jardirs option specifies the extra extension directories to use when you deploy web services.

distributeApp

The distributeApp option specifies that the application management component distribute application binaries. This option does not require a value. This setting is the default.

EmbeddedRar

The EmbeddedRar option binds Java 2 Connector objects to JNDI names. You must bind each Java 2 Connector object in your application or module, such as, J2C connection factories, J2C activation specifications and J2C administrative objects, to a JNDI name. Each element of the EmbeddedRar option contains the following fields: `RARModule`, `uri`, `j2cid`, `j2c.name`, `j2c.jndiName`. You can assign the following values to the fields: `j2c.name`, `j2c.jndiName`.

The current contents of the option after running default bindings include:

```
Module: <rar module name>
URI: <rar name>,META-INF/ra.xml
Object identifier: <identifier of the J2C object>
name: j2cid
JNDI name: eis/j2cid
```

Where j2cid is:

```
J2C connection factory: connectionFactoryInterface
J2C admin object: adminObjectInterface
J2C activation specification: message listener type
```

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

If the ID is not unique in the `ra.xml` file, -<number> will be added. For example, javax.sql.DataSource-2.

Batch mode example usage

Using Jacl:

```
$AdminApp install $embeddedEar {-EmbeddedRar {{"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml
 javax.sql.DataSource javax.sql.DataSource1 eis/javax.sql.javax.sql.DataSSource1} {"FVT Resource Adapter"
 jca15cmd.rar,META-INF/ra.xml javax.sql.DataSource2 javax.sql.DataSource2 eis/javax.sql.DataSource2} {"FVT Resource Adapter"
 jca15cmd.rar,META-INF/ra.xml javax.jms.MessageListener javax.jms.MessageListener1 eis/javax.jms.MessageListener1} {"FVT Resource
 Adapter" jca15cmd.rar,META-INF/ra.xml javax.jms.MessageLListener2 javax.jms.MessageListener2 eis/javax.jms.MessageListener2}
 {"FVT Resource Adapter" jca15cmd.rar,META-INF/ra.xml fvt.adapter.message.FVTMessageProvider
 fvt.adapter.message.FVTMessageProvider1 eis/fvt.adapter.message.FVTMessageProvider1} {"FVT Resource Adapter"
 jca15cmd.rar,META-INF/ra.xml fvt.adapter.message.FVTMessageProvider2 fvt.adapter.message.FVTMessageProvider2
 eis/fvt.adapter.message.FVTMessageProvider2}}}
```

Using Jacl with pattern matching:

```
$AdminApp install $embeddedEar {-EmbeddedRar {{.*  .*  .*  javax.sql.DataSource1
 eis/javax.sql.javax.sql.DataSSource1} {.*  .*  .*  javax.sql.DataSource2  eis/javax.sql.DataSource2} {.*  .*  .*
 javax.jms.MessageListener1  eis/javax.jms.MessageListener1} {.*  .*  .*  javax.jms.MessageListener2
 eis/javax.jms.MessageListener2} {.*  .*  .*  fvt.adapter.message.FVTMessageProvider1
 eis/fvt.adapter.message.FVTMessageProvider1} {.*  .*  .*  fvt.adapter.message.FVTMessageProvider2
 eis/fvt.adapter.message.FVTMessageProvider2}}}
```

Using Jython:

```
AdminApp.install(embeddedEar, ['-EmbeddedRar', [["FVT Resource Adapter",
 'jca15cmd.rar,META-INF/ra.xml', 'javax.sql.DataSource', 'javax.sql.DataSource1', 'eis/javax.sql.javax.sql.DataSSource1'], ["FVT
 Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml javax.sql.DataSource2', 'javax.sql.DataSource2', 'eis/javax.sql.DataSource2'],
 ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageListener', 'javax.jms.MessageListener1',
 'eis/javax.jms.MessageListener1'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml', 'javax.jms.MessageLListener2',
 'javax.jms.MessageListener2', 'eis/javax.jms.MessageListener2'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml
 fvt.adapter.message.FVTMessageProvider', 'fvt.adapter.message.FVTMessageProvider1',
 'eis/fvt.adapter.message.FVTMessageProvider1'], ["FVT Resource Adapter", 'jca15cmd.rar,META-INF/ra.xml',
 'fvt.adapter.message.FVTMessageProvider2', 'fvt.adapter.message.FVTMessageProvider2',
 'eis/fvt.adapter.message.FVTMessageProvider2']]])
```

Using Jython with pattern matching:

```
AdminApp.install(embeddedEar, ['-EmbeddedRar', [['.*', '.*', '.*', 'javax.sql.DataSource1',
 'eis/javax.sql.javax.sql.DataSSource1'], ['.*', '.*', '.*', 'javax.sql.DataSource2', 'eis/javax.sql.DataSource2'], ['.*', '.*',
 '.*', 'javax.jms.MessageListener1', 'eis/javax.jms.MessageListener1'], ['.*', '.*', '.*', 'javax.jms.MessageListener2',
 'eis/javax.jms.MessageListener2'], ['.*', '.*', '.*', 'fvt.adapter.message.FVTMessageProvider1',
 'eis/fvt.adapter.message.FVTMessageProvider1'], ['.*', '.*', '.*', 'fvt.adapter.message.FVTMessageProvider2',
 'eis/fvt.adapter.message.FVTMessageProvider2']]])
```

enableClientModule

Note: The enableClientModule option specifies to deploy client modules. Select this option if the file to deploy has one or more client modules that you want to deploy. If you select this option, install the client modules only onto a Version 8.0 deployment target.

This option does not require a value. The default value is noenableClientModule. By default, the enableClientModule option is ignored during deployment and client modules are not deployed.

EnsureMethodProtectionFor10EJB

The EnsureMethodProtectionFor10EJB option selects method protections for unprotected methods of 1.x enterprise beans. Specify to leave the method as unprotected, or assign protection which denies all access. Use this option to provide missing data or to update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install Increment.jar {-EnsureMethodProtectionFor10EJB
 {{"Increment EJB Module" IncrementEJBBean.jar,META-INF/ejb-jar.xml ""}
 {"Timeout EJB Module" TimeoutEJBBean.jar,META-INF/ejb-jar.xml
 methodProtection.denyAllPermission}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-EnsureMethodProtectionFor10EJB
 {{.* IncrementEJBBean.jar,.* ""} {.* TimeoutEJBBean.jar,.*
 methodProtection.denyAllPermission}}}
```

Using Jython:

```
AdminApp.install('Increment.jar', '[-EnsureMethodProtectionFor10EJB
 [["Increment EJB Module" IncrementEJBBean.jar,META-INF/ejb-jar.xml ""]
 ["Timeout EJB Module" TimeoutEJBBean.jar,META-INF/ejb-jar.xml
 methodProtection.denyAllPermission]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-EnsureMethodProtectionFor10EJB',
 [['.*', 'IncrementEJBBean.jar,.*', ""], ['.*', 'TimeoutEJBBean.jar,.*',
 'methodProtection.denyAllPermission']]])
```

The last field in each entry of this task specifies the value of the protection. Valid protection values include: `methodProtection.denyAllPermission`. You can also leave the value blank if you want the method to remain unprotected.

EnsureMethodProtectionFor20EJB

The EnsureMethodProtectionFor20EJB option selects method protections for unprotected methods of 2.x enterprise beans. Specify to assign a security role to the unprotected method, add the method to the exclude list, or mark the method as cleared. You can assign multiple roles for a method by separating roles names with commas. Use this option to provide missing data or to update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or require an update the existing data.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-EnsureMethodProtectionFor20EJB
 {{CustmerEjbJar customerEjb.jar,META-INF/ejb-jar.xml methodProtection.uncheck}
 {SupplierEjbJar supplierEjb.jar,META-INF/ejb-jar.xml methodProtection.exclude}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-EnsureMethodProtectionFor20EJB
 {{.* customerEjb.jar,.* methodProtection.uncheck}
 {.* supplierEjb.jar,.* methodProtection.exclude}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-EnsureMethodProtectionFor20EJB
 [[CustmerEjbJar customerEjb.jar,META-INF/ejb-jar.xml methodProtection.uncheck]
 [SupplierEjbJar supplierEjb.jar,META-INF/ejb-jar.xml methodProtection.exclude]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-EnsureMethodProtectionFor20EJB',
 [['.*', 'customerEjb.jar,.*', 'methodProtection.uncheck'],
 ['.*', 'supplierEjb.jar,.*', 'methodProtection.exclude']]])
```

The last field in each entry of this task specifies the value of the protection. Valid protection values include: `methodProtection.uncheck`, `methodProtection.exclude`, or a list of security roles that are separated by commas.

filepermission

The filepermission option enables you to set the appropriate file permissions on application files that are located in the installation directory. File permissions that you specify at the application level must be a subset of the node level file permission that defines the most lenient file permission that can be specified. Otherwise, node level permission values are used to set file permissions in the installation destination. The file name pattern is a regular expression. The default value is the following:

```
.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755
```

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-appname MyApp -cell myCell -node myNode -server server1
 -filepermission .*\\.jsp=777#.*\\.xml=755}
```

Using Jython:

```
AdminApp.install("app_server_root/installableApps/DynaCacheEsi.ear",
 ["-appname", "MyApp", "-cell", "myCell", "-node", "myNode", "-server", "server1",
 "-filepermission", ".*\\.jsp=777#.*\\.xml=755"])
```

installdir (deprecated)

Note: The installdir option is deprecated. This option is replaced by the `installed.ear.destination` option.

installed.ear.destination

The installed.ear.destination option specifies the directory to place application binaries.

JSPCompileOptions

The JSPCompileOptions option specifies Java ServerPages (JSP) compilation options for web modules. This option is only valid if you use the preCompileJSPs option also. The current contents of this option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- JSP Class Path: <jsp class path>
- Use Full Package Names: Yes | No
- JDK Source Level: xx
- Disable JSP Runtime Compilation: Yes | No

For Use Full Package Names and Disable JSP Runtime Compilation, specify a `Yes` or `No` value in the language specific for the locale. The product supports the values `AppDeploymentOption.Yes` and `AppDeploymentOption.No` instead of `Yes` and `No`. However, it is recommended that you specify `Yes` or `No` in the language for your locale.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -preCompileJSPs -JSPCompileOptions {{"IVT Application"
 ivt_app.war,WEB-INF/web.xml jspcp Yes 15 No}}}
```

Using Jacl with pattern matching:

```
$AdminApp install ivtApp.ear {-appname MyApp -preCompileJSPs -JSPCompileOptions
 {{.* .* jspcp Yes 15 No}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp', '-preCompileJSPs', '-JSPCompileOptions',
 [["IVT Application", 'ivt_app.war,WEB-INF/web.xml', 'jspcp',
 'Yes', 15, 'No']]])
```

Using Jython with pattern matching:

```
AdminApp.install('ivtApp.ear', ['-appname', 'MyApp', '-preCompileJSPs',
 '-JSPCompileOptions', [['.*', '.*', 'jspcp', 'Yes', 15, 'No']]])
```

JSPReloadForWebMod

The JSPReloadForWebMod option edits the JSP reload attributes for the web module. You can specify the reload attributes of the servlet and JSP for each module. The current contents of the option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- JSP enable class reloading: <Yes | No>
- JSP reload interval in seconds: <jsp reload internal number>

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application.

