WebSphere® Application Server for z/OS, Version 6.1

Using the administrative clients

SA23-2208-01

Note

Before using this information, be sure to read the general information under “Notices” on page 987.

Compilation date: May 3, 2006

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 919-254-0206.

  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.

Chapter 1. Using the administrative console

This topic provides information on using the administrative console to perform administrative activities.

The administrative console is a Web-based tool that you use to manage the IBM WebSphere Application Server product as well as the Network Deployment product. The administrative console supports a full range of product administrative activities.

1. Optionally install the administrative console through the wsadmin command.
2. Start the server for the administrative console.

   For the Network Deployment product, the administrative console belongs to the deployment manager (dmgr) process, which you start with the **startmanager** command.

   For more information on the deployment manager process, see the *Administering applications and their environment* PDF.
3. Access the administrative console.
4. Specify console preferences.
5. Access help.

Installing and uninstalling the administrative console

You can install the administrative console during profile creation or after you create a profile. You can uninstall any administrative console that you install. To install an administrative console after profile creation, or to uninstall the administrative console, use the **wsadmin** command. This topic discusses how to use the **wsadmin** command to install and uninstall the administrative console.

If you install the administrative console through the **wsadmin** command, a profile that does not have an administrative console installed must exist.

Run the deployConsole script on the **wsadmin** command whenever you want to uninstall the administrative console, or whenever you want to install the administrative console to a profile that does not have an administrative console installed.

You can run the script in either connected or disconnected mode.

The usual security restrictions for the **wsadmin** command apply to this script. In connected mode, the user must authenticate if security is enabled.

Whenever the administrative console is installed, it is installed onto the deployment manager server. The **wsadmin** command attempts to remotely connect to the deployment manager to install or uninstall the administrative console. However, various situations can keep the **wsadmin** command from connecting to the deployment manager, and an error message results. The command cannot connect if:

- The deployment manager is not running.
- The deployment manager is running, but the script was run somewhere other than on the deployment manager. You are most likely running the script on a federated node.

The deployConsole.py script is located in the *profile_root*/bin directory.

- To install the administrative console, issue the following command:

  ```
  wsadmin.sh –f deployConsole.py install
  ```
- To uninstall the administrative console, issue the following command:

  ```
  wsadmin.sh –f deployConsole.py remove
  ```

The administrative console is installed or uninstalled, depending on whether you specified the install or remove option.

Starting and logging off the administrative console

This topic describes how to set up the administrative console environment, to access the administrative console, and to log out of the administrative console.

To access the administrative console, you must first install WebSphere Application Server and the administrative console.

The administrative console application is installed during the initial installation process.

To access the administrative console, you must start it and then log in. After you finish working in the administrative console, save your work and log out.

1. Start the administrative console.
   a. Enable cookies in the Web browser that you use to access the administrative console.
   b. Optionally enable JavaScript. Enablement of JavaScript is recommended so that all the features of the administrative console are available to you.
   c. In the same Web browser, type `http://`*`your_fully_qualified_server_name`*`:9060/ibm/console`, where *your_fully_qualified_server_name* is the fully qualified host name for the machine that contains the administrative server. If security is enabled, your request is redirected to `https://`*`your_fully_qualified_server_name`*`:9043/ibm/console`, where *your_fully_qualified_server_name* is the fully qualified host name for the machine that contains the administrative server.

      If you cannot start the administrative console because the console port conflicts with an application that is already running on the machine, do one of the following actions:
      - Change the port number and propagate the number to the appropriate files:
        1) Change all the occurrences of port 9060 (or the port that is selected during profile creation for WebSphere Application Server) to the port for the console. Make the port changes in the *`installation root`*`/profiles/`*`profile name`*`/config/cells/`*`cell_name`*`/nodes/`*`node_name`*`/servers/`*`server_name`*`/serverindex.xml` file and the *`installation root`*`/profiles/`*`profile name`*`/config/cells/`*`cell_name`*`/virtualhosts.xml` files.
        2) Run the ./wsc2n.sh script from the *`installation root`*`/WebSphere/AppServer/bin` directory. The ./wsc2n.sh script generates the `was.env` file, the `control.jvm.options` file, the `servant.jvm.options` file, and the `adjunct.jvm.options` file for each server and the `was.env` file for the location service daemon. These generated files will contain the updated administrative console port number.
      - Shut down the other application that uses the conflicting port before starting the WebSphere Application Server product.

      For a listing of supported Web browsers, see WebSphere Application Server system requirements at

      `http://www.ibm.com/support/docview.wss?rs=180&uid=swg27006921`

      The Web address displays on two lines for printing purposes. Enter the Web address on one line in your browser.
   d. Wait for the console to load into the browser.

      A login page displays after the console starts.
2. Log into the console.
   a. Enter your user name or user ID.

      The user ID lasts only for the duration of the session for which it is used to log in.

Changes made to server configurations are saved to the user ID. Server configurations also are saved to the user ID if a session timeout occurs.

See the *Administering applications and their environment* PDF for more information on server configurations.

If you enter an ID that is already in use and in session, you are prompted to do one of the following actions:

- Log out the other user with the same user ID. You can recover changes made during the other user's session.
- Return to the login page and enter a different user ID.

b. If the console is secure, you must also enter a password for the user name. The console is secure if someone has taken the following actions for the console:
   - Specified security user IDs and passwords
   - Enabled global security

   See the *Securing applications and their environment* PDF for more information.

c. Click **OK**.

3. Log off the administrative console. Click **System administration > Save changes to Master Repository > Save** to save work. Then click **Logout** to exit the console.

   If you close the browser before saving your work, you can recover any unsaved changes the next time that you log in under the same user ID.

You have set up the administrative console environment, accessed the administrative console, and logged out of the administrative console.

Use the administrative console to manage WebSphere Application Server.

Logging in

To access the console, enter your **User ID** and **Password** and then click **Log in**. The password is required only if security is enabled.

After you are logged in, be sure to use the **Logout** link in the console toolbar when you are finished using the console and to prevent unauthorized access. If there is no activity during this login session for an extended period of time, the session expires and you must login again to access the console. The administrator can change the session timeout. See "Changing the console session expiration" on page 27 for more information. The default is set to 30 minutes.

If the user ID that you provide is already logged in at a different location, you are prompted to choose between logging out from the other location or returning to the login page. If you log out the user from the other location, you might be prompted to recover unsaved changes made by that user.

If you have one or more different standalone servers running on the same machine and wish to administer them concurrently from the same or a different machine then you must:

1. Ensure that each server uses a unique value for its admin console port.
2. Run a separate web browser process for each admin console that you wish to access concurrently.

Save changes to the master configuration

Use this topic to update the master repository with your administrative console changes, to discard your administrative console changes and continue working with the master repository, or to continue working with your administrative console changes that are not saved to the master repository.

Until you save changes to the master repository, the administrative console uses a local workspace to track your changes.

Total changed documents

Specifies the total number of documents that you changed for your session, but that are not saved to the master repository. By clicking the +/- toggle key, you can see additional information about the changed documents:

- **Changed items**

  When you change your local configuration, each path and configuration file that you can apply the update to in the master repository is displayed in the list.

- **Status**

  The status can contain the following options:

  – **Added:** If you save your changes to the master repository, a new configuration file is created on the indicated path.

  – **Updated:** If you save your changes to the master repository, an existing configuration file is updated on the indicated path.

  – **Deleted:** If you save your changes to the master repository, an existing configuration file is deleted on the indicated path.

Synchronize changes with nodes

Specifies whether you want to force node synchronization at the time that you save your changes to the master repository, rather than when node synchronization normally occurs.

Save conflict

Specifies that another user changed some configuration information since you began making changes. You can either click **Save** to overwrite the other user information, or **Discard** to discard your changes and keep the changes that the other user made.

Administrative console buttons

This page describes the button choices that are available on various pages of the administrative console, depending on which product features you enable.

-  **Select all items.** Selects each resource that is listed on the administrative console panel, in preparation for performing an action against the selected resources.
- **Deselect all items.** Removes all the listed resources from each selection so that no action is performed against any of the resources.
- **Show filter function.** Produces a dialog box for specifying the resources to view in the table on this administrative console page.

  Hide filter function. Hides the dialog box for specifying the resources to view in the table on this administrative console page.

  When you produce the dialog box, select the column to filter and enter the filter criteria.

  Column to filter

  Select the column to filter from the drop-down list. When you apply the filter, only those items in the selected column that meet the filter criteria are displayed.

  For example, select **Names** to enter criteria by which to filter application server names.

  Filter criteria

  Enter a string that must be found in the name of a collection entry to qualify the entry to display in the collection table. The string can contain percent sign (%), asterisk (*), or question mark (?) symbols as wildcard characters. For example, enter `*App*` to find any application server whose name contains the string `App`.

  Prefix each of the following characters ( ) ^ * % { } \ + $ with a backslash (\) so that the regular expression engine performing the search correctly matches the search criteria. For example, to search for all Java DataBase Connectivity (JDBC) providers containing (`XA`) in the provider name, specify the following string:

*\(XA\)

- **Clear filter value.** Clears your filter changes and restores the most recently saved values.
- **Abort.** Stops a transaction that is not yet in the prepared state. All operations that the transaction completed are undone.
- **Activate.** Activates a group member.
- **Add.** Adds the selected or typed item to a list, or produces a dialog for adding an item to a list.
- **Add Node.** Displays the Add Node page, in which you specify the host name and SOAP connector port for a node that you want added to a cell.
- **Apply.** Saves your changes to a page without exiting the page.
- **Back.** Displays the previous page or item in a sequence. The administrative console does not support using the Back and Forward options of a browser, which can cause intermittent problems. Use Back or Cancel on the administrative console panels instead.
- **Balance.** Balances active members in high availability groups across servers that host the high availability groups. The administrator must first determine which groups have active members and select those groups before selecting Balance.
- **Browse.** Opens a dialog that enables you to look for a file on your system.
- **Calculate groups.** Calculates the number of high availability groups that are returned based on the match set.
- **Cancel.** Exits the current page or dialog, discarding unsaved changes. The administrative console does not support using the Back and Forward options of a browser, which can cause intermittent problems. Use Cancel on the administrative console panels instead.
- **Change.** In the context of security, you can search the user registry for a user ID for an application to run under. In the context of container properties, you can change the data source that the container is using.
- **Clear.** Clears your changes and restores the most recently saved values.
- **Clear selections.** Clears any selected cells in the tables on this tabbed page.
- **Close.** Exits the dialog.
- **Collapse all.** Collapses all the expanded items.
- **Commit.** Releases all locks that are held by a prepared transaction and forces the transaction to commit.
- **Copy.** Creates copies of the selected application servers.
- **Create.** Saves your changes to all the tabbed pages in a dialog and exits the dialog.
- **Create tables.** Develops scheduler database tables.
- **Deactivate.** Deactivates a group member. The group member must be in the active state to be deactivated. The deactivate option causes the group member to move to the idle state. The group policy overrides which members are activated and deactivated for a group. The policy is enforced for every member state change. If the deactivate option conflicts with the group policy, the policy resets who is the active member of the group.
- **Delete.** Removes the selected instance.
- **Details.** Shows the details about a transaction.
- **Disable.** Disables a group or group member. When you disable a group or group member, the active group or group member is first deactivated. If the deactivate option is successful, the group or group member moves to the disable state. A disabled group or group member cannot be activated.
- **Disable Auto Start.** Requires you to start the application manually.
- **Discard.** Discards your local changes instead of saving them to the master configuration.
- **Done.** Saves your changes to all the tabbed pages in a dialog and exits the dialog.
- **Down.** Moves through a list.
- **Drop tables.** Removes scheduler database tables.
- **Dump.** Activates a dump of a traced application server.
- **Edit.** Lets you edit the selected item in a list, or produces a dialog box for editing the item.
- **Enable.** Enables a group or a group member.
- **Enable Auto Start.** Starts an application automatically when the server on which the application resides starts.
- **Expand all.** Expands all the collapsed items.
- **Export.** Accesses a page for exporting enterprise archive (EAR) files for an enterprise application.