For JSP enable class reloading, specify a `Yes` or `No` value in the language specific for the locale. The product supports the values `AppDeploymentOption.Yes` and `AppDeploymentOption.No` instead of `Yes` and `No`. However, it is recommended that you specify `Yes` or `No` in the language for your locale.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -JSPReloadForWebMod {{"IVT Application"
 ivt_app.war,WEB-INF/web.xml Yes 5}}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -JSPReloadForWebMod {{.* .* Yes 5}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp', '-JSPReloadForWebMod [["IVT Application"
 ivt_app.war,WEB-INF/web.xml Yes 5 ]]']])
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp', '-JSPReloadForWebMod', [['.*', '.*', 'Yes', '5']]])
```

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

MapEJBRefToEJB

The MapEJBRefToEJB option maps enterprise Java references to enterprise beans. You must map each enterprise bean reference defined in your application to an enterprise bean. Use this option to provide missing data or update to a task.

If the EJB reference is from an EJB 3.x, Web 2.4, or Web 2.5 module, the JNDI name is optional. If you specify the useAutoLink option, the JNDI name is optional. The run time provides a container default. An EJB 3.0 or later module cannot contain container-managed or bean-managed persistence entity beans. Installation fails when a container-managed or bean-managed persistence entity bean is packaged in an EJB 3.x module of a Java EE application. You can only package container-managed or bean-managed persistence beans in an EJB 2.1 or earlier module.

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application. You only need to provide data for rows or entries that are missing information, or those where you want to update the existing data.

Batch mode example usage

Using Jacl:

```
$AdminApp install techapp.ear {-MapEJBRefToEJB {{"JAASLogin" "" JAASLoginWeb.war,
WEB-INF/web.xml WSsamples/JAASLogin jaasloginejb.SampleBean TechApp/JAASLogin}}
-MapWebModToVH {{"Web Application" Tech.war,WEB-INF/web.xml default_host}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapEJBRefToEJB {{.* .* .* .* .* MyBean}}}
```

Using Jython:

```
AdminApp.install('techapp.ear','[
-MapEJBRefToEJB [["JAASLogin" "" JAASLoginWeb.war,WEB-INF/web.xml WSsamples/JAASLogin]]
-MapWebModToVH [["Web Application" tech.war,WEB-INF/web.xml default_host]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-MapEJBRefToEJB',
 [['.*', '.*', '.*', '.*', '.*', 'MyBean']]])
```

MapEnvEntryForApp

The MapEnvEntryForApp option edits the env-entry value of the application. You can use this option to edit the value of env-entry in the `application.xml` file.

The current contents of this option after running default bindings are the following:

- Name: xxx
- Type: xxx
- Description: xxx

  gotcha: If after running the default bindings, the value is a `null` value, you must change the value to `.*` before you run this task. The `.*` value is the wildcard value that lets any description be a match. If you leave the value as `null`, exceptions will occur when you run the MapEnvEntryForApp task.
- Value: <env-entry value>

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -MapEnvEntryForApp {{MyApp/appEnvEntry String null newEnvEntry}}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp2 -MapEnvEntryForApp {{.* .* .* newEnvEntry}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp3', '-MapEnvEntryForApp', [['MyApp/appEnvEntry', 'String', 'null', 'newEnvEntry']]])
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp4', '-MapEnvEntryForApp', [['.*', '.*', '.*', 'newEnvEntry']]])
```

If there is a new line character in the description, use the following syntax:

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp5 -MapEnvEntryForApp {{.* .* (?s).* newEnvEntry}}}
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp6', '-MapEnvEntryForApp',
 [['.*', '.*', '(?s).*', 'newEnvEntry']]])
```

MapEnvEntryForEJBMod

The MapEnvEntryForEJBMod option edits the env-entry value of the EJB module. You can use this option to edit the value of environment entries in the EJB module, ejb-jar.xml.

The current contents of this option after running default bindings are the following:

- EJB module: xxx
- URI: xxx
- EJB: xxx
- Name: xxx
- Type: xxx
- Description: xxx

  gotcha: If after running the default bindings, the value is a `null` value, you must change the value to `.*` before you run this task. The `.*` value is the wildcard value that lets any description be a match. If you leave the value as `null`, WASX7017E and WASX7111E exceptions will occur when you run the MapEnvEntryForEJBMod task.
- Value: <env-entry value>

If the EJB module uses EJB 3.0 or later, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -MapEnvEntryForEJBMod {{"IVT EJB Module" ivtEJB.jar,META-INF/ejb-jar.xml
  ivtEJBObject ejb/ivtEJBObject String null newValue}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.jar {-appname MyApp -MapEnvEntryForEJBMod {{.*  .*  .*
  .* .* .* newValue}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp', '-MapEnvEntryForEJBMod', [["IVT EJB Module",
 'ivtEJB.jar,META-INF/ejb-jar.xml', 'ivtEJBObject', 'ejb/ivtEJBObject',
 'String', 'null', 'newValue']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForEJBMod',
 [['.*', '.*', '.*', '.*', '.*', '.*', 'newValue']]])
```

If there is a new line character in the description, use the following syntax:

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -MapEnvEntryForEJBMod {{.* .* .*
 (?s).* newValue}}}
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForEJBMod',
 [['.*', '.*', '.*', '(?s).*', '*', '*' 'newValue']]])
```

MapEnvEntryForWebMod

The MapEnvEntryForWebMod option edits the env-entry value of the web module. You can use this option to edit the value of env-entry in the web.xml file.

The current contents of this option after running default bindings are the following:

- Web module: xxx
- URI: xxx
- Name: xxx
- Type: xxx
- Description: xxx

  gotcha: If, after running the default bindings, the value is a `null` value, you must change the value to `.*` before you run this task. The `.*` value is the wildcard value that lets any description be a match. If you leave the value as `null`, WASX7017E and WASX7111E exceptions will occur when you run the MapEnvEntryForWebMod task.
- Value: <env-entry value>

If the web module uses a Servlet 2.5, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -MapEnvEntryForWebMod {{"IVT Application" ivt_app.war,
 WEB-INF/web.xml ivt/ivtEJBObject String null newEnvEntry}}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp2 -MapEnvEntryForWebMod {{.* .* .* .* newEnvEntry}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp3', '-MapEnvEntryForWebMod', [["IVT Application",
 'ivt_app.war,WEB-INF/web.xml', 'ivt/ivtEJBObject', 'String', 'null', 'newEnvEntry']]])
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp4', '-MapEnvEntryForWebMod', [['.*', '.*', '.*', '.*', 'newEnvEntry']]])
```

If there is a new line character in the description, use the following syntax:

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp5 -MapEnvEntryForWebMod {{.* .* .* (?s).* newEnvEntry}}}
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp6', '-MapEnvEntryForWebMod',
 [['.*', '.*', '.*', '(?s).*', 'newEnvEntry']]])
```

MapInitParamForServlet

The MapInitParamForServlet option edits the initial parameter of a web module. You can use this option to edit the initial parameter of a servlet in the `web.xml` file. The current contents of this option after running the default bindings are the following:

- Web module: xxx
- URI: xxx
- Servlet: xxx
- Name: xxx
- Description: null
- Value: <initial parameter value>

If the web module uses a Servlet 2.5 or later, the contents of this option are populated only from the XML deployment descriptor. You cannot get deployment information from annotations with this option.

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/ivtApp.ear
 {-appname MyApp -MapInitParamForServlet {{"IVT Application"
 ivt_app.war,WEB-INF/web.xml ivtservlet pName1 null MyInitParamValue}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-appname MyApp -MapInitParamForServlet
 {{.*  .*  .*  .*  .*  MyInitParamValue}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/ivtApp.ear',
 ['-appname', 'MyApp', '-MapInitParamForServlet', [["IVT Application",
 'ivt_app.war,WEB-INF/web.xml', 'ivtservlet', 'pName1', 'null', 'MyInitParamValue']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapInitParamForServlet',
 [['.*', '.*', '.*', '.*', '.*', 'MyInitParamValue']]])
```

MapJaspiProvider

The MapJaspiProvider option specifies the web application or web modules where you want to override the Java Authentication SPI (JASPI) settings from the global or domain security configuration. The contents of this option are the following:

- Module: xxx
- URI: xxx
- Use JASPI: Yes | No | Inherit

- JASPI provider name: xxx

For Use JASPI, specify a `Yes`, `No`, or `Inherit` value in the language specific for the locale.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are missing information or that require an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapJaspiProvider {{myapp META-INF/application.xml Yes Provider1} }}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-appname myapp
 -MapJaspiProvider [[myapp META-INF/application.xml Yes Provider1] ]]')
```

MapMessageDestinationRefToEJB

The MapMessageDestinationRefToEJB option maps message destination references to Java Naming and Directory Interface (JNDI) names of administrative objects from the installed resource adapters. You must map each message destination reference that is defined in your application to an administrative object. Use this option to provide missing data or to update a task.

The current contents of the option after running default bindings include:

- Module: ejb-jar-ic.jar
- EJB: MessageBean
- URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml
- Message destination object: jms/GSShippingQueue
- Target Resource JNDI Name: [jms/GSShippingQueue]:

If the message destination reference is from an EJB 3.0 or later module, then the JNDI name is optional and the run time provides a container default.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install BankFull.ear {-MapMessageDestinationRefToEJB {{BankCMRQLEJB
 Sender BankCMRQLEJB.jar,META-INF/ejb-jar.xml BankJSQueue BankJSQueue} }}
```

Using Jacl with pattern matching:

```
$AdminApp install $earfile {-MapMessageDestinationRefToEJB {{.*  .*  .*  MyConnection
 jndi2} {.* .*  .*  PhysicalTopic jndi3} {.*  .*  .*  jms/ABC jndi4}}}
```

Using Jython:

```
AdminApp.install('BankFull.ear', '[ -MapMessageDestinationRefToEJB [[ BankCMRQLEJB
 Sender BankCMRQLEJB.jar,META-INF/ejb-jar.xml BankJSQueue BankJSQueue ]]]' )
```

Using Jython with pattern matching:

```
AdminApp.install(ear1, ['-MapMessageDestinationRefToEJB', [['.*', '.*', '.*', 'MyConnection',
 'jndi2'], ['.*', '.*', '.*', 'PhysicalTopic', 'jndi3'], ['.*', '.*', '.*', 'jms/ABC', 'jndi4']]])
```

MapModulesToServers

The MapModulesToServers option specifies the application server where you want to install modules that are contained in your application. You can install modules on the same server, or disperse them among several servers. Use this option to provide missing data or to update to a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

The following example installs `Increment.jar` on one server:

Using Jacl:

```
$AdminApp install Increment.jar {-MapModulesToServers
 {{"Increment Enterprise Java Bean" Increment.jar,META-INF/ejb-jar.xml
 WebSphere:cell=mycell,node=mynode,server=server1}}}
```

Using Jython:

```
AdminApp.install('Increment.jar', '[-MapModulesToServers
 [["Increment Enterprise Java Bean" Increment.jar,META-INF/ejb-jar.xml
 WebSphere:cell=mycell,node=mynode,server=server1] ]]')
```

The following example removes `server1` from the application that is installed:

Using Jacl:

```
$AdminApp edit myapp {-MapModulesToServers
 {{.* .* -WebSphere:cell=mycell,node=mynode,server=server1}}
 -update -update.ignore.old}
```

Using Jython:

```
AdminApp.edit('myapp', ['-MapModulesToServers',
 [['.*', '.*', '-WebSphere:cell=mycell,node=mynode,server=server1']]])
```

MapResEnvRefToRes

The MapResEnvRefToRes option maps resource environment references to resources. You must map each resource environment reference that is defined in your application to a resource. Use this option to provide missing data or to update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/sibwsjmsepl.ear {
 -BindJndiForEJBMessageBinding {{ SOAPJMSEndpoint SOAPJMSEndpoint
   SOAPJMSEndpoint.jar,META-INF/ejb-jar.xml lp1 "" "" "" }}
 -MapResEnvRefToRes {{SOAPJMSEndpoint SOAPJMSEndpoint
   SOAPJMSEndpoint.jar,META-INF/ejb-jar.xml jms/WebServicesReplyQCF
   javax.jms.QueueConnectionFactory jndi1}}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/sibwsjmsepl.ear {
 -MapResEnvRefToRes {{.* .* .* .* .* jndi1}}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/sibwsjmsepl.ear', '[
 -BindJndiForEJBMessageBinding [[ SOAPJMSEndpoint SOAPJMSEndpoint
   SOAPJMSEndpoint.jar,META-INF/ejb-jar.xml lp1 "" "" "" ]]
 -MapResEnvRefToRes [[ SOAPJMSEndpoint SOAPJMSEndpoint
   SOAPJMSEndpoint.jar,META-INF/ejb-jar.xml jms/WebServicesReplyQCF
   javax.jms.QueueConnectionFactory jndi1 ]]]' )
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/sibwsjmsepl.ear',
 ['-MapResEnvRefToRes', [['.*', '.*', '.*', '.*', '.*', 'jndi1']]])
```

MapResRefToEJB

The MapResRefToEJB option maps resource references to resources. You must map each resource reference that is defined in your application to a resource. Use this option to provide missing data or to update a task.

The parameters for MapResRefToEJB include:

- EJBModule: Ejb1
- EJB: MailEJBObject
- URI: deplmtest.jar,META-INF/ejb-jar.xml
- Reference binding: jms/MyConnectionFactory
- Resource type: javax.jms.ConnectionFactory
- JNDI name: [jms/MyConnectionFactory]:
- Login Configuration Name: [null]: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Properties: []: Use this option to create a custom login configuration. The client can use JAAS to create a login design.
- Extended Data source properties: []: Use this option so that a data source that uses heterogeneous pooling can connect to a DB2 database. The pattern for the property is property1=value1+property2=value 2.

The DefaultPrincipalMapping login configuration is used by Java 2 Connectors (J2C) to map users to principals that are defined in the J2C authentication data entries. If the login.config.name is set to DefaultPrincipalMapping, a property is created with the name com.ibm.mapping.authDataAlias. The value of the property is set by the auth.props. If the login.config name is not set to DefaultPrincipalMapping, the auth.props can specify multiple properties. The string format is `websphere:name=<name1>,value=<value1>,description=<desc1>`. Specify multiple properties using the plus sign (+).

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapResRefToEJB
 {{deplmtest.jar MailEJBObject deplmtest.jar,META-INF/ejb-jar.xml mail/MailSession9
 javax.mail.Session jndi1 login1 authProps1 "clientApplicationInformation=new application"}
 {"JavaMail Sample WebApp" "" mtcomps.war,WEB-INF/web.xml mail/MailSession9
 javax.mail.Session jndi2 login2 authProps2 ""}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-MapResRefToEJB
 {{deplmtest.jar .*  .*  .*  .*  jndi1 login1 authProps1
 "clientApplicationInformation=new application"}
 {"JavaMail Sample WebApp" .*  .*  .*  .*  jndi2 login2 authProps2 ""}}}
```

Note: If you use multiple values for extended data source properties, you must assign the values to a variable and use that variable for the property value. For example:

```
set s \"value1,value2\"
```

In the command substitution, reference the variable as shown in the following example:

```
"clientApplicationInformation=value1,value2" with "clientapplication=$s"
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-MapResRefToEJB',
 [['deplmtest.jar', 'MailEJBObject', 'deplmtest.jar,META-INF/ejb-jar.xml
 mail/MailSession9', 'javax.mail.Session', 'jndi1', 'login1', 'authProps1',
 'clientApplicationInformation=new application+clientWorkstation=9.10.117.65'],
 ["JavaMail Sample WebApp", "", 'mtcomps.war,WEB-INF/web.xml', 'mail/MailSession9',
 'javax.mail.Session', 'jndi2', 'login2', 'authProps2', '']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-MapResRefToEJB',
 [['deplmtest.jar', '.*', '.*', '.*', '.*', 'jndi1', 'login1', 'authProps1',
 'clientApplicationInformation=new application+clientWorkstation=9.10.117.65'],
 ["JavaMail Sample WebApp", '', '.*', '.*', '.*', 'jndi2', 'login2',
 'authProps2', '']]])
```

Note: If you use multiple values for extended data source properties, surround the two values with double quote characters. For example:

```
'property1="value1,value2"+property2="value3,value4"'
```

MapRolesToUsers

The MapRolesToUsers option maps users to roles. You must map each role that is defined in the application or module to a user or group from the domain user registry. You can specify multiple users or groups for a single role by separating them with a pipe (|). Use this option to provide missing data or to update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapRolesToUsers {{"All Role" No Yes "" ""}
 {"Every Role" Yes No "" ""} {DenyAllRole No No user1 group1}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-MapRolesToUsers [["All Role" No Yes "" ""]
 ["Every Role" Yes No "" ""] [DenyAllRole No No user1 group1]]]')
```

where `{{"All Role" No Yes "" ""}` corresponds to the following:

- "All Role": Represents the role name
- No: Indicates to allow access to everyone (yes/no)
- Yes: Indicates to allow access to all authenticated users (yes/no)
- "": Indicates the mapped users

- "": Indicates the mapped groups

MapRunAsRolesToUsers

The MapRunAsRolesToUsers option maps RunAs Roles to users. The enterprise beans that you install contain predefined RunAs roles. Enterprise beans that need to run as a particular role for recognition while interacting with another enterprise bean use RunAs roles. Use this option to provide missing data or to update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MapRunAsRolesToUsers {{UserRole user1
 password1} {AdminRole administrator administrator}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', '[-MapRunAsRolesToUsers [[UserRole user1
 password1][AdminRole administrator administrator]]]')
```

MapSharedLibForMod

The MapSharedLibForMod option assigns shared libraries to application or every module. You can associate multiple shared libraries to applications and modules. The current contents of this option after running default bindings are the following:

- Module: xxx
- URI: META-INF/application.xml
- Shared libraries: <share libraries>

Use the taskInfo command of the AdminApp object to obtain information about the data needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-appname DynaCacheEsi -MapSharedLibForMod {{ DynaCacheEsi META-INF/application.xml
 sharedlib1 }{ DynaCacheEsi DynaCacheEsi.war,WEB-INF/web.xml sharedlib2 }}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-appname DynaCacheEsi -MapSharedLibForMod {{ .* .* sharedlib1 }
 { .* .* sharedlib2 }}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/DynaCacheEsi.ear',
 '[ -MapSharedLibForMod [[ DynaCacheEsi META-INF/application.xml sharedlib1 ]
 [ DynaCacheEsi DynaCacheEsi.war,WEB-INF/web.xml sharedlib2 ]]]' )
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/DynaCacheEsi.ear',
 '[ -MapSharedLibForMod [[ .* .* sharedlib1 ][ .* .* sharedlib2 ]]]' )
```

MapWebModToVH

The MapWebModToVH option selects virtual hosts for web modules. Specify the virtual host where you want to install the web modules that are contained in your application. You can install web modules on the same virtual host, or disperse them among several hosts. Use this option to provide missing data or to update a task.

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application. You need to provide data for rows or entries that are either missing information, or requiring an update.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-appname DynaCacheEsi
 -MapWebModToVH {{ DynaCacheEsi DynaCacheEsi.war,WEB-INF/web.xml default_host }}}
```

Using Jacl with pattern matching:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-MapWebModToVH {{ .* .* default_host }}}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/DynaCacheEsi.ear',
 '[-MapWebModToVH [[ DynaCacheEsi DynaCacheEsi.war,WEB-INF/web.xml default_host ]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('app_server_root/installableApps/DynaCacheEsi.ear',
 ['-MapWebModToVH', [['.*', '.*', 'default_host']]])
```

MetadataCompleteForModules

The MetadataCompleteForModules option enables each Java EE 5 or later module to write out the complete deployment descriptor, including deployment information from annotations. Then the system marks the deployment descriptor for the module as complete.

Note: If your Java EE 5 or later application uses annotations and a shared library, do not use this option. When your application uses annotations and a shared library, setting the `metadata-complete` attribute to `true` causes the product to incorrectly represent an `@EJB` annotation in the deployment descriptor as `<ejb-ref>` rather than `<ejb-local-ref>`. For web modules, setting the `metadata-complete` attribute to `true` might cause InjectionException errors. If you must set the `metadata-complete` attribute to `true`, avoid errors by not using a shared library, by placing the shared library in either the `classes` or `lib` directory of the application server, or by fully specifying the metadata in the deployment descriptors.

The current contents of this option after running default bindings are the following:

- Module: EJBDD_1.jar
- URI: EJBDD_1.jar,META-INF/ejb-jar.xml
- Lock deployment descriptor: [false]:
- Module: EJBNoDD_2.jar
- URI: EJBNoDD_2.jar,META-INF/ejb-jar.xml
- Lock deployment descriptor: [false]:

Use the taskInfo command of the AdminApp object to obtain information about the data that is needed for your application.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-MetadataCompleteForModules
 {{EJBDD_1.jar EJBDD_1.jar,META-INF/ejb-jar.xml false}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear
 {-MetadataCompleteForModules {{.* EJBDD_1.jar,.* false}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-MetadataCompleteForModules',
 [['EJBDD_1,jar',  'EJBDD_1.jar,META-INF/ejb-jar.xml', 'false']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear',
 ['-MetadataCompleteForModules', [['.*', 'EJBDD_1.jar,.*', 'false']]]
```

ModuleBuildID

The ModuleBuildID option displays the build identifier of a module in an application EAR file. You cannot modify this option because it is read-only. This option returns the build information for a module if you have specified the build identifer in the `MANIFEST.MF` of a module or application EAR file. The build information consists of the module name, module URI, and build identifier.

The current contents of the option after running default bindings resemble the following:

- Module: CacheClient
- URI: CacheClient.jar,META-INF/application-client.xml
- Build ID: alpha
- Module: MailConnector
- URI: Mail.rar,META-INF/ra.xml
- Build ID: abc
- Module: CacheWeb1
- URI: CacheWeb1.war,WEB-INF/web.xml
- Build ID: alpha
- Module: JspWeb1
- URI: JspWeb1.war,WEB-INF/web.xml
- Build ID: v1.0
- Module: Servlet
- URI: Servlet.war,WEB-INF/web.xml
- Build ID: 0.5

Batch mode example usage

Using Jython:

```
print AdminApp.taskInfo('/temp/Cache2.ear', 'ModuleBuildID')
```

noallowDispatchRemoteInclude

The noallowDispatchRemoteInclude option disables the enterprise application that dispatches `includes` to resources across web modules in different Java virtual machines in a managed node environment through the standard request dispatcher mechanism.

noallowPermInFilterPolicy

The noallowPermInFilterPolicy option specifies not to continue with the application deployment process when the application contains policy permissions that are in the filter policy. This option is the default setting and does not require a value.

noallowServiceRemoteInclude

The noallowServiceRemoteInclude option disables the enterprise application that services an include request from an enterprise application that has the allowDispatchRemoteInclude option set to `true`.

node

The node option specifies the node name to install or update an entire application or to update an application in order to add a new module. If you want to update an entire application, this option only applies if the application contains a new module that does not exist in the installed application.