- **Export DDL.** Accesses a page for exporting data definition language (DDL) files for an enterprise application.
- **Export Keys.** Exports Lightweight Third-Party Authentication (LTPA) keys to other domains.
- **Export route table.** Exports the route table information for a selected cluster to a binary file in the configuration.
- **Filter.** Produces a dialog box for specifying the resources to view in the tables on this tabbed page.
- **Finish.** Forces a transaction to finish, regardless of whether its outcome has been reported to all participating applications.
- **First.** Displays the first record in a series of records.
- **Force delete.** Forces the removal of a node that is not removed properly from the cell in the master repository. The **Remove node** action is preferred over the **Force delete** action to delete a node from the configuration. If you click **Force delete**, but the node still exists in the configuration, uninstall the node or run the **removeNode** command by using the -force parameter on that node. Force delete action is equivalent to running the cleanupNode command at the deployment manager.
- **Full resynchronize.** Synchronizes the user's configuration immediately. Click full resynchronize on the Nodes page if automatic configuration synchronization is disabled, or if the synchronization interval is set to a long time, and a configuration change is made to the cell repository that needs to be replicated to that node. Clicking this option clears all synchronization optimization settings and performs configuration synchronization again, so no mismatches occur between node and cell configuration after this operation is performed. This operation can take awhile to perform.
- **Generate keys.** Generates new LTPA keys. When security is turned on for the first time with LTPA as the authentication mechanism, LTPA keys are automatically generated with the password entered in the panel. To generated new keys, use this option after the server is up with security turned on. Clicking this option generates the keys and propagates them to all active servers (cell, node, and application servers). The new keys can be used to encrypt and decrypt the LTPA tokens. Click **Save** on the console taskbar to save the new keys and the password in the repository.
- **Immediate stop.** Stops the server, but bypasses the normal server quiesce process that supports in-flight requests to complete before shutting down the entire server process. This shutdown mode is faster than the normal server stop processing, but some application clients can receive exceptions.
- **Import keys.** Imports new LTPA keys from other domains. To support single sign-on (SSO) in WebSphere Application Server across multiple WebSphere domains (cells), share LTPA keys and a password among the domains. After exporting the keys from one of the cells into a file, click this option to import the keys into all the active servers (cell, node, and application servers). The new keys can be used to encrypt and decrypt the LTPA token. Click **Save** on the console taskbar to save the new keys and the password in the repository.
- **Install.** Displays the Preparing for application installation page, which you use to deploy an application, an enterprise bean, or a Web component onto an application server.
- **Install RAR.** Opens a dialog that is used to install a Java 2 Platform, Enterprise Edition Connector Architecture (JCA) connector and to create a resource adapter.
- **Manage state.** Displays a list of MBeans that corresponds to your previous selection of data source or connection factory configurations. You can apply JCA lifecycle management operations to these MBeans to control the run-time status of the corresponding resources.
- **Manage transactions.** Displays a list of active transactions running on a server. You can forcibly finish any transaction that has stopped processing because a transactional resource is not available.
- **Modify.** Opens a dialog that is used to change a specification.
- **Move.** Moves the selected application servers to a different location in the administrative cell. When prompted, specify the target location.
- **Move down.** Moves downward through a list.
- **Move up.** Moves upward through a list.
- **New.** Displays a page that you use to define a new instance. For example, clicking **New** on the Application Servers page displays a page on which you can configure a new application server.
- **Next.** Displays the next page, frame, or item in a sequence.
- **OK.** Saves your changes and exits the page.
- **Pause.** In the context of JCA lifecycle management, stops all outbound communication that is conducted through a resource on a specified server to a backend.
- **Ping.** Attempts to contact selected application servers.

- **Previous.** Displays the previous page, frame, or item in a sequence.
- **Quit.** Exits a dialog box and discards any unsaved changes.
- **Reference shared libraries.** Opens the collection of shared library references available for use by your application or module. If no references are available, a message is displayed stating that there are no references.
- **Refresh.** Refreshes the view of data for instances that are currently listed on this tabbed page.
- **Regenerate encryption key.** Regenerates a key for global data replication. If you are using the DES or TRIPLE_DES encryption type, regenerate a key at regular intervals (for example, monthly) to enhance security.
- **Remove.** Deletes the selected item.
- **Remove file.** Removes the specified file from the selected application or module.
- **Remove node.** Deletes the selected node.
- **Reset.** Clears your changes on the tab or page and restores the most recently saved values.
- **Restart all servers on node.** Stops all application servers on the node and starts them again.
- **Resume.** In the context of JCA lifecycle management, restarts the activity of a data source or a connection factory that was paused by a previous JCA lifecycle management operation.
- **Retrieve new.** Retrieves a new record.
- **Rollout update.** Sequentially updates an application that is installed on multiple cluster members across a cluster. After you update application files or a configuration, click **Rollout update** to install the configuration or the updated files for an application on all the cluster members of a cluster on which the application is installed. The Rollout update option applies the following steps to each cluster member in sequence:

  1. Saves an updated configuration.
  2. Stops the cluster member.
  3. Updates the application on the node by synchronizing the configuration.
  4. Restarts the cluster member.

  Use **Rollout Update** if the application is deployed on one or more clusters spread across multiple nodes. This action reduces the amount of time that any single cluster member is unavailable to serve requests to the smallest interval possible. For a single node environment, use **Update** and then save and synchronize the node instead.
- **Save.** Saves the changes in your local configuration to the master configuration.
- **Select.** For resource analysis, lets you select a scope in which to monitor resources.
- **Set.** Saves your changes to settings in a dialog.
- **Settings.** Displays a dialog for editing servlet-related resource settings.
- **Settings in use.** Displays a dialog showing the settings in use.
- **Show groups.** Displays a collection of high availability groups, based on the match set.
- **Show servers.** Displays a collection of servers that are contained in the high availability groups that match the match set.
- **Start.** In the context of application servers, starts selected application servers. In the context of data collection, starts collecting data for the tables on this tabbed page.
- **Stop.** In the context of server components such as application servers, stops the selected server components. In the context of a data collection, stops collecting data for the tables on a tabbed page. In the context of nodes, stops servers on the selected nodes. In the context of deployment managers, stops the deployment manager server.
- **Synchronize.** Synchronizes the user's configuration immediately. Click Synchronize on the Nodes page if automatic configuration synchronization is disabled, or if the synchronization interval is set to a long time, and a configuration change is made to the cell repository that needs replicating to that node. A node synchronization operation is performed using the normal synchronization optimization algorithm. This operation is fast, but might not fix problems from manual file edits that occur on the node. It is possible for the node and cell configuration to be out of synchronization after this operation is performed. If problems persist, use Full Resynchronize.
- **Terminate.** Deletes the Application Server process or another process that cannot be stopped by the **Stop** or **Immediate Stop** commands. Some application clients can receive exceptions. Always attempt an immediate stop before using this option.

- **Test connection.** After you define and save a data source, you can select this option to ensure that the parameters in the data source definition are correct. On the Collection panel, you can select multiple data sources and test them simultaneously.
- **Uninstall.** Deletes a deployed application from the WebSphere Application Server configuration repository. Also deletes application binary files from the file system.
- **Update.** For applications, replaces an application that is deployed on a server with an updated application. As part of the updating, you might need to complete steps on the Preparing for application installation and Update application pages.

  For clusters, changes the configured weight or runtime weight assigned to a cluster member.
- **Update resource list.** Updates the data on a table. Discovers and adds new instances to the table.
- **Use cell CSI.** Enables Object Management Group (OMG) Common Secure Interoperability (CSI) protocol.
- V6.0.x **Use cell SAS.** Enables IBM Secure Authentication Service (SAS).
- **Use cell Security.** Enables cell security.
- **Verify tables.** Validates the mapping between the table names, scheduler resource, and data sources.
- **View.** Opens a dialog on a file.

Administrative console page features

This topic provides information about the basic elements of an administrative console page, such as the various tabs.

Administrative console pages are arranged in a few basic patterns. Understanding their layout and behavior can help you use them more easily.

Collection pages

Use collection pages to manage a collection of existing administrative objects. A collection page typically contains one or more of the following elements:

Scope Scope is described in “Administrative console scope settings” on page 16.

Preferences
Preferences are described in “Administrative console preference settings” on page 15.

Table of existing objects
The table displays existing administrative objects of the type specified by the collection page. The table columns summarize the values of the key settings for these objects. If no objects exist yet, an empty table is displayed. Use the available options to create a new object.

Buttons for performing actions
The available actions are described on the Administrative console buttons help panel. In most cases, you need to select one or more of the objects in the table, then click an action. The action is applied to the selected objects.

Sort toggle buttons
The column headings in the table are followed by icons for sort ascending (^) and sort descending (v). By default, items such as names are sorted in descending order (alphabetically). To enable another sorting order, click the icons for the column that you want to sort.

Detail pages

Use detail pages to configure specific administrative objects, such as an application server. A detail page typically contains one or more of the following elements:

Configuration tabbed page
This tabbed page is for modifying the configuration of an administrative object. Each configuration page has a set of general properties that is specific to the administrative object. Other sets of properties display on the page, but vary depending on the administrative object.

Runtime tabbed page
This tabbed page displays the configuration that is currently in use for the administrative object. The object is read-only in most cases. Some detail pages do not have runtime tabs.

If you can edit runtime properties, these properties directly affect the current runtime environment, but are not preserved when that environment is stopped.

Local Topology tabbed page
This tabbed page displays the topology that is currently in use for the administrative object. View the topology by expanding and collapsing the different levels of the topology. Some detail pages do not have local topology tabs.

Buttons for performing actions
Buttons to perform specific actions display on the configuration tabbed page and the Runtime tabbed page. The displayed buttons vary based on the administrative object. The available buttons are described on the Administrative console buttons help panel.

Wizard pages

Use wizard pages to complete a configuration process comprised of several steps. Be aware that wizards show or hide certain steps depending on the characteristics of the specific object that you are configuring.

Console layout

This topic describes the layout of the user interface for Integrated Solutions Console.

See Navigating the console for instructions on how to use the console controls.

Banner
Displays a common image across all Integrated Solutions Console installations. The banner includes a greeting to the user who is logged in and links to log out of the console and to open console help.

Navigation tree
Lists the tasks available in the console. Tasks are grouped into organizational nodes that represent categories of tasks, for example, Servers, or Applications. The organizational nodes can be nested in multiple levels.

The tasks shown are only those for which the user has access. When you click a task in the navigation, a page is displayed in the work area containing one or more modules for completing the task. Use the View selection list at the top of the navigation area to modify the list of tasks according to your preferences. You can organize the tasks as follows:

All tasks
This shows all tasks in the console. Tasks are grouped into organizational nodes, for example, Guided activities, Servers, or Applications.

My tasks
This shows only the tasks that you have added to the view. This list is initially empty, but provides a link to the **My Tasks** module. Use **My Tasks** to add and remove from the My Tasks list in the navigation.

Product selection
Selecting a product name shows only the tasks for that particular product, for example, WebSphere Application Server.