Batch mode example usage

Using Jacl:

```
$AdminApp install "/myapp.ear" {-node nodeName}
```

Using Jython:

```
AdminApp.install('/myapp/myapp.ear', '[-node nodeName]')
```

nocreateMBeansForResources

The nocreateMBeansForResources option specifies that MBeans are not created for all resources, such as servlets, JavaServer Pages files, and enterprise beans, that are defined in an application when the application starts on a deployment target. This option is the default setting and it does not require a value.

nodeployejb

The nodeployejb option specifies not to run the EJBDeploy tool during installation. This option is the default setting and does not require a value.

nodeployws

The nodeployws option specifies not to deploy web services during installation. This option is the default setting and does not require a value.

nodistributeApp

The nodistributeApp option specifies that the application management component does not distribute application binaries. This option does not require a value. The default setting is the distributeApp option.

noenableClientModule

The noenableClientModule option specifies not to deploy client modules. This option does not require a value. This is the default. The default is not to deploy client modules. Use the enableClientModule option to deploy client modules.

noreloadEnabled

The noreloadEnabled option disables class reloading. This option is the default setting and does not require a value. To specify that the file system of the application be scanned for updated files so that

changes reload dynamically, use the reloadEnabled option.

nopreCompileJSPs

The nopreCompileJSPs option specifies not to precompile JavaServer Pages files. This option is the default setting and does not require a value.

noprocessEmbeddedConfig

The noprocessEmbeddedConfig option specifies that the system should ignore the embedded configuration data that is included in the application. This option does not require a value. If the application enterprise archive (EAR) file does not contain embedded configuration data, the noprocessEmbeddedConfig option is the default setting. Otherwise, the default setting is the processEmbeddedConfig option.

nouseAutoLink

The nouseAutoLink option specifies not to use the useAutoLink option, and not to automatically resolve Enterprise JavaBeans (EJB) references from EJB module versions prior to EJB 3.0 and from web module versions that are prior to Version 2.4.

nouseMetaDataFromBinary

The nouseMetaDataFromBinary option specifies that the metadata that is used at run time, for example, deployment descriptors, bindings, extensions, and so on, come from the configuration repository. This option is the default setting and does not require a value. Use this option to indicate that the metadata that is used at run time comes from the enterprise archive (EAR) file.

nousedefaultbindings

The nousedefaultbindings option specifies not to use default bindings for installation. This option is the default setting and does not require a value.

novalidateSchema

The novalidateSchema option specifies not to validate the deployment descriptors against published Java EE deployment descriptor schemas. This option does not require a value. This is the default. The default is not to validate deployment descriptors.

operation

The operation option specifies the operation that you want to perform. This option only applies to the update or updateInteractive commands. The valid values include:

- `add` - Adds new content.
- `addupdate` - Adds or updates content based on the existence of content in the application.
- `delete` - Deletes content.
- `update` - Updates existing content.

The operation option is required if the content type is file or modulefile. If the value of the content type is `app`, the value of the operation option must be `update`. The contenturi option is ignored when the content type is `app`.

Batch mode example usage

The following examples show how to use the options for the update command to update an entire deployed enterprise application:

Using Jacl:

```
$AdminApp update "IVT Application" app {-operation update
 -contents app_server_root/installableApps/ivtApp.ear}
```

Using Jython string:

```
AdminApp.update('IVT Application', 'app', '[ -operation update
 -contents app_server_root/installableApps/ivtApp.ear ]' )
```

The following examples show how to use the options for the update command to update a single file in a deployed application:

Using Jacl:

```
$AdminApp update app1 file {-operation update
 -contents /apps/app1/my.xml -contenturi app1.jar/my.xml}
```

Using Jython string:

```
AdminApp.update('app1', 'file', '[-operation update
 -contents /apps/app1/my.xml -contenturi app1.jar/my.xml]')
```

Using Jython list:

```
AdminApp.update('app1', 'file', ['-operation',  'update', '-contents',
 '/apps/app1/my.xml', '-contenturi', app1.jar/my.xml'])
```

where `AdminApp` is the scripting object, `update` is the command, `app1` is the name of the application you want to update, `file` is the content type, `operation` is an option of the update command, `update` is the value of the `operation` option, `contents` is an option of the update command, */apps/app1/my.xml* is the value of the `contents` option, `contenturi` is an option of the update command, *app1.jar/my.xml* is the value of the `contenturi` option.

processEmbeddedConfig

The processEmbeddedConfig option processes the embedded configuration data that is included in the application. This option does not require a value. If the application enterprise archive (EAR) file contains embedded configuration data, this option is the default setting. If not, the default setting is the nonprocessEmbeddedConfig option.

This setting affects installation of enhanced EAR files. An enhanced EAR file results when you export an installed application. Enhanced EAR files have an embedded configuration that consists of files such as `resource.xml` and `variables.xml`. This option loads an embedded configuration to the application scope from the EAR file.

If you exported the application from a cell other than the current cell and did not specify the `$(CELL)` variable for the installed.ear.destination option when first installing the application, use the nonprocessEmbeddedConfig option to expand the enhanced EAR file in the *profile_root*`/installedApps/`*current_cell_name* directory. Otherwise, use this processEmbeddedConfig option to expand the enhanced EAR file in the *profile_root*`/installedApps/`*original_cell_name* directory, where *original_cell_name* is the cell on which the application was first installed. If you specified the `$(CELL)` variable for the installed.ear.destination option when you first installed the application, installation expands the enhanced EAR file in the *profile_root*`/installedApps/`*current_cell_name* directory.

preCompileJSPs

The preCompileJSPs option specifies to precompile the JavaServer Pages files. This option does not require a value. The default value is nopreCompileJSPs. If you want to precompile JavaServer Pages files, specify it as a part of installation. The default is not to precompile JavaServer Pages files. The preCompileJSPs option is ignored during deployment and JavaServer Pages files are not precompiled. The flag is set automatically using assembly tools.

reloadEnabled

The reloadEnabled option specifies that the file system of the application will be scanned for updated files so that changes reload dynamically. If this option is enabled and if application classes are changed, then the application is stopped and restarted to reload updated classes. This option is not the default setting and does not require a value. The default setting is the noreloadEnabled option.

reloadInterval

The reloadInterval option specifies the time period in seconds that the file system of the application will be scanned for updated files. Valid range is greater than zero. The default is three seconds.

SharedLibRelationship

The SharedLibRelationship option assigns assets or composition unit IDs as shared libraries for each Java EE module.

The current contents of the option after running default bindings include:

- Module: EJB3BNDBean.jar
- URI: EJB3BNDBean.jar,META-INF/ejb-jar.xml
- Relationship IDs: specify asset or composition unit IDs, such as `[WebSphere:cuname=sharedLibCU1,cuedition=1.0]` or `WebSphere:assetname=sharedLibAsset1.jar`
- Composition Unit names: optionally specify composition unit names for asset relationship IDs. The system uses the same name as the asset if you do not specify a composition unit name. []
- Match target: [Yes]:

You can specify assets and composition unit IDs in the relationship, as the following guidelines explain:

- If you specify an asset, the system creates a composition unit with that asset in the same business level application where the Java EE application belongs.
- If you specify a value for the composition unit names, then the system position matches each name with the corresponding relationship IDs values.
- If the relationship ID is a composition unit ID, then the system ignores the corresponding composition unit name.
- If the relationship ID is an asset ID, then the system creates the composition unit using the corresponding composition unit name.

To specify more than one asset or composition unit ID, separate each value with the plus sign character (+).

When using the edit command for the Java EE application, you can override the relationship with a new set of composition unit relationship IDs, or you can add or remove existing composition unit relationships. You cannot specify an asset relationship when using the edit command. Specify the first character of the composition unit ID as the plus sign character (+) to add to the relationship, or specify the number sign character (#) to remove the composition unit ID from existing relationships. For example, the

+cuname=cu2.zip composition unit syntax adds the cu2 composition unit to the relationship. The #cuname=cu1.zip+cuname=cu2.zip composition unit syntax removes the cu1 and cu2 composition units from the relationship.

Batch mode example usage

Using Jacl:

```
$AdminApp install myapp.ear {-SharedLibRelationship {{EJB3BNDBean.jar
 EJB3BNDBean.jar,META-INF/ejb-jar.xml WebSphere:cuname=sharedLibCU1 "" Yes}}}
```

Using Jacl with pattern matching:

```
$AdminApp install myapp.ear {-SharedLibRelationship {{.*
 EJB3BNDBean.jar,.* WebSphere:cuname=sharedLibCU1 "" Yes}}}
```

Using Jython:

```
AdminApp.install('myapp.ear', ['-SharedLibRelationship',
 [['EJB3BNDBean,jar', 'EJB3BNDBean.jar,META-INF/ejb-jar.xml',
 'WebSphere:cuname=sharedLibCU1,' '', 'Yes']]])
```

Using Jython with pattern matching:

```
AdminApp.install('myapp.ear', ['-SharedLibRelationship', [['.*',
 'EJB3BNDBean.jar,.*', 'WebSphere:cuname=sharedLibCU1', '', 'Yes']]]
```

server

The server option specifies the name of the server on which you want to perform one of the following actions:

- Install a new application.
- Replace an existing application with an updated version of that application. In this situation, the server option is meaningful only if the updated version of the application contains a new module that does not exist in the already installed version of the application.
- Add a new module to an existing application.

Batch mode example usage

Using Jacl:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-server server1}
```

Using Jython:

```
AdminApp.install('app_server_root/installableApps/DynaCacheEsi.ear',
 '[-server server1]')
```

target

The target option specifies the target for the installation functions of the AdminApp object. The following is an example of a target option: WebSphere:cell=mycell,node=mynode,server=myserver

You can specify multiple targets by delimiting them with a plus (+) sign. By default, the targets that you specify when you install or edit an application replace the existing target definitions in the application. You can use a leading plus (+) sign to add targets or a negative (-) sign to remove targets without having to specify the targets that are not changed.

Batch mode example usage

The following example specifies server1 as the target server for the application installation:

Using Jacl:

```
$AdminApp install app_server_root/installableApps/DynaCacheEsi.ear
 {-appname MyApp -target WebSphere:cell=myCell,node=myNode,server=server1}
```

Using Jython:

```
AdminApp.install("app_server_root/installableApps/DynaCacheEsi.ear",
 ["-appname", "MyApp", "-target", "WebSphere:cell=myCell,node=myNode,server=server1"])
```

update

The update option updates the installed application with a new version of the enterprise archive (EAR) file. This option does not require a value.

The application to update, which is specified by the appname option, must already be installed in the WebSphere Application Server configuration. The update action merges bindings from the new version with the bindings from the old version, uninstalls the old version, and installs the new version. The binding information from new version of the EAR file or module is preferred over the corresponding one from the old version. If any element of binding is missing in the new version, the corresponding element from the old version is used.

update.ignore.new

The update.ignore.new option specifies that during the update action, binding information from the old version of the application or module is preferred over the corresponding binding information from the new version. If any element of the binding does not exist in the old version, the element from the new version is used. That is, bindings from the new version of the application or module are ignored if a binding exists in the old version. Otherwise, the new bindings are honored and not ignored. This option does not require a value.

This option applies only if you specify one of the following items:

- The update option for the install command.
- The modulefile or app as the content type for the update command.

update.ignore.old

The update.ignore.old option specifies that during the update action, the binding information from the new version of the application or module is preferred over the corresponding binding information from the old version. The bindings from the old version of the application or module are ignored. This option does not require a value. This option applies only if you specify one of the following items:

- The update option for the install command.
- The modulefile or app as the content type for the update command.

useAutoLink

Use the useAutoLink option to automatically resolve Enterprise JavaBeans (EJB) references from EJB module versions prior to EJB 3.0, and from web module versions that are prior to Version 2.4. If you enable the useAutoLink option, you can optionally specify the JNDI name for MapEJBRefToEJB option. Each module in the application must share one common target to enable autolink support.

Batch mode example usage

Using Jacl:

```
$AdminApp install /myapp.ear {-useAutoLink}
```

Using Jython:

```
AdminApp.install('myapp.ear','[-useAutoLink]')
```

usedefaultbindings

The usedefaultbindings option specifies to use default bindings for installation. This option does not require a value. The default setting is nousedefaultbindings.

To use the existing listener port instead of using or creating a new activation specification, determine whether the EJB JAR version is lower than 2.1. The system automatically creates and uses an activation specification when you specify the -usedefaultbindings option to deploy an application. If an activation specification exists, the system ignores the listener port, and instead uses the activation specification. To deploy an application with an EJB JAR version greater than or equal to 2.1 using the defined listener ports instead of a new activation specification, set the com.ibm.websphere.management.application.dfltbndng.mdb.preferexisting system property to `true` in the `wsadmin.properties` file in the properties directory of the profile of interest.

useMetaDataFromBinary

The useMetaDataFromBinary option specifies that the metadata that is used at run time, for example, deployment descriptors, bindings, extensions, and so on, come from the EAR file. This option does not require a value. The default value is `nouseMetaDataFromBinary`, which means that the metadata that is used at run time comes from the configuration repository.

validateinstall

The validateinstall option specifies the level of application installation validation. Valid option values include:

- off - Specifies no application deployment validation. This value is the default.
- warn - Performs application deployment validation and continues with the application deployment process even when reported warnings or error messages exist.
- fail - Performs application deployment validation and does not to continue with the application deployment process when reported warnings or error messages exist.

validateSchema

Note: The validateSchema option specifies to validate the deployment descriptors against published Java EE deployment descriptor schemas. When this application deployment option is selected, the product analyzes each deployment descriptor to determine the Java EE specification version for the deployment descriptor, selects the appropriate schema, and then checks the deployment descriptor against the Java EE deployment descriptor schema. Validation errors result in error messages.

This option does not require a value. The default value is novalidateSchema. By default, the validateSchema option is ignored during deployment and the product does not validate deployment descriptors.

verbose

The verbose option causes additional messages to display during installation. This option does not require a value.

WebServicesClientBindDeployedWSDL

The WebServicesClientBindDeployedWSDL option identifies the client Web service that you are modifying. The scoping fields include: Module, EJB, and web service. The single mutable value for this task is the deployed WSDL file name. It indicates the Web Services Description Language (WSDL) the client uses.

The Module field identifies the enterprise or web application within the application. If the module is an enterprise bean , the EJB field identifies a particular enterprise bean within the module. The web service field identifies the web service within the enterprise bean or the web application module. This identifier corresponds to the wsdl:service attribute in the WSDL file, prepended with `service/`, for example, `service/WSLoggerService2`.

The deployed WSDL attribute names a WSDL file relative to the client module. An example of a deployed WSDL for a web application is the following: `WEB-INF/wsdl/WSLoggerService`.

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindDeployedWSDL {{AddressBookW2JE.jar
 AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl}}}
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindDeployedWSDL {{.*  .*  .*
 META-INF/wsdl/DeployedWsdl1.wsdl}}}
```

To install the `WebServicesSamples.ear` sample, which is in `installableApps` directory of SCA samples, you must specify the BindJndiForEJBNonMessageBinding and MapWebModToVH options as well as the WebServicesClientBindDeployedWSDL option:

```
$AdminApp install WebServicesSamples.ear {
-WebServicesClientBindDeployedWSDL {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2
 META-INF/wsdl/DeployedWsdl1.wsdl}}
-BindJndiForEJBNonMessageBinding {{ "Stock Quote Sample EJB"
 com_ibm_websphere_samples_webservices_stock_StockQuote StockQuote.jar,META-INF/ejb-jar.xml
jndi1 "" "" } { "Address Book Sample EJB" com_ibm_websphere_samples_webservices_addr_AddressBookW2JE
 AddressBookW2JE.jar,META-INF/ejb-jar.xml jndi2 "" "" }{ "Address Book Sample EJB"
 com_ibm_websphere_samples_webservices_addr_AddressBookJ2WE AddressBookJ2WE.jar,META-INF/ejb-jar.xml
 jndi3 "" "" }}
-MapWebModToVH {{ "module_name" web_services.war,WEB-INF/web.xml default_host }
 { "AddressBook Bottom Up Java Bean" AddressBookJ2WB.war,WEB-INF/web.xml default_host }
 { "AddressBook Top Down Java Bean" AddressBookW2JB.war,WEB-INF/web.xml default_host }} }
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindDeployedWSDL
 [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl]]]')
```

To install the `WebServicesSamples.ear` sample, you must specify the BindJndiForEJBNonMessageBinding and MapWebModToVH options as well as the WebServicesClientBindDeployedWSDL option:

```
$AdminApp install WebServicesSamples.ear', '[
-WebServicesClientBindDeployedWSDL [[ AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2
 META-INF/wsdl/DeployedWsdl1.wsdl ]]
-BindJndiForEJBNonMessageBinding [[ "Stock Quote Sample EJB"
 com_ibm_websphere_samples_webservices_stock_StockQuote StockQuote.jar,META-INF/ejb-jar.xml
jndi1 "" "" ] [ "Address Book Sample EJB" com_ibm_websphere_samples_webservices_addr_AddressBookW2JE
 AddressBookW2JE.jar,META-INF/ejb-jar.xml jndi2 "" "" ][ "Address Book Sample EJB"
 com_ibm_websphere_samples_webservices_addr_AddressBookJ2WE AddressBookJ2WE.jar,META-INF/ejb-jar.xml
 jndi3 "" "" ]]
-MapWebModToVH [[ "module_name" web_services.war,WEB-INF/web.xml default_host ]
 [ "AddressBook Bottom Up Java Bean" AddressBookJ2WB.war,WEB-INF/web.xml default_host ]
 [ "AddressBook Top Down Java Bean" AddressBookW2JB.war,WEB-INF/web.xml default_host ]]]' )
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindDeployedWSDL',
['.*', '.*', '.*',  'META-INF/wsdl/DeployedWsdl1.wsdl']]])
```

WebServicesClientBindPortInfo

The WebServicesClientBindPortInfo option identifies the port of a client web service that you are modifying. The scoping fields include: Module, EJB, web service and Port. The mutable values for this task include: `Sync Timeout`, `BasicAuth ID`, `BasicAuth Password`, `SSL Config`, and `Overridden Endpoint URI`. The basic authentication and Secure Sockets Layer (SSL) fields affect transport level security, not Web Services Security.

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPortInfo {{AddressBookW2JE.jar
 AddressBookW2JE service/WSLoggerService2 WSLoggerJMS 3000 newHTTP_ID newHTTP_pwd sslAliasConfig
 http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS}}}
```

To install the `WebServicesSamples.ear` sample, you must specify the BindJndiForEJBNonMessageBinding and MapWebModToVH options as well as the WebServicesClientBindPortInfo option:

```
$AdminApp install WebServicesSamples.ear {
-WebServicesClientBindPortInfo {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2
 WSLoggerJMS 3000 newHTTP_ID newHTTP_pwd sslAliasConfig
 http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS}}
-BindJndiForEJBNonMessageBinding {{ "Stock Quote Sample EJB"
 com_ibm_websphere_samples_webservices_stock_StockQuote StockQuote.jar,META-INF/ejb-jar.xml
 jndi1 "" "" }
 { "Address Book Sample EJB" com_ibm_websphere_samples_webservices_addr_AddressBookW2JE
 AddressBookW2JE.jar,META-INF/ejb-jar.xml jndi2 "" "" }{ "Address Book Sample
 EJB" com_ibm_websphere_samples_webservices_addr_AddressBookJ2WE AddressBookJ2WE.jar,META-INF/ejb-jar.xml
 jndi3 "" "" }}
-MapWebModToVH {{ "module_name" web_services.war,WEB-INF/web.xml default_host }
 { "AddressBook Bottom Up Java Bean" AddressBookJ2WB.war,WEB-INF/web.xml default_host }
 { "AddressBook Top Down Java Bean" AddressBookW2JB.war,WEB-INF/web.xml default_host }} }
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPortInfo {{.*  .*  .*  .*  3000
 newHTTP_ID newHTTP_pwd sslAliasConfig http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPortInfo
 [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS 3000 newHTTP_ID
 newHTTP_pwd sslAliasConfig http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS]]]')
```

To install the `WebServicesSamples.ear` sample, you must specify the BindJndiForEJBNonMessageBinding and MapWebModToVH options as well as the WebServicesClientBindPortInfo option:

```
AdminApp.install('WebServicesSamples.ear', '[
-WebServicesClientBindPortInfo [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS
 3000 newHTTP_ID newHTTP_pwd sslAliasConfig http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS ]]
-BindJndiForEJBNonMessageBinding [[ "Stock Quote Sample EJB"
 com_ibm_websphere_samples_webservices_stock_StockQuote StockQuote.jar,META-INF/ejb-jar.xml jndi1 "" "" ]
 [ "Address Book Sample EJB" com_ibm_websphere_samples_webservices_addr_AddressBookW2JE
 AddressBookW2JE.jar,META-INF/ejb-jar.xml jndi2 "" "" ][ "Address Book Sample EJB"
 com_ibm_websphere_samples_webservices_addr_AddressBookJ2WE AddressBookJ2WE.jar,META-INF/ejb-jar.xml
 jndi3 "" "" ]]
-MapWebModToVH [[ "module_name" web_services.war,WEB-INF/web.xml default_host ]
 [ "AddressBook Bottom Up Java Bean" AddressBookJ2WB.war,WEB-INF/web.xml default_host ]
 [ "AddressBook Top Down Java Bean" AddressBookW2JB.war,WEB-INF/web.xml default_host ]]]' )
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindPortInfo',
 [['.*', '.*', '.*', '.*', '3000', 'newHTTP_ID', 'newHTTP_pwd', 'sslAliasConfig',
 'http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS']]])
```

WebServicesClientBindPreferredPort

The WebServicesClientBindPreferredPort option associates a preferred port (implementation) with a port type (interface) for a client web service. The immutable values identify a port type of the client web service that you are modifying. The scoping fields include: Module, EJB, Web service and Port Type. The mutable value for this task is `Port`.

- Port Type - QName ("{namespace} localname") of a port type that is defined by a wsdl:portType attribute in the WSDL file that identifies an interface.
- Port - QName of a port defined by a wsdl:port attribute within a wsdl:service attribute in a WSDL file that identifies an implementation that has preference.

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort
 {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort}}}
```

To install the `WebServicesSamples.ear` sample, you must specify the BindJndiForEJBNonMessageBinding and MapWebModToVH options as well as the WebServicesClientBindPreferredPort option:

```
$AdminApp install WebServicesSamples.ear
{-WebServicesClientBindPreferredPort {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2
 WSLoggerJMS WSLoggerJMSPort}}
-BindJndiForEJBNonMessageBinding {{ "Stock Quote Sample EJB"
 com_ibm_websphere_samples_webservices_stock_StockQuote StockQuote.jar,META-INF/ejb-jar.xml jndi1 "" "" }
 { "Address Book Sample EJB" com_ibm_websphere_samples_webservices_addr_AddressBookW2JE
 AddressBookW2JE.jar,META-INF/ejb-jar.xml jndi2 "" "" }{ "Address Book Sample EJB"
 com_ibm_websphere_samples_webservices_addr_AddressBookJ2WE AddressBookJ2WE.jar,META-INF/ejb-jar.xml
 jndi3 "" "" }}
-MapWebModToVH {{ "module_name" web_services.war,WEB-INF/web.xml default_host }
 { "AddressBook Bottom Up Java Bean" AddressBookJ2WB.war,WEB-INF/web.xml default_host }
 { "AddressBook Top Down Java Bean" AddressBookW2JB.war,WEB-INF/web.xml default_host }} }
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort {{.* .* .* .*
 WSLoggerJMSPort}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPreferredPort
 [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort]]]')
```

To install the `WebServicesSamples.ear` sample, you must specify the BindJndiForEJBNonMessageBinding and MapWebModToVH options as well as the WebServicesClientBindPreferredPort option:

```
AdminApp.install('WebServicesSamples.ear',
'[ -WebServicesClientBindPreferredPort
 [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort ]]
-BindJndiForEJBNonMessageBinding [[ "Stock Quote Sample EJB"
 com_ibm_websphere_samples_webservices_stock_StockQuote StockQuote.jar,META-INF/ejb-jar.xml jndi1 "" "" ]
 [ "Address Book Sample EJB" com_ibm_websphere_samples_webservices_addr_AddressBookW2JE
 AddressBookW2JE.jar,META-INF/ejb-jar.xml jndi2 "" "" ][ "Address Book Sample EJB"
 com_ibm_websphere_samples_webservices_addr_AddressBookJ2WE AddressBookJ2WE.jar,META-INF/ejb-jar.xml
 jndi3 "" "" ]]
-MapWebModToVH [[ "module_name" web_services.war,WEB-INF/web.xml default_host ]
 [ "AddressBook Bottom Up Java Bean" AddressBookJ2WB.war,WEB-INF/web.xml default_host ]
 [ "AddressBook Top Down Java Bean" AddressBookW2JB.war,WEB-INF/web.xml default_host ]]]' )
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindPreferredPort',
 [['.*', '.*', '.*', '.*', 'WSLoggerJMSPort']]])
```

WebServicesServerBindPort

The WebServicesServerBindPort option sets two attributes of a web service port. The immutable values identify the port of a web service that you are modifying. The scope fields include: Module, Web service and Port. The mutable values include: `WSDL Service Name`, and `Scope`.