Work area
When you launch a page, the content of the page is displayed in the work area. If you have not launched any pages, the Welcome page is displayed in the work area. A page contains one or more console modules that are used to perform operations. Each console module has its own navigation controls. Some pages include a control to close the page and return to the Welcome page.

Navigating the console

This topic describes how to navigate pages and tasks in the Integrated Solutions Console.

- Launching pages from the navigation tree
- Filtering tasks in the navigation
- Using the title bar controls
- Accessing help
- Using the console help controls

Be sure you understand the terms and information in the “Console layout” on page 9 before reading this section.

Launching pages from the navigation tree

The console navigation provides a hierarchical view of all of the *tasks* available in the console. A task is a page in the work area consisting of one or more console modules. All of the modules on the page are provided to start and complete the task. To open a task, simply click the task name in the navigation. The task is opened in a new page in the work area.

The following table describes the controls for the console navigation tree and entries in the tree.

Icon	Function
	Represents an organizational node in the navigation tree that contains pages or other navigation nodes. Click the icon to expand the node.
	Closes an organizational node.

Filtering tasks in the navigation

When you first access the console, all tasks to which you have access are displayed in the navigation. Use the view menu at the top of the navigation to filter the list of tasks by product. Or, you can create a customized list by selecting My tasks from the **View** menu. For instructions on creating and managing your custom list of tasks, see My tasks.

Using the title bar controls

Each page contains one or more Web applications or *console modules*. A console module enables you to perform an operation, such as displaying a list or stopping a managed system. The title and the controls for the module are displayed on the title bar. Depending on the functions supported by the module, the following icons might be displayed on the title bar:

- The  icon is displayed if the module allows you to edit settings for the portlet. For example, a module that retrieves performance data could permit you to specify the server to be analyzed. When you click the icon, an edit screen is displayed. Click the icon to return to the previous screen.
- The icon allows you to return to the previous screen.
- The  icon is displayed if help is available for the module. When you click the icon, the help is displayed in a separate browser window.
- The  icon allows you to minimize the module view. When you click the icon, only the title bar is visible on the page. Click the icon to return to the maximize state.

- The  icon allows you to maximize a module view. When you click the icon, the full portlet view is visible on the page. Click the  icon to return to the minimize state.

In addition to the controls on the title bar, a module can include controls for other actions, such as a button to submit input. Some modules have controls that launch other modules. If a module launches another module, the newly launched module is displayed on a new page.

Accessing help

Help is available for the entire console or for a specific module in the console.

To access console help, perform the following steps:

1. Click **Help** on the console toolbar. The Help is displayed in a separate browser window.
2. In the help navigation tree, click the help set you want to view. For example, click **Console help** to view topics that provide helpful information for new console users. Use the console help controls as needed.

To access help for a module on a page, perform the following steps:

1. On the title bar for the module, click the  icon. That icon is displayed only if help is available for the module. The help is displayed in a separate browser window.
2. Close the help window when you are finished viewing it.

Using the console help controls

Icon	Function
	Use these controls to navigate the list of pages you have viewed. Click  to return to the previous help topic that was displayed. Click  to move forward in the history list.
	Click either of these icons to synchronize the navigation tree with the current topic. The current topic will be highlighted in the navigation tree. This function is useful if you followed links from one help topic to other topics and you want to determine where the current topic is listed in the help navigation tree.
	Permits you to add the current page to your browser favorites list or bookmarks.
	Displays a window for printing the help topic that is displayed.
	Maximizes the target view. This control is available for the Table of Contents view, the Search Results view, and the topic display area.
	Restores a maximized view to its normal size.
	Changes the view to the Search Results view. To search all of the help topics, type a word or words in the **Search** field. Enclose a phrase within double quotes. You can use Boolean operators (such as OR) in the search string. To limit the scope of the search, click **Search scope**. Click **GO** to start the search. A list of topics that contain the target strings are displayed in the results frame.

Icon	Function
	Changes from the Search Results view to the Table of Contents view.

Supported browsers

The following Web browsers are supported for use with Integrated Solutions Console:

- Microsoft Internet Explorer V6.0 SP1
- Mozilla V1.7.8
- Firefox V1.5

Console accessibility

- Accessibility features
- Navigating the console by using the keyboard
- Navigating help by using the keyboard

Accessibility features

The Integrated Solutions Console has the following accessibility features:

- The following features are for vision-impaired users:
  - Supports interfaces commonly used by screen readers (Windows systems only)
  - Can be operated by using only the keyboard
  - Communicates all information independent of color
  - Supports interfaces commonly used by screen magnifiers (Windows systems only)
  - Supports the attachment of alternate output devices
  - Provides help information in an accessible format
- The following features are for users who have mobility impairments or limited use of their hands:
  - Allows the user to request more time to complete timed responses
  - Can be operated by using only the keyboard
  - Supports the attachment of alternative input and output devices
- The following features are for the deaf and hard of hearing users:
  - Supports alternatives to audio information
  - Supports adjustable volume control
- The console does not flash the screen at rates that could induce epileptic seizures.

The help system for Integrated Solutions Console has the following accessibility features:

- Uses the accessibility support enabled by the browser that is used to display the help
- Enables navigation by using the keyboard

Navigating the console by using the keyboard

To move through the controls on a particular page, use the Tab key.

To click a link or control on a page using the keyboard, navigate to the link or control and press Enter.

To change the navigation view using the keyboard, follow these steps.

1. Navigate to the **View** selection list using the Tab key.

2. Use the up and down arrows to change the value of the selection list.
3. Press Enter. The tasks displayed in the navigation are changed according to your selection.

Navigating help by using the keyboard

Use the following key combinations to navigate the help system by keyboard:

- To bring the Topic pane (the right hand side) into focus, press Alt+K, and then press Tab.
- In the Topic pane, to go to the next link, press Tab. To go to the previous link, press Shift+Tab.
- To go directly to the Search Results view in the left hand side, press Alt+R, and then press Enter or Up arrow to enter the view.
- To go directly to the Navigation (Table of Contents) view in the left hand side, press Alt+C, and then press Enter or Up arrow to enter the view.
- To navigate your browser history, press Alt+Left arrow to go back. If you have navigated back to a previously view page, you can use Alt+Right arrow to navigate forward again.
- To expand and collapse a node in the navigation tree, tab to the + or - image next to it to bring the image into focus, and then press the Right or Left arrows.
- To go to the next frame in the help system, press F6. To go to the previous frame in the help system, press Shift+F6.
- In the navigation, to move to the next topic node, press the Down arrow or Tab. To move to the previous topic node, press the Up arrow or Shift+Tab.
- To go to the next link, button, or topic node from inside a view, press Tab.
- To scroll all the way up or down in a frame, press Home or End, respectively.
- To print the active pane, press Ctrl+P.
- To move to the search entry field, press Alt+S.

Welcome

The Welcome page displays the products that are installed that use the Integrated Solutions Console for administrative tasks. The page lists the product name and version number. If provided by the product, you can click the product name to display a page that provides more information about the product.

My tasks

Use My tasks to create and edit a list of tasks to view in the console navigation. A task includes a page that contains one or more Web applications, or *console modules*, that are used to complete that task. When you first access the console, all tasks to which you have access are displayed in the navigation. My tasks is especially useful to customize the navigation to show only the tasks you use most often. After you customize your tasks, My Tasks is initially displayed each time you log in to the console.

Follow these general steps to customize your task list in the navigation.

1. Select My tasks from the **View** selection list in the navigation. If you have never used My tasks before, you must click **Add tasks** to open it.
2. Use the checkboxes to select and deselect tasks from the My tasks navigation.
3. To save your changes, click **Apply**.
4. To cancel your changes, click **Reset**.

After applying your selections, your customized task list is displayed in the navigation.

Use the following buttons to customize your task selections.

Apply Saves the current selections.

Reset Backtracks all changes to the selections that were set since the last time My tasks was applied. This is useful if you need to cancel your changes.

Select All
Checks every task.

Deselect All
Unchecks every task.

Expand All
Expands each node in the display and reveals all subtasks in the navigation.

Collapse All
Collapses each node in the display so that only the top level nodes are displayed.

Specifying console preferences

Use this topic to customize how much data displays on an administrative console panel.

Throughout the administrative console are pages that have Preferences fields, Scope fields, and Filter radio buttons. By selecting these fields and radio buttons you can customize how much data is shown.

For example, examine the Preferences field for the Enterprise Applications page:

1. Go to the navigation tree of the administrative console and click **Applications > Enterprise Applications**.
2. Expand **Preferences**.
3. For the **Maximum rows** field, specify the maximum number of rows to display when the collection is large. The default is 20. Rows that exceed the maximum number display on subsequent pages.
4. Select **Retain filter criteria** if you want to retain the last filter criteria that is entered in the filter function. When you return to the Applications page, the page initially uses the retained filter criteria to display the collection of applications in the table following the preferences. Otherwise, clear **Retain filter criteria** and the last filter criteria is not retained.
5. Click **Apply** to apply your selections or click **Reset** to return to the default values. The default is not to enable (not have a check mark beside) **Retain filter criteria**.

Other pages have similar fields and radio buttons that you can use to specify console preferences. While Preferences fields, Scope fields, and Filter buttons control how much data is shown in the console, the **Preferences** option controls general behavior of the console. Click **System administration > Console Preferences** to view the Preferences page.

Console Preferences settings

Use the Console Preferences page to specify how you want features of the administrative console workspace to behave.

To view this administrative console page, click **System administration > Console Preferences**.

Turn on workspace automatic refresh

Specifies whether you want the administrative console workspace to redraw automatically after the administrative configuration changes.

The default is for the workspace to redraw automatically. If you direct the console to create a new instance of, for example, an application server, the Application Servers page refreshes automatically and shows the new server name in the collection of servers.

Specifying that the workspace not redraw automatically means that you must access a page again by clicking the console navigation tree or links on collection pages to see the changes that are made to the

administrative configuration.

Default true (selected)

No confirmation on workspace discard
Specifies whether the confirmation dialog is displayed after a request is received to discard the workspace. The default is to display confirmation dialogs.

Default false (cleared)

Use default scope
Specifies whether the default scope is the administrative console node.

Default false (cleared)

Show the help portlet
Specifies whether the help portlet on the right of the console displays.

Default true (selected)

Synchronize changes with nodes
Specifies whether to synchronize changes that are saved to the deployment manager profile with all the nodes that are running.

Default false (cleared)

Enable command assistance notifications
Specifies whether to send Java Management Extensions (JMX) notifications that contain command assistance data from the administrative console. Enablement of the notifications allows integration with product tools such as the WebSphere Application Server Toolkit (AST) Jython editor. Enablement of this option is recommended for non-production environments only.

Default false (cleared)

Log command assistance commands
Specifies whether to log all the command assistance wsadmin data to a file. This file is saved to `${LOG_ROOT}/server/commandAssistanceJythonCommands_user name.log` , where:

- *server* is the server process where the console runs, such as `dmgr` or `server1`.
- *user name* is the administrative console user name.

Occasionally clean out the file to manage its growth.

Default false (cleared)

Administrative console preference settings
Use the preference settings to specify how you want information to display on an administrative console page.

Maximum rows
Indicates the maximum number of rows to display per page when the collection is large.

Filter history
Indicates whether to use the same filter criteria to display this page the next time that you visit.

Select the **Retain filter criteria** check box to retain the last filter criteria entered. When you return to the page, retained filter criteria control the application collection that is displayed in the table.

Show resources in the scope hierarchy
Select the check box if you want to display the resources in the hierarchy for a particular scope.