The scope determines the life cycle of implementing the Java bean. The valid values include: `Request` (new instance for each request), `Application` (one instance for each web-app), and `Session` (new instance for each HTTP session).

The scope attribute does not apply to web services that a Java Message Service (JMS) transport. The scope attribute does not apply to enterprise beans.

The WSDL service name identifies a service when more than one service has the same port name. The WSDL service name is represented as a QName string, for example, `{namespace}localname` .

Batch mode example usage

Using Jacl:

```
$AdminApp install WebServicesSamples.ear {-WebServicesServerBindPort
 {{AddressBookW2JE.jar service/WSLoggerService2 WSLoggerJMS {} Session}}}
```

Using Jacl with pattern matching:

```
$AdminApp install WebServicesSamples.ear {-WebServicesServerBindPort
 {{.*  WSClientTestService WSClientTest Request} {.*  StockQuoteService StockQuote Application}
 {.*  StockQuoteService StockQuote2 Session}}}
```

Using Jython:

```
AdminApp.install('WebServicesSamples.ear', '[-WebServicesServerBindPort
 [[AddressBookW2JE.jar service/WSLoggerService2 WSLoggerJMS "" Session]]]')
```

Using Jython with pattern matching:

```
AdminApp.install('WebServicesSamples.ear', ['-WebServicesServerBindPort',
 [['.*', 'WSClientTestService', 'WSClientTest', 'Request'],
 ['.*', 'StockQuoteService', 'StockQuote', 'Application'],
 ['.*', 'StockQuoteService', 'StockQuote2', 'Session']]])
```

WebServicesClientCustomProperty

The WebServicesClientCustomProperty option supports the configuration of the name value parameter for the description of the client bind file of a web service. The immutable values identify the port of the web service that you are modifying. The scope fields include: Module, Web service, and Port. The mutable values include: `name` and `value`.

The format of the `name` and `value` values include a string that represents multiple name and value pairs by using the + character as a separator. For example, name string = "n1+n2+n3" value string = "v1+v2+v3" yields name/value pairs: {{"n1" "v1"}, {"n2" "v2"}, {"n3" " v3"}},

Batch mode example usage

Using Jacl:

```
$AdminApp edit WebServicesSamples {-WebServicesClientCustomProperty
 {{join.jar com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest service/StockQuoteService
 STockQuote propname1 propValue1}
{ejbclientonly.jar Exchange service/STockQuoteService STockQuote propname2 propValue2}}}
```

Using Jacl with pattern matching:

```
$AdminApp edit WebServicesSamples {-WebServicesClientCustomProperty
 {{join.jar com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest .* .* propname1 propValue1}
 {ejbclientonly.jar Exchange .* .* propname2 propValue2}}}
```

Using Jython:

```
AdminApp.edit('WebServicesSamples', ['-WebServicesClientCustomProperty',
 [['join.jar', 'com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest',
 'service/StockQuoteService', 'STockQuote', 'propname1', 'propValue1'],
 ['ejbclientonly.jar', 'Exchange', 'service/STockQuoteService', 'STockQuote',
 'propname2', 'propValue2']]])
```

Using Jython with pattern matching:

```
AdminApp.edit('WebServicesSamples', ['-WebServicesClientCustomProperty',
 [['join.jar', 'com_ibm_ws_wsfvt_test_multiejbjar_client_WSClientTest',
 '.*', '.*', 'propname1', 'propValue1'],
 ['ejbclientonly.jar', 'Exchange', '.*', '.*', 'propname2', 'propValue2']]])
```

WebServicesServerCustomProperty

The WebServicesServerCustomProperty option supports the configuration of the name value parameter for the description of the server bind file of a web service. The scoping fields include the following: Module, EJB, and web service. The mutable values for this task include: `name` and `value`.

The format of the these values include a string that represents multiple name and value pairs by using the plus (+) character as a separator. For example, name string = "n1+n2+n3" value string = "v1+v2+v3" yields name and value pairs: {{"n1" "v1"}, {"n2" "v2"}, {"n3" " v3"}}.

Batch mode example usage

Using Jacl:

```
$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty
 {{AddressBookW2JE.jar AddressBookService AddressBook
 com.ibm.websphere.webservices.http.responseContentEncoding deflate}}}
```

Using Jacl with pattern matching:

```
$AdminApp edit WebServicesSamples {-WebServicesServerCustomProperty
 {{.* .* .* com.ibm.websphere.webservices.http.responseContentEncoding deflate}}}
```

Using Jython:

```
AdminApp.edit ( 'WebServicesSamples', '[ -WebServicesServerCustomProperty
 [[AddressBookW2JE.jar AddressBookService AddressBook
 com.ibm.websphere.webservices.http.responseContentEncoding deflate]]]')
```

Using Jython with pattern matching:

```
AdminApp.edit ( 'WebServicesSamples', ['-WebServicesServerCustomProperty', [['.*', '.*',
 '.*', 'com.ibm.websphere.webservices.http.responseContentEncoding', 'deflate']]])
```

Usage table for the options of the AdminApp object install, installInteractive, update, updateInteractive, edit, and editInteractive commands using wsadmin scripting

This table lists all of the options available for the **install**, **installInteractive**, **update**, **updateInteractive**, **edit**, and **editInteractive** commands of the AdminApp object.

The table indicates the applicable commands for each option. Some option names are split on multiple lines for printing purposes.

Table 577. AdminApp object options. See what commands apply to the options.

Option name	install and installInteractive commands - Install an application	update and updateInteractive commands - Update an application	update and updateInteractive commands - Add a module	update and updateInteractive commands - Update a module	edit and editInteractive commands - Edit an application	edit and editInteractive commands - Edit a module
ActSpecJNDI	Yes	Yes	Yes	Yes	Yes	Yes
allowDispatch RemoteInclude	Yes	Yes	No	No	Yes	No
allowPermInFilterPolicy	Yes	Yes	No	No	No	No
allowService RemoteInclude	Yes	Yes	No	No	Yes	No
appname	Yes	Yes	No	No	No	No
BackendIdSelection	Yes	Yes	Yes	Yes	No	No
BindJndiForEJBMessageBinding	Yes	Yes	Yes	Yes	Yes	Yes
BindJndiForEJBNonMessageBinding	Yes	Yes	Yes	Yes	Yes	Yes
cell	Yes	Yes	Yes	No	No	No
cluster	Yes	Yes	Yes	No	No	No
contents		Yes	Yes	Yes	No	No
contenturi		Yes	Yes	Yes	No	No
contextroot	Yes	Yes	Yes	No	No	No
CorrectOracleIsolationLevel	Yes	Yes	Yes	Yes	Yes	Yes
CorrectUseSystemIdentity	Yes	Yes	Yes	Yes	Yes	Yes
createMBeansForResources	Yes	Yes	No	No	Yes	No
CtxRootForWebMod	Yes	Yes	Yes	Yes	Yes	Yes
custom	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor10CMPBeans	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor20CMPBeans	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor10EJBModules	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor20EJBModules	Yes	Yes	Yes	Yes	Yes	Yes
defaultbinding.datasource.jndi	Yes	Yes	Yes	Yes	No	No
defaultbinding.cf.jndi	Yes	Yes	Yes	Yes	No	No
defaultbinding.cf.resauth	Yes	Yes	Yes	Yes	No	No
defaultbinding.datasource.password	Yes	Yes	Yes	Yes	No	No
defaultbinding.datasource.username	Yes	Yes	Yes	Yes	No	No
defaultbinding.ejbjndi.prefix	Yes	Yes	Yes	Yes	No	No
defaultbinding.force	Yes	Yes	Yes	Yes	No	No
defaultbinding.strategy.file	Yes	Yes	Yes	Yes	No	No
defaultbinding.virtual.host	Yes	Yes	Yes	Yes	No	No
depl.extension.reg **Note:** depl.extension.reg is deprecated in Version 7.0	No	No	No	No	No	No
deployejb	Yes	Yes	Yes	Yes	No	No
deployejb.classpath	Yes	Yes	Yes	Yes	No	No
deployejb.complianceLevel	Yes	Yes	Yes	Yes	No	No
deployejb.dbschema	Yes	Yes	Yes	Yes	No	No
deployejb.dbtype	Yes	Yes	Yes	Yes	No	No

Table 577. AdminApp object options (continued). See what commands apply to the options.

Option name	install and installInteractive commands - Install an application	update and updateInteractive commands - Update an application	update and updateInteractive commands - Add a module	update and updateInteractive commands - Update a module	edit and editInteractive commands - Edit an application	edit and editInteractive commands - Edit a module
deployejb.dbaccesstype	Yes	Yes	Yes	Yes	No	No
deployejb.rmic	Yes	Yes	Yes	Yes	No	No
deployejb.sqljclasspath	Yes	Yes	Yes	Yes	No	No
deployws	Yes	Yes	Yes	Yes	No	No
deployws.classpath	Yes	Yes	Yes	Yes	No	No
deployws.jardirs	Yes	Yes	Yes	Yes	No	No
distributeApp	Yes	Yes	No	No	Yes	No
EmbeddedRar	Yes	Yes	Yes	Yes	Yes	Yes
EnsureMethod ProtectionFor 10EJB	Yes	Yes	Yes	Yes	No	No
EnsureMethod ProtentionFor 20EJB	Yes	Yes	Yes	Yes	No	No
filepermission	Yes	Yes	No	No	Yes	No
JSPCompile Options	Yes	Yes	Yes	Yes	No	No
JSPReloadFor WebMod	Yes	Yes	Yes	Yes	Yes	Yes
installdir **Note:** installdir is deprecated in Version 7.0	No	No	No	No	No	No
installed.ear.destination	Yes	Yes	No	No	Yes	No
MapInitParam ForServlet	Yes	Yes	Yes	Yes	Yes	Yes
MapMessage Destination RefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapJaspiProvider	Yes	Yes	Yes	Yes	Yes	Yes
MapModulesToServers	Yes	Yes	Yes	Yes	Yes	Yes
MapEJBRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapEnvEntryFor WebMod	Yes	Yes	Yes	Yes	Yes	Yes
MapResEnvRefToRes	Yes	Yes	Yes	Yes	Yes	Yes
MapResRefToEJB	Yes	Yes	Yes	Yes	Yes	Yes
MapRolesToUsers	Yes	Yes	No	No	Yes	Yes
MapRunAsRolesToUsers	Yes	Yes	Yes	Yes	Yes	Yes
MapSharedLib ForMod	Yes	Yes	Yes	Yes	Yes	Yes
MapWebModToVH	Yes	Yes	Yes	Yes	Yes	Yes
noallowDispatch Remote Include	Yes	Yes	No	No	Yes	No
noallowPermIn FilterPolicy	Yes	Yes	No	No	No	No
noallowService Remote Include	Yes	Yes	No	No	Yes	No
nocreateMBeans ForResources	Yes	Yes	No	No	Yes	No
node	Yes	Yes	Yes	No	No	No
nodeployejb	Yes	Yes	Yes	Yes	No	No
nodeployws	Yes	Yes	Yes	Yes	No	No
nodistributeApp	Yes	Yes	No	No	Yes	No
nopreCompileJSPs	Yes	Yes	Yes	Yes	No	No
noprocessEmbedded Config	Yes	Yes	No	No	No	No
noreloadEnabled	Yes	Yes	No	No	Yes	No
nousedefaultbindings	Yes	Yes	Yes	Yes	No	No
nouseMetaData FromBinary	Yes	Yes	No	No	Yes	No
operation	No	Yes	Yes	Yes	No	No
preCompileJSPs	Yes	Yes	Yes	Yes	No	No
processEmbedded Config	Yes	Yes	No	No	No	No

Table 577. AdminApp object options (continued). See what commands apply to the options.

Option name	install and installInteractive commands - Install an application	update and updateInteractive commands - Update an application	update and updateInteractive commands - Add a module	update and updateInteractive commands - Update a module	edit and editInteractive commands - Edit an application	edit and editInteractive commands - Edit a module
reloadEnabled	Yes	Yes	No	No	Yes	No
reloadInterval	Yes	Yes	No	No	Yes	No
server	Yes	Yes	Yes	No	No	No
target	Yes	Yes	Yes	No	No	No
update	Yes	Yes	No	No	No	No
update.ignore.old	Yes	Yes	No	Yes	No	No
update.ignore.new	Yes	Yes	No	Yes	No	No
useMetaData FromBinary	Yes	Yes	No	No	Yes	No
usedefaultbindings	Yes	Yes	Yes	Yes	No	No
validateinstall	Yes	No	No	No	Yes	No
verbose	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient BindingDeployed WSDL	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient BindPortInfo	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient BindPreferred Port	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesClient CustomProperty	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesServer BindPort	Yes	Yes	Yes	Yes	Yes	Yes
WebServicesServer CustomProperty	Yes	Yes	Yes	Yes	Yes	Yes

Example: Obtaining option information for AdminApp object commands using wsadmin scripting

Use the **taskInfo** command of the AdminApp object to obtain information about the data that is needed for your application.

You must provide data for rows or entries that are either missing information or require an update.

- You can use the **options** command to see the requirements for an enterprise archive (EAR) file if you construct installation command lines. The **taskInfo** command provides detailed information for each task option with a default binding applied to the result.
- The options for the AdminApp **install** command can be complex if you specify various types of binding information; for example, Java Naming and Directory Interface (JNDI) name, data sources for enterprise bean modules, or virtual hosts for web modules. An easy way to specify command-line installation options is to use a feature of the **installInteractive** command that generates the options for you. After you install the application interactively once and specify all the updates that you need, look for message WASX7278I in the wsadmin output log. The default output log for wsadmin is `wsadmin.traceout`. You can cut and paste the data in this message into a script, and modify it.

Commands for the AdminTask object using wsadmin scripting

Use the AdminTask object to run administrative commands with the wsadmin tool.

Administrative commands are loaded dynamically when you start the wsadmin tool. The administrative commands that are available for you to use, and what you can do with them, depends on the edition of the product that you use.

You can start the scripting client without having a server running by using the `-conntype NONE` option with the wsadmin tool. The AdminTask administrative commands are available in both connected and local modes. If a server is currently running, it is not recommended to run the AdminTask commands in local

mode because any configuration changes made in local mode are not reflected in the running server configuration and vice versa. If you save a conflicting configuration, you can corrupt the configuration.

Configuration note: With the Jacl scripting language, the *subst* command enables you to substitute a previously set value for a variable in the command. For example:

```
set nodeparm "node1"
$AdminTask setJVMMaxHeapSize [subst {-serverName server1 -nodeName $nodeparm
-maximumHeapSize 100}]
```

The following AdminTask commands are available but do not belong to a group:

- "createTCPEndPoint"
- "getTCPEndPoint" on page 948
- "help" on page 949
- "listTCPEndPoints" on page 950
- "listTCPThreadPools" on page 951
- "updateAppOnCluster" on page 951

createTCPEndPoint

The createTCPEndPoint command creates a new endpoint that you can associate with a TCP inbound channel.

Target object

Parent instance of the TransportChannelService that contains the TCPInboundChannel. (ObjectName, required)

Required parameters

-name
Specifies the name for the new endpoint. (String, required)

-host
Specifies the host for the new endpoint. (String, required)

-port
Specifies the port for the new endpoint. (String, required)

Optional parameters

None.

Sample output

The command returns the object name of the endpoint that was created.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask createTCPEndPoint (cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)
  {-name Sample_End_Pt_Name -host mybuild.location.ibm.com -port 8978}
```

- Using Jython string:

```
AdminTask.createTCPEndPoint('cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1',
  '[-name Sample_End_Pt_Name -host mybuild.location.ibm.com -port 8978]')
```

- Using Jython list:

```
AdminTask.createTCPEndPoint('cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1',
 ['-name', 'Sample_End_Pt_Name', '-host', 'mybuild.location.ibm.com', '-port', '8978'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask createTCPEndPoint {-interactive}
```

- Using Jython:

```
AdminTask.createTCPEndPoint('-interactive')
```

getTCPEndPoint

The getTCPEndPoint command obtains the named end point that is associated with either a TCP inbound channel or a chain that contains a TCP inbound channel.

Target object

TCPInboundChannel, or containing chain, instance that is associated with a NamedEndPoint. (ObjectName, required)

Required parameters

None.

Optional parameters

None.

Sample output

The command returns the object name of an existing named end point that is associated with the TCP inbound channel instance or a channel chain.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask getTCPEndPoint TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#TCPInboundChannel_1)

$AdminTask getTCPEndPoint DCS(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#Chain_3)
```

- Using Jython string:

```
print AdminTask.getTCPEndPoint('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#TCPInboundChannel_1)')

print AdminTask.getTCPEndPoint('DCS(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#Chain_3)')
```

- Using Jython list:

```
print AdminTask.getTCPEndPoint('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#TCPInboundChannel_1)')

print AdminTask.getTCPEndPoint('DCS(cells/mybuildCell01/nodes/mybuildCellManager01
/servers/dmgr|server.xml#Chain_3)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask getTCPEndPoint {-interactive}
```

- Using Jython:

```
print AdminTask.getTCPEndPoint('-interactive')
```

help

The **help** command provides a summary of the help commands and ways to invoke an administrative command. You can use wildcard characters (*) or Java regular expressions (.*) in the command syntax to customize the search query.

Target object

None.

Optional parameters

-commands
The **help** command provides a list of available administrative commands if you use the `-commands` parameter. (String, optional)

-commandGroups
The **help** command provides a list of administrative command groups if you use the `-commandGroups` parameter. (String, optional)

-commandName
The **help** command provides help information for a given administrative command. (String, optional)

-stepName
The **help** command provides help information for a given step of an administrative command. (String, optional)

Sample output

The command returns general command information for the AdminTask object.

Examples

Batch mode example usage:

The following command examples return general help information for the AdminTask object:

- Using Jacl:

```
$AdminTask help
```

- Using Jython:

```
print AdminTask.help()
```

The following command examples display each command for the AdminTask object:

- Using Jacl:

```
$AdminTask help -commands
```

- Using Jython:

```
print AdminTask.help('-commands')
```

The following command examples return detailed command information for the createJ2CConnectionFactory command for the AdminTask object:

- Using Jacl:

```
$AdminTask help createJ2CConnectionFactory
```

- Using Jython:

```
print AdminTask.help('createJ2CConnectionFactory')
```

The following examples demonstrate the use of the wildcard character (*) to return each command that contains the `create` string:

- Using Jacl:

```
$AdminTask help -commands *create*
```

- Using Jython:

```
print AdminTask.help('-commands *create*')
```

The following examples demonstrate the syntax to use regular Java expressions (.*):

- Using Jacl:

```
$AdminTask help -commands <pattern>
```

- Using Jython:

```
print AdminTask.help('-commands <pattern>')
```

listTCPEndPoints

The listTCPEndPoints command lists all the named end points that can be associated with a TCP inbound channel.

Target object

TCP Inbound Channel instance for which named end points candidates are listed. (ObjectName, required)

Required parameters

None.

Optional parameters

-excludeDistinguished
Specifies whether to show only non-distinguished named end points. This parameter does not require a value. (Boolean, optional)

-unusedOnly
Specifies whether to show the named end points not in use by other TCP inbound channel instances. This parameter does not require a value. (Boolean, optional)

Sample output

The command returns a list of object names for the eligible named end points.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listTCPEndPoints TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
$AdminTask listTCPEndPoints TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
  {-excludeDistinguished}
$AdminTask listTCPEndPoints TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
   {-excludeDistinguished -unusedOnly}
```

- Using Jython string:

```
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   '[-excludeDistinguished]')
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   '[-excludeDistinguished]')
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   '[-excludeDistinguished -unusedOnly]')
```

- Using Jython list:

```
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
  ['-excludeDistinguished'])
```

```
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
   ['-excludeDistinguished'])
print AdminTask.listTCPEndPoints('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)',
  ['-excludeDistinguished', '-unusedOnly'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listTCPEndPoints {-interactive}
```

- Using Jython:

```
print AdminTask.listTCPEndPoints('-interactive')
```

listTCPThreadPools

The listTCPThreadPools command lists all of the thread pools that can be associated with a TCP inbound channel or TCP outbound channel.

Target object

TCPInboundChannel or TCPOutboundChannel instance for which ThreadPool candidates are listed. (ObjectName, required)

Required parameters

None.

Optional parameters

None.

Sample output

The command returns a list of eligible thread pool object names.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask listTCPThreadPools TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)
```

- Using Jython string:

```
print AdminTask.listTCPThreadPools('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)')
```

- Using Jython list:

```
print AdminTask.listTCPThreadPools('TCP_1(cells/mybuildCell01/nodes/mybuildCellManager01/servers/dmgr|server.xml#TCPInboundChannel_1)')
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask listTCPThreadPools {-interactive}
```

- Using Jython:

```
print AdminTask.listTCPThreadPools('-interactive')
```

updateAppOnCluster

The updateAppOnCluster command can be used to synchronize nodes and restart cluster members for an application update that is deployed to a cluster. After an application update, this command can be used to synchronize the nodes without stopping all the cluster members on all the nodes at one time. This

command synchronizes one node at a time. Each node is synchronized by stopping the cluster members on which the application is targeted, performing a node synchronization operation, and restarting the cluster members.

This command might take more time than the default connector timeout period, depending on the number of nodes that the target cluster spans. Be sure to set proper timeout values in the `soap.client.props` file in the *profile_root*`/properties` directory, when a SOAP connector is used; in the `sas.client.props` file, when a JSR160RMI connecor or an RMI connector is used; and in the `ipc.client.props` file when an IPC connector is used.

This command is not supported in local mode.

Target object

None.

Required parameters

-ApplicationNames
Specifies the names of the applications that are updated. (String, required)

Optional parameters

-timeout
Specifies the timeout value in seconds for each node synchronization. The default is `300` seconds. (Integer, optional)

Sample output

The command does not return output.