The hierarchy is:

- cell > node > server
- cell > cluster

For example, if you select a node scope, all node scope resources and all cell scope resources display for the node.

This preference is available for resource factory panels only.

Show built-in resources
Select the check box if you want to display resources that are pre-defined to support certain internal components of the product. For example, the product includes built-in configurations of a Cloudscape JDBC provider and a data source to support the Universal Description, Discovery and Integration Protocol (UDDI) registry for Web services.

Show confirmation for stop command
Select the check box if you want a confirmation that the **stop** command is successful.

Show confirmation for immediate stop command
Select the check box if you want a confirmation that the **immediate stop** command is successful.

Show confirmation for terminate command
Select the check box if you want a confirmation that the **terminate** command is successful.

Administrative console scope settings
Use this page to specify the level at which a resource is visible on the administrative console panel. A resource can be visible in the administrative console collection table at the cell, node, cluster, or server scope. By changing the value for Scope, you see only the resources that are defined at that scope. The contents of the collection table might change.

If the list of scopes is less than fifty, the console displays a drop-down list of all the scopes available. To change the scope, select any item from the drop-down list.

If the list is fifty or greater, click **Browse** next to a field to see choices for changing the scope of the field. If a field is read-only, you cannot change the scope. For example, if only one server exists, you cannot switch the scope to a different server.

All scopes is the default. You cannot select All scopes to create a new resource. You must select one of the available scopes from the drop down list to create a new resource.

You always create resources at the current scope that is selected in the administrative console panel, even though the resources might be visible at more than one scope.

Resources such as Java Database Connectivity (JDBC) providers, namespace bindings, or shared libraries can be defined at multiple scopes. Resources that are defined at more specific scopes override duplicate resources that are defined at more general scopes:

- The application scope has precedence over all the scopes.
- The server scope has precedence over the node, cell, and cluster scopes.
- The cluster scope has precedence over the node and cell scopes.
- The node scope has precedence over the cell scope.

Despite the scope of a defined resource, the resource properties apply at an individual server level only. For example, if you define the scope of a data source at the cell level, all the users in that cell can look up and use that data source, which is unique within that cell. However, resource property settings are local to each server in the cell. For example, if you define the maximum connections as 10, then each server in that cell can have 10 connections.

The cell scope is the most general scope and does not override any other scope. The recommendation is that you generally specify a more specific scope than the cell scope. When you define a resource at a more specific scope, you provide greater isolation for the resource. When you define a resource at a more general scope, you provide less isolation. Greater exposure to cross-application conflicts occur for a resource that you define at a more general scope.

Cell Limits the visibility to all servers on the named cell. The resource factories within the cell scope are:
- Defined for all servers within this cell
- Overridden by any resource factories that are defined within application, server, cluster, and node scopes that are in this cell and have the same Java Naming and Directory Interface (JNDI) name

The resource providers that are required by the resource factories must be installed on every node within the cell before applications can bind or use them.

Cluster

Limits the visibility to all the servers on the named cluster. All cluster members must at least be at Version 6 to use cluster scope for the cluster. The resource factories that are defined within the cluster scope:
- Are available for all the members of this cluster to use
- Override any resource factories that have the same JNDI name that is defined within the cell scope

The resource factories that are defined within the cell scope are available for this cluster to use, in addition to the resource factories, that are defined within this cluster scope.

Node Limits the visibility to all the servers on the named node. The node scope is the default scope for most resource types. The resource factories that are defined within the node scope:
- Are available for servers on this node to use
- Override any resource factories that have the same JNDI name defined within the cell scope

The resource factories that are defined within the cell scope are available for servers on this node to use, in addition to the resource factories that are defined within this node scope.

Server

Limits the visibility to the named server. The server scope is the most specific scope for defining resources. The resource factories that are defined within the server scope:
- Are available for applications that are deployed on this server
- Override any resource factories that have the same JNDI name defined within the node and cell scopes

The resource factories that are defined within the node and cell scopes are available for this server to use, in addition to the resource factories that are defined within this server scope.

Application

Limits the visibility to the named application. Application scope resources can be viewed and edited from the console, but not created. You can additionally use the WebSphere Application Server Toolkit (AST) or the wsadmin tool to view or edit the application scope resource configuration. The resource factories that are defined within the application scope are available for this application to use only. The application scope overrides all other scopes.

You can configure resources and the product variables under all five scopes. You can configure namespace bindings and shared libraries under cell, node, and server scopes only.

Accessing help and product information from the administrative console

This topic describes how to use administrative console help and how to link to product documentation from the administrative console.

You must have a connection to the Internet to access information about WebSphere Application Server from the Welcome page of the administrative console.

All of the helps panels that you can access from the administrative console, you can access from the WebSphere Application Server Information Center. This topic describes how to access the help panels, the information center, and other product documentation from the administrative console.

- Click **Welcome** on the administrative console navigation tree. In the workspace to the right of the navigation tree, select the appropriate links to access the WebSphere Application Server Information Center, the WebSphere Application Server product information, and the WebSphere Application Server technical information on developerWorks.
- Access help in the following ways:
  - Click either of the following tabs of an online help page:
    - Click the **Help index** tab and select from the list of help panels to view administrative console help information.
    - Click the **Search** tab, provide search terms, and then click **Search**. Under Results, select a help panel that contains the search information.
  - In the help portal that is on the right side of the administrative console panel, do one or all of the following tasks:
    - Click a field label or a list marker in the administrative console panel for the help to display under Field help. Alternatively, place the cursor over the field label or the list marker for the corresponding help to display at the cursor.
    - Click the link under Page help to access the help panel for the administrative console panel. The help panel is the same help panel that displays when you click the **?** icon.
    - If Command assistance is listed, click the link under Command assistance to view wsadmin scripting commands for the last action run for this console panel.

You can continue to access help information from the administrative console. Alternatively, you can access the help information from the WebSphere Application Server Information Center.

You can continue to access the WebSphere Application Server Information Center, the WebSphere Application Server product information, and the WebSphere Application Server technical information on developerWorks from the administrative console. Alternatively you can access the information from the IBM Web site.

Accessing command assistance from the administrative console

Using command assistance, you can view wsadmin scripting commands in the Jython language for the last action run in the administrative console. This topic discusses how to access command assistance from the administrative console.

You must have WebSphere Application Server and the administrative console running to access command assistance.

Use command assistance to see wsadmin scripting commands that correspond to actions in the administrative console. Seeing these commands might help you develop the commands necessary to administer WebSphere Application Server from the wsadmin utility.

If a command assistance link is listed in the help portlet, **wsadmin** commands exist for the last console action that you completed, and command assistance is available for that action.

When command assistance is unavailable in the help portlet: Some console actions do not have **wsadmin** commands directly associated with them. When the help portlet on the right side of the administrative console panel does not have a command assistance link in it, no command assistance data is available for the last console action.

1. Click the link under **Command assistance** to view wsadmin scripting commands for the last action run for this administrative console panel.

   After the Command assistance window opens, it refreshes automatically when new command assistance data is available.

   Examples of actions include a click on a button or a click on a link in the navigation bar, a collection panel, or a detail panel. The editing of forms is not a user action.

   The wsadmin scripting commands display in the Jython language in a secondary window that you can view by clicking on the Command assistance link in the help portlet.

   If you perform an administrative console action after you launch the Command assistance window, whether or not the scripting commands display in the window depends on whether your browser supports Java scripts. If your browser supports Java scripts, the Command assistance window automatically refreshes the command list to reflect the most recent console action. If the browser does not support Java scripts, click the link again under **Command assistance** in the help portal to refresh the command list.
2. To view the description of a specific **wsadmin** command, place your cursor over the command.

   Hover text is displayed.
3. Optionally, log the command assistance data to a file by selecting the **Log command assistance commands** setting on the Preferences page of the administrative console.

   A timestamp and the breadcrumb trail of the panel that produced the command assistance data are provided with the wsadmin data.
4. Optionally, allow command assistance to emit Java Management Extensions (JMX) notifications by selecting the **Enable command assistance notifications** setting on the Preferences page of the administrative console.

   Enablement of the notifications allows integration with product tools such as the WebSphere Application Server Toolkit (AST) Jython editor to assist you in writing scripts.

   The notification type is websphere.command.assistance.jython.*user_name* where *user_name* is the name of the administrative console user.

You have viewed wsadmin scripting commands from the administrative console, optionally logged the commands to a file, and optionally allowed command assistance to emit JMX notifications.

You can continue your administration of the administrative console.

Administrative console actions with command assistance

Using command assistance, you can view wsadmin scripting commands in the Jython language for the last action run in the administrative console. This topic lists the administrative console actions that have **wsadmin** commands available in the command assistance option of the Help portlet.

The table lists the components and the actions in the administrative console that have command assistance for a particular component.

Component	Action
Applications	• List the applications. • Set the application information. • Install the application.
Web servers	• Create a Web server. • Delete a Web server. • List the virtual hosts. • Create Secure Sockets Layer (SSL) virtual hosts. • Generate a plug-in configuration. • Propagate the plug-in configuration. • Propagate the key ring for the plug-in configuration. • Start the Web server. • Stop the Web server. • Ping the Web server. • Terminate the Web server.
Channel framework	• List the SSL repertoires. • List the Transmission Control Protocol (TCP) endpoints. • List the TCP thread pools. • Delete a chain. • List the chains. • Get the TCP endpoint. • Create a TCP endpoint. • Create a chain.
Node groups	• Create a node group. • Remove a node group. • Add a node group member. • Remove a node group member. • Modify a node group property.
Core group	• List eligible bridge interfaces. • Get a named TCP endpoint. • Create a core group. • List the core groups. • Delete a core group. • Move a core group server. • Move a core group cluster.

Component	Action
Clusters	• Create a cluster. • Delete a cluster. • Create a cluster member. • Delete a cluster member. • Start a cluster. • Stop a cluster. • Stop a cluster immediately. • Ripplestart a cluster. • Get a cluster weight table. • Set a cluster weight table. • Set a backup cluster. • Get a backup cluster.
Topology	• Create an unmanaged node. • Remove an unmanaged node. • Synchronize the nodes. • Stop a node. • Stop a node agent. • Restart a node.
Servers	• Create an application server. • Delete an application server. • Create an application server template. • Delete an application server template. • List the application server templates. • Start an application server. • Stop an application server. • Create a generic server. • Delete a generic server. • Create a proxy server. • Delete a proxy server. • List ports for a server. • Modify port settings.
Resources	• Install the resource adapters. • Create a data source. • Create a Java Database Connector (JDBC) provider.
Transaction services	• List the transactions. • Set the total transaction lifetime timeout. • Set the asynchronous response timeout. • Enable file locking. • Enable protocol security. • Get the client inactivity timeout. • Set the client inactivity timeout. • Set the maximum transaction timeout. • Get the transaction log directory. • List the summary information.

Component	Action
Security	• Enable security. • Validate Lightweight Directory Access Protocol (LDAP) connections. • List scoped objects. • Delete scoped objects. • Get the inherited SSL configuration. • Validate the SSL configuration. • Get the SSL configuration. • Create an SSL configuration. • Modify the SSL configuration. • List the SSL ciphers. • List the SSL configuration groups. • Create an SSL configuration group. • Delete an SSL configuration group. • Modify an SSL configuration group. • Create a dynamic SSL configuration. • List the key sets. • Generate a key for a key set. • Create a key set. • List the key set groups. • Create a key set group. • Generate keys for the key set group. • List the keystores. • Validate the keystores. • Change the keystore password. • List the key managers. • Create a key manager. • List the key file aliases. • Create the key reference. • List the trust managers. • Create the trust manager. • Validate the truststore. • List the personal certificates. • Get a personal certificate.

Component	Action
Security (continued)	• Receive a personal certificate. • Create a personal certificate. • Replace a personal certificate. • Extract a personal certificate. • Import a personal certificate. • Export a personal certificate. • Delete a personal certificate. • Add a signer certificate. • Extract a signer certificate. • Retrieve a signer certificate from a port. • Get a signer certificate. • Delete a signer certificate. • List the signer certificates. • Start the certificate expiration monitor. • Create a self-signed certificate. • Exchange signers. • List the schedules. • Create a schedule. • Modify a schedule. • List the notifiers. • Create a notifier. • Modify a notifier. • List the certificate expiration monitors. • Check if the administrator is locked out. • Validate the administrative name. • Add a name to the administrative authorization list. • Open the configuration for the federated repositories. • Modify the configuration for the federated repositories. • Add a base entry to the realm. • Modify the base entry details. • Configure a new LDAP repository. • Modify an existing LDAP repository configuration. • Delete an existing LDAP repository configuration.

Component	Action
Security (continued)	• View performance data for the LDAP repository under an LDAP configuration. • Modify the LDAP performance data. • View the LDAP entity types under the LDAP configuration. • Modify the existing LDAP entity types. • View the group attribute definition under the LDAP configuration. • Modify the group attribute definition under the LDAP configuration. • View the member attributes under the LDAP group attribute definitions. • Configure the member attribute details under the LDAP group attribute definitions. • Delete an existing member attribute detail. • View the dynamic member attributes under the LDAP group attribute definitions. • Configure the dynamic member attributes under the LDAP group attribute definitions. • Delete an existing dynamic member attribute detail. • View the list of repositories to manage. • Configure the federated repositories to use a built-in repository. • Remove the built-in repository from the federated repository configuration. • View the federated repository property extension. • Configure the federated repository property extension. • View the federated repository entry mapping repository. • Configure the federated repository entry mapping repository. • View the federated repository supported entity types list. • View the details of a supported entity type. • Modify an existing supported entity type. • View the authentication mechanism and expiration policy for the federated repository user identity. • Configure the administrative user password.

Component	Action
Service integration	• Create a bus. • Delete a bus. • Add a bus member. • Delete a bus member. • List the bus members. • Delete the messaging engine. • Create a queue. • Create a topic space. • Create an alias destination. • Create a foreign destination. • Mediate a destination. • Unmediate a destination. • Delete a destination. • Create a mediation. • Delete a mediation. • Modify a mediation. • Delete a foreign bus. • Create a Java Message Service (JMS) activation specification. • Modify a JMS activation specification • Delete a JMS activation specification. • List the JMS activation specifications. • Show a JMS activation specification. • Create a JMS connection factory. • Modify a JMS connection factory. • Delete a JMS connection factory. • List the JMS connection factories. • Show a JMS connection factory. • Create a JMS queue connection factory. • Modify a JMS queue connection factory. • Delete a JMS queue connection factory. • List the JMS queue connection factories. • Show a JMS queue connection factory. • Create a JMS topic connection factory. • Modify a JMS topic connection factory. • Delete a JMS topic connection factory. • List the JMS topic connection factories. • Show a JMS topic connection factory.

Component	Action
Service integration (continued)	• Create a JMS queue. • Modify a JMS queue. • Delete a JMS queue. • List the JMS queues. • Show a JMS queue. • Create a JMS topic. • Modify a JMS topic. • Delete a JMS topic. • List the JMS topics. • Show a JMS topic. • Create a WebSphere MQ server. • Modify a WebSphere MQ server. • Modify a WebSphere MQ server bus member. • Add a permitted transport. • Add a user to a bus connector role. • Add a group to a bus connector role. • Add an inbound port to an inbound service. • Add an outbound port to an outbound service. • Connect an endpoint listener to a service integration bus. • Create an endpoint listener. • Create an inbound service. • Create an outbound service. • Delete an endpoint listener. • Delete an inbound service. • Delete an outbound service. • Disconnect an endpoint listener from a service integration bus. • Publish an inbound service to a Universal Description, Discovery, and Integration (UDDI) registry. • Refresh the Web Services Description Language (WSDL) definition for an inbound service. • Refresh the WSDL definition for an outbound service. • Remove an inbound port. • Remove an outbound port. • Set the default outbound port for an outbound service. • Remove an inbound service from a UDDI registry. • Add a target service for routing from a gateway service. • Create a gateway service. • Create a proxy service deployed to a gateway instance. • Delete a gateway service. • Delete a gateway instance. • Delete a proxy service. • Remove a target service.

Administrative console: Resources for learning

Use the following links to find relevant supplemental information about the IBM WebSphere Application Server administrative console. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information:

Administration

- IBM WebSphere Application Server Redbooks

  This site contains a listing of all WebSphere Application Server Redbooks.
- IBM WebSphere developerWorks

  This site is the home of technical information for developers working with WebSphere products. You can download WebSphere software, take a fast path to developerWorks zones, such as VisualAge Java or WebSphere Application Server, learn about WebSphere products through a newcomers page, tutorials, technology previews, training, and Redbooks, get answers to questions about WebSphere products, and join the WebSphere community, where you can keep up with the latest developments and technical papers.
- WebSphere Application Server Support page

  Take advantage of the Web-based Support and Service resources from IBM to quickly find answers to your technical questions. You can easily access this extensive Web-based support through the IBM Software Support portal at Web address `http://www-3.ibm.com/software/support/` and search by product category, or by product name. For example, if you are experiencing problems specific to WebSphere Application Server, click **WebSphere Application Server** in the product list. The WebSphere Application Server Support page appears.

Changing the console session expiration

Run this JACL script to set how long Integrated Solutions Console can be used until the login session expires.

The following JACL script serves as an example of how to set the duration that an Integrated Solutions Console can be used until the login session expires. Other scripting types, such as JYTHON, could be used.

1. Copy the following script into a file.

```
set dep [$AdminConfig getid /Deployment:isclite/]
set appDep [$AdminConfig list ApplicationDeployment $dep]
set sesMgmt [$AdminConfig list SessionManager $appDep]

# check if existing sesMgmt there or not, if not then create a new one, if exist then modify it
if {$sesMgmt == ""} {
     # get applicationConfig to create new SessionManager
     set appConfig [$AdminConfig list ApplicationConfig $appDep]
     if {$appConfig == ""} {
         # create a new one
         set appConfig [$AdminConfig create ApplicationConfig $appDep {}]
         # then create a new SessionManager using new Application Config just created
         set sesMgmt [$AdminConfig create SessionManager $appConfig {}]
     } else {
          #  create new SessionManager using the existing ApplicationConfig
          set sesMgmt [$AdminConfig create SessionManager $appConfig {}]
```

```
     }
}

# get tuningParams config id
set tuningParams [$AdminConfig showAttribute $sesMgmt tuningParams]
if {$tuningParams == ""} {
    # create a new tuningParams
    $AdminConfig  create TuningParams  $sesMgmt {{invalidationTimeout <timeout value>}}

} else {
     #modify the existing one
     $AdminConfig modify $tuningParams {{invalidationTimeout <timeout value>}}

}

# saving the configuration changes
$AdminConfig save
```

2. Change the *<timeout value>* on the two lines of this sample to the new session expiration value. This number specifies the number of minutes the console preserves the session during inactivity.
3. Save the file to any directory using, for example, the filename timeout.jacl.
4. Start the wsadmin scripting client from the <WAS-install>/profiles/<profile_name>/bin directory.
5. Issue the following command.

```
wsadmin -f <path to jacl file>/timeout.jacl
```

Changing the class loader order of the console module deployed in Integrated Solutions Console

Run this JACL script to change the class loader order of the console module deployed in the Integrated Solutions Console.

The following JACL script serves as an example of how to change the class loader order of the console module deployed in the Integrated Solutions Console. Other scripting types, such as JYTHON, could be used.

1. Copy the following script into a file.

```
set app [$AdminConfig getid /Deployment:isclite/]
set webModules [$AdminConfig list WebModuleDeployment $app]

foreach webModule $webModules {
     set uri [$AdminConfig showAttribute  $webModule uri]
     if {$uri  == "<WAR_NAME>"} {
         #modify the classloader for <WAR_NAME>
         set cl [$AdminConfig list Classloader $webModule]
         # check if the classloader exist
         if {$cl == ""} {
                # create a new one with the appropriate mode
                $AdminConfig create Classloader $webModule  {{mode <MODE>}}
         } else {
              # modify the existing one
             $AdminConfig modify $cl  {{mode <MODE>}}
         }
     }
}

# save the configuration change
$AdminConfig save
```

2. Change the *<WAR_NAME>* on the two lines of this sample to the name of the console module file deployed in the Integrated Solutions Console which class loader order you want to change.
3. Change the <MODE> on the two lines of this sample to PARENT_LAST or PARENT_FIRST as required.

4. Save the file to any directory using, for example, the file name classloaderorder.jacl.
5. Start the wsadmin scripting client from the <WAS-install>/profiles/<profile_name>/bin directory.
6. Issue the following command.

```
wsadmin -f <path to jacl file>/classloader.jacl
```

Chapter 2. Developing console modules

Console modules are Web applications that are accessed from Integrated Solutions Console. Console modules provide the business logic and transaction processes that enable administration functions.

The following skills are essential for developing and testing console modules.

- Java 2 Platform , Enterprise Edition (J2EE)
- XML
- Portlet development using the Java Portlet Specification (JSR 168)

The console also supports existing modules that have been developed using Struts and Tiles APIs. This is only for legacy support. New console modules are supported only if they are developed using standard portlet APIs.

- Review the console module samples. The sample console modules provide examples of portal application archives and how to use the APIs and other features.
- Set up the development environment. IBM provides WebSphere Application Server Toolkit and Rational Application Developer to enable you to quickly develop, test, and deploy your portlet applications.
- Develop your first console module. This topic is for developers new to console module development. It is assumed that you already have a development environment, like Rational Application Developer, already prepared. The topics in this section take you through the process of creating a simple console module. The console module in this exercise consists of a single portlet which is deployed to a single page as a member of the sample console modules provided by IBM. Be sure you have successfully deployed the sample console modules before starting.
- Adding advanced API features. The class files for console modules are developed using the portlet API of the Java Portlet Specification (JSR 168). Integrated Solutions Console includes additional APIs for launching pages, passing properties to other console modules, and launching Eclipse-based help. This topic provides information about the APIs available to console modules.

Overview of Integrated Solutions Console

Integrated Solutions Console offers several advantages for implementing administration functions, as described in this topic.

Integrated Solutions Console provides a single, common interface for system administration. It provides the main platform on which IBM and non-IBM products can build administrative user interfaces as individual plug-ins to a common console framework. Standardizing product administration functions to run on the Integrated Solutions Console platform gives them a more common look and feel and a more consistent behavior, thereby reducing the learning curve and adoption as new management components are introduced. Administrators can interact with multiple IBM and non-IBM products from a single browser-based console.

Consistency across administrative interfaces

Integrated Solutions Console provides a common appearance (for example, theme, layout and banner) and behavior (for example, navigation and authentication) to enable consistent user interaction for administering software products.

A standards-based architecture

Integrated Solutions Console provides a standards-based architecture for Web administration. Each Integrated Solutions Console module consists of one or more Web applications that have access to services within the Java 2 Enterprise Edition (J2EE) environment provided by WebSphere Application

Server. The help interface is implemented using the Eclipse open standard. Console modules are developed using the Java Portlet Specification.