Examples

Batch mode example usage:

- Using Jacl:

```
$AdminTask updateAppOnCluster {-ApplicationNames app1}
$AdminTask updateAppOnCluster {-ApplicationNames app1 -timeout 600}
```

- Using Jython string:

```
AdminTask.updateAppOnCluster('[-ApplicationNames app1]')
AdminTask.updateAppOnCluster('[-ApplicationNames app1 -timeout 600]')
```

- Using Jython list:

```
AdminTask.updateAppOnCluster(['-ApplicationNames', 'app1'])
AdminTask.updateAppOnCluster(['-ApplicationNames', 'app1', '-timeout', '600'])
```

Interactive mode example usage:

- Using Jacl:

```
$AdminTask updateAppOnCluster -interactive
```

- Using Jython:

```
AdminTask.updateAppOnCluster('-interactive')
```

Administrative command invocation syntax using wsadmin scripting

The administrative command uses a specific syntax to invoke operations.

You can use an administrative command in batch mode or in interactive mode. The following syntax is used for an administrative command:

Using Jacl:

```
$AdminTask cmdName [targetObject] [options]
```

where `options` include:

```
{
  [-paramName paramValue] [-paramName] ...
  [-stepName {{stepParamValue ...} ...} ...]
  [-delete {-stepName {{stepKeyParamValue ...} ...} ...} ...]
  [-interactive]
}
```

or

```
{
  [-paramName paramValue] [-paramName] ...
  [-stepName {{stepParamName stepParamValue} {stepParamName stepParamValue} ...]
  [-delete {-stepName {{stepKeyParamValue ...} ...} ...} ...]
  [-interactive]
}
```

Using Jython:

```
AdminTask.cmdName(['targetObject'], [options])
```

where `options` include:

```
'[
[-paramName paramValue] [-paramName ...]
[-stepName [[stepParamValue ...] ...] ...]
[-delete [-collectionStepName [[stepKeyParamValue ...] ...] ...] ...]
[-interactive]
]'
```

or

```
'[
[-paramName paramValue] [-paramName ...]
[-stepName [[stepParamName stepParamValue] [stepParamName stepParamValue] ...]
[-delete [-collectionStepName [[stepKeyParamValue ...] ...] ...] ...]
[-interactive]
]'
```

where:

Table 578. Administrative command parameter descriptions. Run commands with parameters that complete the task.

`cmdName`	represents the name of an administrative command to run.
`targetObject`	represents the target object on which the command operates. Depending on the administrative command, this input can be required, optional, or nonexistent. This input corresponds to the Target object that is displayed in the command-specific help.
`paramName`	represents the parameter name of the command that was run. Depending on the administrative command, this input can be required, optional, or nonexistent. Each parameter name corresponds to an argument name that is displayed in the Arguments area of the command-specific help.
`paramValue`	represents the parameter value to set for the preceding parameter name. Parameters are specified as name-value pairs. The parameter value is not required if a parameter has Boolean as its value type. If you specify the parameter name only, without specifying a value for a Boolean type parameter, the value is set to `true`.
`stepName`	represents the step name of the command. This input corresponds to a step name that is displayed in the Steps area of the command-specific help.

Table 578. Administrative command parameter descriptions (continued). Run commands with parameters that complete the task.

`stepParamName`	represents the parameter name for a step. Depending on the administrative command, this input can be required, optional, or nonexistent. Each parameter name corresponds to an argument name that displays in the step area of the command-specific help.
`stepParamValue ...`	represents the values of the parameters for a step. Provide all the parameter values of a step in the correct order, as displayed in the step-specific help. For any optional parameters that you do not want to specify a value, put "" instead of the value. If a command step is a collection type, for example, it contains multiple objects where each object has the same set of parameters. You can specify multiple objects with each object enclosed by a pair of braces. For collection type steps, each step parameter is a key or a non-key. Key parameters in a step are used to uniquely identify an object in the collection. If data exists in the step, key parameter values that are provided in the input are compared with key parameter values in the existing data. If a match is found, the existing data is updated. Otherwise, if the specified step supports the addition of new objects, the input values are added.
`delete`	represents the option to delete existing data from a specified step that supports collection.
`collectionStepName`	represents the collection step name.
`stepKeyParamValue ...`	represents the values of key parameters to uniquely identify an object to delete from a collection step. You must provide the key parameter values of an object in the order that they are displayed in the step specific help.
`interactive`	represents the option to enter interactive mode.
`[ ]`	represents a Jython list bracket.
[ ]	indicates that the parameters or options inside the brackets are optional. Do not type these brackets as part of the syntax.

Administrative properties for using wsadmin scripting

Scripting administration utilizes several Java property files. Property files can be used to control your system configurations. Before any property file is specified on the command line, three levels of default property files are loaded. These property files include an installation default file, a user default file, and a properties file.

The first level represents an installation default, located in the `profile_root/properties` directory for each application server profile called `wsadmin.properties`. The second level represents a user default, and is located in the Java user.home property. This properties file is also called `wsadmin.properties`. The third level is a properties file that is pointed to by the `WSADMIN_PROPERTIES` environment variable. This environment variable is defined in the environment where the wsadmin tool starts.

If one or more of these property files is present, they are interpreted before any properties file that is present on the command line. The three levels of property files load in the order that they are specified. The properties file that is loaded last overrides the ones loaded earlier.

The following Java properties are used by scripting:

com.ibm.ws.scripting.appendTrace

Determines if the trace file appends to the end of the existing log file. The default setting, `false`, overrides the log file on each invocation.

com.ibm.ws.scripting.classpath

Searches for classes and resources, and is appended to the list of paths.

com.ibm.ws.scripting.connectionType

Determines the connector to use. This value can either be `SOAP`, `JSR160RMI`, `RMI`, `IPC`, or `NONE`. The `wsadmin.properties` file specifies `SOAP` as the connector.

com.ibm.ws.scripting.crossDocumentValidationEnabled

Determines whether the validation mechanism examines other documents when changes are made to one document.

Possible values are `true` and `false`. The default value is `true`.

com.ibm.ws.scripting.defaultLang

Indicates the language to use when running scripts. The `wsadmin.properties` file specifies Jacl as the scripting language.

The supported scripting languages are Jacl and Jython.

com.ibm.ws.scripting.echoparams

Determines if the parameters or arguments output to STDOUT or to a wsadmin log file. The default setting, `true`, outputs the parameters or arguments to a log file.

com.ibm.ws.scripting.emitWarningForCustomSecurityPolicy

Controls whether the WASX7207W message is emitted when custom permissions are found.

The possible values are `true` and `false`. The default value is `true`.

com.ibm.ws.scripting.host

Determines the host to use when attempting a connection. If not specified, the default is the local machine.

com.ibm.ws.scripting.ipchost

The `ipchost` property determines the host that the system uses to connect to the IPC connector. Use the host name or IP address of the loopback adapter that the IPC connector listens to, such as `localhost`, `127.0.0.1`, or `::1`. The default value is `localhost`.

com.ibm.ws.scripting.port

Specifies the port to use when attempting a connection. The `wsadmin.properties` file specifies `8879` as the SOAP port for a single server installation.

com.ibm.ws.scripting.profiles

Specifies a list of profile scripts to run automatically before running user commands, scripts, or an interactive shell.

The `wsadmin.properties` file specifies `securityProcs.jacl` and `LTPA_LDAPSecurityProcs.jacl` as the values of this property. If Jython is specified with the wsadmin -lang option, the wsadmin tool performs a conversion to change the profile script names that are specified in this property to use the file extension that matches the language specified. Use the provided script procedures with the default settings to make security configuration easier.

com.ibm.ws.scripting.traceFile

Determines where trace and log output is directed. The `wsadmin.properties` file specifies the `wsadmin.traceout` file that is located in the `profile_root/logs` directory of each application server profile as the value of this property.

If multiple users work with the wsadmin tool simultaneously, set different traceFile properties in the user properties files. If the file name contains double-byte character set (DBCS) characters, use a unicode format, such as `\uxxxx`, where `xxxx` is a number.

com.ibm.ws.scripting.traceString

Turns on tracing for the scripting process. The default has tracing turned off.

com.ibm.ws.scripting.tempdir

Determines the directory to use for temporary files when installing applications.

The Java virtual machine (JVM) API uses `java.io.temp` as the default value.

com.ibm.ws.scripting.validationLevel

Determines the level of validation to use when configuration changes are made from the scripting interface.

Possible values are: `NONE`, `LOW`, `MEDIUM`, `HIGH`, `HIGHEST`. The default is `HIGHEST`.

com.ibm.ws.scripting.validationOutput

Determines where the validation reports are directed. The default file is `wsadmin.valout` which is located in the `profile_root/logs` directory of each application server profile.

If multiple users work with the wsadmin tool simultaneously, set different validationOutput properties in the user properties files. If the file name contains double-byte character set (DBCS) characters, use unicode format, such as `\uxxxx`, where `xxxx` is a number.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
: The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
: The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
: The default Application Client for IBM WebSphere Application Server profile root is the `/QIBM/UserData/WebSphere/AppClient/V8/client/profiles/`*`profile_name`* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the /QIBM/ProdData/WebSphere/AppServer/V8/Base directory.

java_home

Table 579. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit
64–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit

plugins_profile_root
The default Web Server Plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/*profile_name* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V8/webserver directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V8/webserver directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS8*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is QWAS8A. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, was.install.library, and is located under the *app_server_root* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/*profile_name* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is QWAS8. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V8/Base directory.

The profiles and profileRegistry subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Appendix. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Application Client for IBM WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppClient/V8/client` directory.

app_client_user_data_root
The default Application Client for IBM WebSphere Application Server user data root is the `/QIBM/UserData/WebSphere/AppClient/V8/client` directory.

app_client_profile_root
The default Application Client for IBM WebSphere Application Server profile root is the `/QIBM/UserData/WebSphere/AppClient/V8/client/profiles/`*`profile_name`* directory.

app_server_root
The default installation root directory for WebSphere Application Server is the `/QIBM/ProdData/WebSphere/AppServer/V8/Base` directory.

java_home

Table 580. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
32–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit`
64–bit IBM Technology for Java	`/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit`

plugins_profile_root
The default Web Server Plug-ins profile root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver/profiles/`*profile_name* directory.

plugins_root
The default installation root directory for Web Server Plug-ins is the `/QIBM/ProdData/WebSphere/Plugins/V8/webserver` directory.

plugins_user_data_root
The default Web Server Plug-ins user data root is the `/QIBM/UserData/WebSphere/Plugins/V8/webserver` directory.

product_library
product_lib
This is the product library for the installed product. The product library for each Version 8.0 installation on the system contains the program and service program objects (similar to `.exe`, `.dll`, `.so` objects) for the installed product. The product library name is `QWAS8`*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 8.0 product installed on the system is `QWAS8A`. The *`app_server_root`*`/properties/product.properties` file contains the value for the product library of the installation, `was.install.library`, and is located under the *`app_server_root`* directory.

profile_root
The default directory for a profile named *profile_name* for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base/profiles/`*`profile_name`* directory.

shared_product_library
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS8`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

user_data_root
The default user data directory for WebSphere Application Server is the `/QIBM/UserData/WebSphere/AppServer/V8/Base` directory.

The `profiles` and `profileRegistry` subdirectories are created under this directory when you install the product.

web_server_root
The default web server path is /www/*web_server_name*.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

APACHE INFORMATION. This information may include all or portions of information which IBM obtained under the terms and conditions of the Apache License Version 2.0, January 2004. The information may also consist of voluntary contributions made by many individuals to the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org. You may obtain a copy of the Apache License at http://www.apache.org/licenses/LICENSE-2.0.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Index

D