Easy deployment of product administration consoles

The Integrated Solutions Console framework provides an XML-based interface for deploying console modules to a console installation. XML descriptors provide the information needed to deploy the portlet, resources, and setup the page layout and navigation in the console. A console module can be easily removed without impact to the remaining console modules.

Accelerated development of solutions

Using Integrated Solutions Console enables products to reduce the time required to develop solutions that require administration functions. The standards-based architecture, common framework, and support for existing investment help reduce the time required to implement solutions.

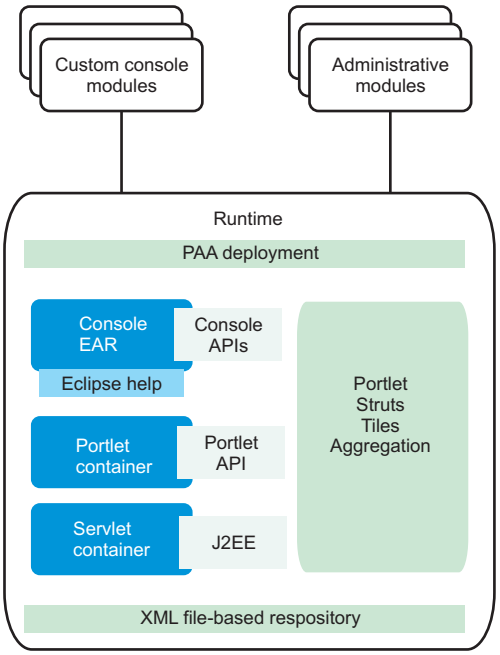
Improved administration efficiency

Customers invest significant resources in training administrative personnel. Providing a common interface across IBM products enables customers to reduce training time and expense.

Integrated Solutions Console components

This topic describes the components of the administrative console.

This figure illustrates the Integrated Solutions Console platform.



The following components and features support console modules running in Integrated Solutions Console.

- A common console

  Provides a common appearance and navigation behavior for console modules. For example, each portlet is rendered using a common skin, rendering the same title bar icons for actions such as editing preferences, maximizing, or requesting help for the portlet.
- Built-in administration console modules

Provide security, log and trace, and other functions that apply to the entire the Integrated Solutions Console installation.

- A portlet container

  Supports portlets developed using the Java Portlet Specification. The current specification as of this writing is JSR 168.

- PAA deployment

  XML descriptors are provided for the developer to specify pages, navigation nodes, and access control for the console module, which are created at deployment. The descriptor can also register the following information about the console module.

  – Custom roles for the application in addition to the roles provided by the application server
  – An about page with links to useful information and resources for a product
  – A pointer to another console module that is the lead application for the product
  – Prerequisite applications and console modules that should be already deployed
  – Portlets that are already deployed on the server that should be reused for the console module
  – Multiple entities of a single portlet definition that can be used on different pages
  – External URLs that can be launched from the console navigation
  – Row and column layout of a page
  – Windows that contain portlets on the page
  – Empty windows on a page that are filled at runtime depending on the user's navigation selection
  – The order and nesting of navigation nodes used by the console module
  – Parent navigation nodes under which the navigation tree for the current console module are placed
  – Navigation nodes from other products that should be included under the current console module's navigation

- An application server

  Provides a complete J2EE runtime environment for console modules.

- APIs for console modules

  Supports the following actions:

  – Passing properties between portlets
  – Launching a page from a portlet
  – Invoking Eclipse-based help

- Eclipse Help

  Eclipse Help is provided as a Web application running on the same server as Integrated Solutions Console. Eclipse Help serves help documents for a single module as well as for the entire console.

What is new in Integrated Solutions Console

If you have developed console modules for previous, stand-alone releases of Integrated Solutions Console, there are some changes in this release.

- Integration with WebSphere Application Server

  This is the first release in which Integrated Solutions Console is delivered as part of WebSphere Application Server, enabling other products to integrate their administrative interfaces into the application server console. The console was previously made available to business partners only through the Autonomic Computing Toolkit.

- Portal application archive

  A new model for packaging and deploying console modules. The portal application archive (PAA) is a standard portlet application with the addition of two XML files that describe the topology and security characteristics of the module. The portal application archive replaces the former method of packaging a deploying console modules as portlet WAR files with an additional component.xml descriptor.

- Migration

There is no supported migration path for console modules written to previous versions of Integrated Solutions Console.

- Changes to APIs provided for Integrated Solutions Console.

  Page launching
  A subset of the Dynamic UI Manager APIs is provided for launching pages and passing properties to portlets on the same or other pages. Launching a module on an existing page is not supported for this release.

  Form persistence
  The FormUtils class is not provided for this release.

  Eclipse-based help
  The EclipseHelp class replaces the ISCHelp class to launch help from the Eclipse help server, which is package and deployed with Integrated Solutions Console. Help is available from the console and from each console module that supports help. Page and message help is not available in this release.

  Obtaining user and group information
  The PUMA SPI was provided in V6.0 to obtain information about the user and the user's group. This API is not available for this release.

  External launching
  Launching the console to a specific page or portlet is not supported in this release.

  Product filtering
  The console allows the user to display tasks for a specific product.

  External URLs
  Console modules can add links to external Web sites to the console navigation.

Console module samples

The sample console modules provide examples of portal application archives and how to use the APIs and other features. These samples are available from the *Samples for WebSphere Application Server* page at the following location

http://www.ibm.com/developerworks/websphere/library/samples/index.html

These samples help understand the structure of a console module WAR package and the portal application descriptors that distinguish the application as a console module. After downloading the samples, follow the instructions in "Deploying a console module" on page 41. The following table lists the samples that are ready for deployment.

Sample	Description
Page layout (pagelayout.war)	Demonstrates how to specify the layout of console modules on a page using the elements of the portal topology descriptor. This sample shows how the lead application of a product provides an about page, which contains helpful information and resources about the product. The about page can be accessed from the console Welcome page. This module also demonstrates how to reuse a portlet from another module (Tree merge).

Sample	Description
Tree merge (TreeMerge.war)	Demonstrates how a subordinate application deploys into the lead application for a product. The <PAA-ref/> element indicates the page layout sample as the lead application. Also, the <requires/> element indicates the page layout sample as a prerequisite that must already be deployed. The <parent-tree/> element merges the navigation nodes for the tree merge sample into the page layout navigation tree.
Portlet context (PortletContext.war)	Demonstrates how to use the Dynamic UI Manager APIs to launch a page and how to pass properties to other console modules. The module receiving these properties can be on the same page or on another page.
Modes (Modes.war)	Demonstrates how to implement the Edit and Help modes in a module and how to invoke Eclipse-based help.
Basic module (Basicmodule.war)	Demonstrates how to implement basic module components.

Setting up the development environment

This topic describes how to setup your development and testing tools to facilitate the development of console modules.

IBM provides the following tool that enables you to quickly develop, test, and deploy your portlet applications.

- WebSphere Application Server Toolkit, version 6.1

You can also add the Integrated Solutions Console in WAS Portal Application Archive (PAA) Visual Designer plugin, available for download at the Integrated Solutions Console in WAS PAA Visual Designer download site. For additional information, see www.ibm.com/support.

Use the plugin to develop the topology and security descriptors necessary to convert portlet applications into console modules that you may deploy into an Integrated Solutions Console in WAS console. The PAA Visual Designer is supported for use with Application Server Toolkit, version 6.1. Follow these steps to add the PAA Visual Designer plugin to Application Server Toolkit.

1. Confirm that Application Server Toolkit is not running.
2. Download PAA Visual Designer.
3. Extract the download package to the /eclipse directory in the Application Server Toolkit.
4. Start Application Server Toolkit.

After installing the PAA Visual Designer plugin, you can add the portal topology and security descriptors to your JSR 168 portlet application projects. See online documentation for details.

As you test your console modules, you might discover that changes that you make to the JSPs are not rendered when you log in to the application server. If so, you probably need to check the **reloadingEnabled** attribute in the module's extended deployment descriptor (ibm-web-ext.xmi) Make sure this value is set to `true` to enable the JSP to reload each time you deploy a new change. See the ″Hot deployment and dynamic reloading″ topic in the *Developing and deploying applications* PDF for more information.

Developing your first console module

This topic is for developers new to console module development.

It is assumed that you already have a development environment, like Application Server Toolkit, already prepared. The topics in this section take you through the process of creating a simple console module. The sample console module consists of a single portlet which is deployed to a single page as part of the set of sample console modules provided by IBM. Be sure you have successfully deployed the sample console modules before starting.

Developing the portlet

This topic is intended for developers who are unfamiliar with portlet development. A simple Java class and JSP are provided.

Developing the source Java class file

The following example shows the Java code for a portlet in its simplest form.

```
package com.ibm.isclite.samples.basicmodule;

import java.io.*;
import javax.portlet.*;

public class BasicModule extends GenericPortlet {

 public void doView(RenderRequest request, RenderResponse response)
              throws PortletException, IOException {
  // Set the MIME type for the render response
  response.setContentType("text/html");

        // Invoke the JSP to render
  PortletRequestDispatcher rd = getPortletContext().getRequestDispatcher("/jsp/basicView.jsp");
  rd.include(request,response);
 }

}
```

The portlet code must extend the GenericPortlet class and output for the response in the doView() method. Portlets are rendered in different modes. The initial mode when a portlet is called to render is the view mode. The response output is provided by a JSP, which provides markup that can be aggregated into a larger HTML page. The package name in this example is consistent with the console module samples.

Developing the JSP

The following shows a portlet JSP for view mode in its simplest form.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 pageEncoding="ISO-8859-1" session="false" buffer="none"%>
<%@ page import="javax.portlet.*" %>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portletAPI" %>

<portletAPI:defineObjects />

<%
  PortletPreferences prefs = renderRequest.getPreferences();
  String URL = prefs.getValue("website","");
%>

<p><a name="<portletAPI:namespace/>basicAnchor">Basic contents</a></p>

<p>
  <a href="<%=URL%>" target="_blank">
    <img src='<%=renderResponse.encodeURL(renderRequest.getContextPath() + "/images/logo.gif")%>'
         alt="logo" />
      </a>
</p>
<p><em>Company logo with link.</em></p>
```

The taglib directive specifies the portlet tags from the Java Portlet Specification. The following tags are used in this example.

namespace
Uniquely qualifies named HTML tags or JavaScript functions to prevent clashes with other portlets on the page using the same name.

defineObjects
Makes the RenderRequest, RenderResponse, and PortletConfig objects available to the JSP. This tag is used in this example so that the image can be rendered using the encodeURL() method of the render response, and to allow the JSP to read preferences in the portlet descriptor.

Creating the descriptors for the console module

The following descriptors are required for the console module.

- Web application descriptor (web.xml). This descriptor is required and described by the Java Servlet Specification.
- Portlet application descriptor (portlet.xml). This descriptor is required and described by the Java Portlet Specification. Refer to the console module samples for examples of how to create the portlet.xml.
- Portal topology descriptor (ibm-portal-topology.xml). This descriptor is unique to console modules and defines the portlets and resources in the module, how it is laid out on the page, and how it is accessed from the navigation.
- Portal security descriptor (ibm-portal-security.xml). This descriptor is unique to console modules and defines roles and access to navigation nodes and portlets.

The topology and security descriptors are the main component of PAA packages that distinguish them from other portal application WAR files. The console module samples include these descriptors and demonstrate their features. The sample descriptors provide inline comments that explain the elements and how they are used.

For your first console module, you should use the PAA Visual Designer plugin to the Application Server Toolkit to develop the descriptors. If you prefer to create them manually to become familiar with the structure of the XML, the steps in this topic show how to create simple descriptors that can be used to deploy an application into the samples provided by IBM. After following these steps, you should refer to the samples and the reference topics for the PAA schemas for more complete information.

Portal topology descriptor

To follow these steps, open a file with the name ibm-portal-topology.xml in a text or XML editor and save each change at the end of each step.

1. Copy the following elements into ibm-portal-topology.xml. Some information is split on multiple lines for printing purposes.

```
<?xml version="1.0" encoding="UTF-8"?>

<ibm-portal-topology
    xmlns="http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-topology.xsd"
    xmlns:base="http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-base.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-topology.xsd
ibm-portal-topology.xsd">

   <application-definition appID="com.ibm.isclite.samples.basicModule" version="6.1">


   </application-definition>
   <PAA-ref>com.ibm.isclite.samples.PageLayout</PAA-ref>

</ibm-portal-topology>
```

- The elements in this step provide the top level content of the topology descriptor.
- Most of the content for this descriptor will be placed inside the <application-definition/> element.
- The value of the **appID** attribute is consistent with the naming convention used by the sample console modules. The **version** attribute matches the version used by the sample console modules.
- The <PAA-ref/> element specifies the **appID** of the page layout sample, which is the lead application to which this application will be deployed.
- The following convention is used for unique name for all of the elements in the descriptor.

  ```
  namespace + element_type + identifier
  ```

2. Create a title for your application. The title is the first element in the application definition element.

   ```
   <title>
         <base:nls-string lang="en">My basic module</base:nls-string>
   </title>
   ```

3. Place the component tree after the <requires/> element as shown.

   ```
   <component-tree uniqueName="com.ibm.isclite.samples.basicModule.appTree">
      <portlet-definition uniqueName="com.ibm.isclite.samples.basicModule.portletDefinition.A">
         <title>
            <base:nls-string lang="en">My basic module</base:nls-string>
         </title>
         <resource-link name="basicModule"
                        portletApplication="com.ibm.isclite.samples.basicmodule.BasicModule.01a"
                        type="static"/>
      </portlet-definition>
      <portlet-entity   uniqueName="com.ibm.isclite.samples.basicModule.portletEntity.A"
                        portletDefinitionRef="com.ibm.isclite.samples.basicModule.portletDefinition.A">
         <title>
            <base:nls-string lang="en">My basic module</base:nls-string>
         </title>
      </portlet-entity>
   </component-tree>
   ```

   - The **name** attribute of the <resource-link/> element must specify the <portlet-name/> from the portlet deployment descriptor.
   - The **portletApplication** attribute of the <resource-link/> element must specify the **id** attribute of the <portlet-app/> element from the portlet deployment descriptor.
   - The **portletDefinitionRef** attribute of the <portlet-entity/> element references the unique name of the <portlet-definition/>. Subsequent elements in this descriptor use the unique name of the <portlet-entity/> to invoke the portlet. Multiple portlet entities can point to the same portlet definition. For the purpose of this exercise, a single portlet definition is provided with a single portlet entity.
   - The portlet title that is rendered in this example is derived from the title in the portlet entity. If that value is not provided, the portlet title is rendered by the title provided by the <window/> element in the layout tree. If that value is also not provided, the title is obtained from the <portlet-name/> element in the portlet.xml for the portlet indicated by the <resource-link/>.

4. Place the layout tree after the component tree as shown.

   ```
   <layout-tree>
      <layout-element uniqueName="com.ibm.isclite.samples.basicModule.layoutElement.A">
         <title>
            <base:nls-string lang="en">My basic module</base:nls-string>
         </title>
         <simple-container orientation="row"
                           uniqueName="com.ibm.isclite.samples.basicModule.container.A">
            <window uniqueName="com.ibm.isclite.samples.basicModule.window.A">
               <title>
                  <base:nls-string lang="en">My basic module</base:nls-string>
               </title>
               <component-definition-ref>
                  com.ibm.isclite.samples.basicModule.portletEntity.A
               </component-definition-ref>
   ```

```
        </window>
      </simple-container>
   </layout-element>
</layout-tree>
```

- Each layout element in the layout tree defines the layout of a page in the console.
- Each page can have nested containers, and each simple container can have multiple windows. Each simple container has either a row or column orientation. The nesting of these containers can create a complex table structure for the page layout.
- The <component-definition-ref/> element specifies the unique name of the portlet entity that provides the content of this window.

5. Create the navigation element after the layout tree as shown.

```
<navigation-element uniqueName="com.ibm.isclite.samples.navigationElement.basicModule"
              layoutElementRef="com.ibm.isclite.samples.basicModule.layoutElement.A">
   <title>
      <base:nls-string lang="en">Basic Module Sample</base:nls-string>
   </title>
   <preference name="ProductFilter">
      <base:value>
         <base:string>com.ibm.isclite.samples.PageLayout</base:string>
      </base:value>
   </preference>
   <parent-tree parentTreeRef="com.ibm.isclite.samples.navigationElement.Samples"/>
</navigation-element>
```

- Each navigation element defines a node in the console navigation. If you nest the navigation elements, it creates a hierarchical tree structure that is reflected in the console navigation. For the purpose of this exercise, only a single navigation element is provided. The **layoutElementRef** attribute indicates the unique name of the layout element that is rendered on the page that is displayed when this element is selected by the console user.
- The content of the `ProductFilter` preference indicates the **appID** of the lead application. This is used to render this navigation element when the lead product is selected by the console user for filtering the navigation.
- The content of the **parentTreeRef** element indicates the unique name of the parent navigation element for this element.

The following figure shows how this navigation element is rendered with the page layout sample. In this figure, the user has selected `Integrated Solutions Console Sample` from the **View** selection list to filter the navigation.

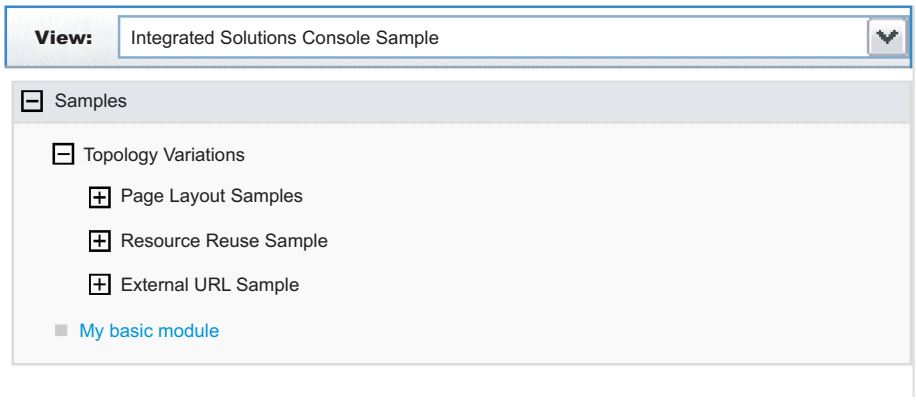


Figure 1. Sample console module navigation

When you have finished, save and close the file and store in the /WEB-INF directory of your console module project.

Sample portal security descriptor

To follow these steps, open a file with the name ibm-portal-security.xml in a text or XML editor and save each change at the end of each step.

1. Copy the following elements into ibm-portal-security.xml. Some information is split on multiple lines for printing purposes.

```
<?xml version="1.0" encoding="UTF-8"?>

<ibm-portal-security
     xmlns="http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-security.xsd"
     xmlns:base="http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-base.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-topology.xsd
ibm-portal-topology.xsd
         http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-base.xsd ibm-portal-base.xsd
         http://www.ibm.com/websphere/appserver/schemas/6.0/ibm-portal-security.xsd ibm-portal-security.xsd">

  </ibm-portal-security>
```

2. Add the <application-role/> element as the content of the <ibm-portal-security/> element.

```
<application-role uniqueName="monitor">


</application-role>
```

   This sample identifies the administrative role, `monitor`, for access rights. See “Portal security schema” on page 62 for a complete description of this element and the uniqueName attribute.

3. Add the following <portal-role/> elements as the content of the application role.

```
<portal-role object-ref="com.ibm.isclite.samples.navigationElement.basicModule"/>
<portal-role object-ref="com.ibm.isclite.samples.basicModule.navigationElement.A"/>
<portal-role object-ref="com.ibm.isclite.samples.basicModule.portletEntity.A"
             role-type="Privileged User"/>
```

   This sample assigns permission to a user with monitor access to view two navigation elements and to view, edit, and access help for one portlet. See “Portal security schema” on page 62 for a complete description of this element and its attributes.

When you have finished, save and close the file and store in the /WEB-INF directory of your console module project.

Packaging a console module

Before you start this task, be sure your Java sources compiles without errors and the XML descriptors have been validated against the corresponding schemas.

An Integrated Solutions Console module must be packaged as a WAR file for deployment. For best results, use a comprehensive development tool such as WebSphere Application Server Toolkit or Rational Application Developer. Otherwise, you can follow these steps to package your console module resources into the WAR file.

1. Create a working directory to collect all of the files that need to be packaged in the WAR file.
2. Organize your portlets and other resource files in the prescribed directory structure for portlet WAR files using the following steps:
   a. Place the following deployment descriptors in the directory /WEB-INF directory:
      - ibm-portal-topology.xml (topology descriptor)
      - ibm-portal-security.xml (security descriptor)
      - portlet.xml (portlet descriptor)
      - web.xml (Web application descriptor)

b. Place individual Java classes for your portlets in the directory /WEB-INF/classes. Use a directory structure that reflects the fully-qualified class name of your portlet, for example, /com/ibm/isclite/samples/basicModule.

c. Put JAR files in the directory /WEB-INF/lib.

d. If the console module supports multiple languages, place the properties file that contains the translated strings (the resource bundles) in the directory /WEB-INF/classes/nls.

e. Put any JavaServer Pages that must be served directly by the client in a directory that is not under the /WEB-INF directory (such /jsp). Resources in the /WEB-INF directory are protected and are not URL-addressable. JavaServer Pages that are accessed only by the portlets in the console module can be placed in the /WEB-INF path.

f. If the console module has help files, copy the Eclipse help plug-in files to the directory /help/*plugin_name*, where *plugin_name* is the name of the Eclipse help plug-in, such as com.ibm.myprod.help.doc.

3. Use the Java JAR utility to create the WAR file. For instructions, see the Java Tutorial at http://java.sun.com/docs/books/tutorial/jar/basics/index.html . You can also use an assembly tool like Application Server Toolkit or Rational Application Developer to create the WAR file for the console module.

Deploying a console module

After you create the WAR file for a console module, you can deploy the console module to a system where the runtime for Integrated Solutions Console is installed and then test that the console module works correctly. Only one instance of a console module can be deployed on an Integrated Solutions Console installation. This rule applies even if the versions of the console module are different.

If you need to update the version of a console module, you must perform these steps.

1. Remove the console module.
2. Update the module with the required changes.
3. Reissue the deploy command.

To deploy a console module on a system where the runtime for Integrated Solutions Console is installed, perform the following procedure:

1. Go to the machine where the Integrated Solutions Console runtime is installed. You can also use a machine that has remote access to the drive where the runtime is installed.
2. Copy the console module WAR file to the *app_server_root*/systemApps/isclite.ear directory. As an alternative, you can extract the WAR file into a subdirectory of this location using the WAR file name for the subdirectory. For example, you would extract pagelayout.war to the *app_server_root*/systemApps/isclite.ear/pagelayout.war directory.
3. Open a command prompt.
4. If you want to deploy the console module to an application server profile other than the default, change to the *app_server_root*/profiles/*profile_name*/bin directory and enter the setupCmdLine.[bat|sh] command. If you want to deploy to the default application server profile, change to the *app_server_root*/bin directory.
5. Start the wsadmin scripting client. If the server is not running, use the -conntype NONE option to run in local mode.
6. At the **wsadmin** command prompt, enter the command syntax. The following example presents the syntax:

```
$AdminApp update isclite modulefile {
-operation add
-contents path_to_war
```

```
-contenturi war_filename
-custom paavalidation=true  -usedefaultbindings
-contextroot /context
-MapWebModToVH {{.* .* admin_host}}}
```

Note: The values of the -contents and -contenturi are case sensitive and should match the war file name that you are ready to deploy. The option custom paavalidation=true is optional.

As an example, if you were to deploy the page layout sample to a Windows machine, the command would be as follows.

```
$AdminApp update isclite modulefile {
-operation add
-contents C://WebSphere//AppServer//systemApps//isclite.ear//pagelayout.war
-contenturi pagelayout.war
-custom paavalidation=true  -usedefaultbindings
-contextroot /pagelayout
-MapWebModToVH {{.* .* admin_host}}}
```

The following messages are displayed if the deployment is successful.

```
Update of isclite has started.
ADMA5009I: An application archive is extracted at D:\IBM\WebSphere_63\AppServer\
profiles\AppSrv01\wstemp\wstemp\app_10793d84b8e\ext/pagelayout.war
ADMA5064I: FileMergeTask completed successfully for isclite.
ADMA5005I: The application isclite is configured in the WebSphere Application Se
rver repository.
ADMA5005I: The application isclite is configured in the WebSphere Application Se
rver repository.
ADMA5110I: The application isclite is installed as a hidden application and will
 not be exposed via administrative interfaces such as GUI client, wsadmin or MBe
an Java API.  In order to perform management operations on this application, the
 application name must be known.
CWLAA10001I: The Integrated Solutions Console module was deployed successfully.
ADMA5005I: The application isclite is configured in the WebSphere Application Se
rver repository.
ADMA5011I: The cleanup of the temp directory for application isclite is complete
.
Update of isclite has ended.
```

7. After the deployment has completed successfully, save your work by issuing the following command:

```
$AdminConfig save
```

If the deployment is successful, the WAR file contents are extracted to the /systemApps/isclite.war directory and the module metadata is copied to the *app_server_root*/profiles/*profilename*/config/cells/*cellname*/applications/isclite.ear/deployments/isclite directory.

To view your console module, you must log back in to the console. Make sure the navigation is updated correctly. If so, select the new navigation nodes verify that the new pages and console modules are rendered correctly.

Testing a console module

Use the following checklist to test your console module after it has been deployed.

1. Log in to Integrated Solutions Console as an administrator.
2. If the module includes an <about-page/> element in the topology descriptor, verify that your suite and its version are listed in the table on the Welcome page. Also confirm that the product is listed in the **View** selection list for the navigation,
3. Verify that the console navigation tree contains all of the organizational nodes and pages that were defined. Verify that each node is at the correct level in the tree.
4. For each page, verify that the portlets on the page work correctly. If a portlet on the page launches a new page in work area, verify that the launching works correctly.

5. To test the access permissions specified in the console module descriptor, logout and then log in as a different user to verify that the correct pages are accessible.
6. If the console module included a help plug-in, make sure you can launch the help.
7. If the console module contains errors, refer to the SystemOut.log and SystemErr.log for any errors or messages. These logs are located in the *app_server_root*/profiles/*profilename*/logs/server1 directory.

Removing a console module

Follow these steps to delete a console module.

1. Ensure that the server is started. You do not need to log in to the console.
2. Open a command prompt.
3. If the console module is deployed to an application server profile other than the default, change to the *app_server_root*/profiles/*profile_name*/bin directory and enter the setupCmdLine.[bat|sh] command. Otherwise, change to the *app_server_root*/bin directory.
4. Start the wsadmin scripting client.
5. Enter the following command from the wsadmin command prompt.

   ```
   $AdminApp update isclite modulefile {-operation delete -contenturi module_file_name.war}
   ```

6. Save your configuration changes.

   ```
   $AdminConfig save
   ```

7. Exit the wsadmin command environment.

   ```
   quit
   ```

Adding advanced API features

The class files for console modules are developed using the Java Portlet Specificaiton. Integrated Solutions Console includes additional APIs for launching pages, passing properties to other portlets, and launching Eclipse-based help. This section provides information about the APIs available to console modules.

Developing portlets

Integrated Solutions Console supports console modules written to the Java Portlet Specification. A packaged console module WAR file contains one or more portlets or reuses portlets from other modules. According to the JSR 168 specification, a *portlet* is a Java technology based Web component, managed by a portlet container, that processes requests and generates content that contributes to the overall portal page. Portlets share many of the same characteristics as servlets. However, portlets also exhibit differences. For example, portlets cannot set HTTP headers and can contribute only markup fragments to the portal page. The portlet specification leverages the functionality of the servlet specification as much as possible, and many parts of the Java Portlet Specification have been modeled after the servlet specification.

The application server includes a JSR 168-compliant portlet container to support deploying and running console modules in Integrated Solutions Console. The JSR 168 portlet container relies on the J2EE architecture implemented by the application server. Consequently, an Integrated Solutions Console module and its portlets are packaged and deployed similar to J2EE Web applications. In addition to the Web application descriptor and the portlet descriptor, the WAR file for an Integrated Solutions Console module includes descriptors for a portal application archive (PAA), which provides additional information needed to deploy the application to a console installation.

The following Web resources can useful for learning more about portlet development.

Resource	Web site URL
Portlet Specification 1.0	http://jcp.org/jsr/detail/168.jsp

Resource	Web site URL
Portlet Specification 2.0	http://jcp.org/en/jsr/detail?id=286
WebSphere Portal Zone	http://www.ibm.com/developerworks/websphere/zones/portal/
Best practices: Developing portlets using JSR 168 and WebSphere Portal	http://www.ibm.com/developerworks/websphere/library/techarticles/0403_hepper/0403_hepper.html
WebSphere Portal external newsgroup	news://news.software.ibm.com/ibm.software.websphere.portal-server
Develop a Faces JSR 168 portlet using IBM Rational Application Developer 6.0 for autonomic computing	http://www.ibm.com/developerworks/edu/ac-dw-ac-jsfisc-i.html
Introduction to JSR 168 - The Portlet Specification	http://developers.sun.com/prodtech/portalserver/reference/techart/jsr168/index.html

Launching pages

Typically, users of the console launch pages using the links in the console navigation. However, portlets in a console module also launch pages directly using the Dynamic UI Manager API.

The DynamicUI Manager APIs are provided by 3 services: PropertyFactory, DynamicUIManagerFactoryService, and URLGeneratorFactoryService. To use these services the portlet must first perform a JNDI lookup to get the PortletServiceHome object which you then can use to get the actual service interface object. This is demonstrated in the GetApps portlet of the PortletContext sample.. The following code is placed in a `try` block of the portlet's init() method.

Some information is split on multiple lines for printing purposes.

```
Context ctx = new InitialContext(com.ibm.portal.jndi.Constants.ENV);
// get propertyfactory service

propertyFactoryServiceHome = (PortletServiceHome)ctx.lookup("portletservice/com.ibm.portal.
propertybroker.service.PropertyBrokerService");
if (propertyFactoryServiceHome == null)
   logger.log(Level.FINE, "no PropertyBrokerService propertyFactoryServiceHome");
else
   propertyFactoryService =

      (PropertyFactory)propertyFactoryServiceHome.getPortletService(com.ibm.portal.
propertybroker.service.PropertyBrokerService.class);
logger.log(Level.FINE, "propertyBrokerService="+propertyFactoryService);
// get dynamicUIManagerFactory service
dynamicUIManagerFactoryServiceHome =
   (PortletServiceHome)ctx.lookup("portletservice/com.ibm.portal.portlet.service.DynamicUIManagerFactoryService");
if (dynamicUIManagerFactoryServiceHome == null)
   logger.log(Level.FINE, "no dynamicUIManagerFactoryService propertyFactoryServiceHome");
else
   dynamicUIManagerFactoryService =

      (DynamicUIManagerFactoryService)dynamicUIManagerFactoryServiceHome.getPortletService
(com.ibm.portal.portlet.service.DynamicUIManagerFactoryService.class);
logger.log(Level.FINE, "dynamicUIManagerFactoryService="+dynamicUIManagerFactoryService);
// get urlGeneratorFactory service
urlGeneratorFactoryServiceHome =
   (PortletServiceHome)ctx.lookup("portletservice/com.ibm.portal.portlet.service.URLGeneratorFactoryService");
if (urlGeneratorFactoryServiceHome == null)
   logger.log(Level.FINE, "no urlGeneratorFactoryService propertyFactoryServiceHome");
else
   urlGeneratorFactoryService =
```

```
      (URLGeneratorFactoryService) urlGeneratorFactoryServiceHome.getPortletService
(com.ibm.portal.portlet.service.URLGeneratorFactoryService.class);
logger.log(Level.FINE, "urlGeneratorFactoryService="+urlGeneratorFactoryService);
```

A portlet can launch a new page using the dynamicUICtrl interface, passing the object ID of the page definition for the new page. The object ID is determined by performing a JNDI lookup on the unique name of the navigation element that launches the page. If you are planning to launch a navigation element, the navigation node must point to a layout element. In other words the navigation element must contain the l**ayoutElementRef** attribute. The addPage() method returns the object ID of the page instance, which should be saved by the portlet to create a URL and launch the page. The following excerpt from the PortletContext sample demonstrates how this works. The portlet GetApps.java includes a launchPage() method with this code.

```
Context ctx;
ObjectID pageDefinitionID = null;
String pagename="com.ibm.isclite.portletcontext.navigationElement.B";
logger.log(Level.FINE, "launch page=" + pagename);

PortletSession ps  = request.getPortletSession(false);
try {
    ctx = new InitialContext();
    pageDefinitionID =(ObjectID)ctx.lookup("portal:uniquename/"+pagename);
} catch (NamingException ne) {
    return;
}
try {
    PropertyController cproperty = propertyFactoryService.createProperty(myconfig);
    PropertyController cproperty2 = propertyFactoryService.createProperty(myconfig);

    cproperty.setType("String");
    cproperty.setName("servername");
    cproperty2.setType("String");
    cproperty2.setName("appDeployed");
    PropertyValue[] propertyValues = new PropertyValue[2];
    propertyValues[0] =
        propertyFactoryService.createPropertyValue(request, cproperty, serverName);
    propertyValues[1] =
        propertyFactoryService.createPropertyValue(request, cproperty2, appDeployed);

    DynamicUICtrl dmanagerCtrl =
       dynamicUIManagerFactoryService.getDynamicUICtrl(request, response, "isc.tasklaunch");

    ObjectID newPageID =
        dmanagerCtrl.addPage(pageDefinitionID, null, propertyValues);

    RedirectURLGenerator urlGenerator =
        urlGeneratorFactoryService.getURLGenerator(request, response);

    EngineURL redirectURL = urlGenerator.createPageURL(newPageID);

    response.sendRedirect(redirectURL.toString());
} catch  ...
```

- The `pagename` String is created with the unique name of the navigation element (from the topology descriptor) that launches the page.
- The object ID is found by performing a JNDI lookup on this Context instance using this String:

  ```
  "portal:uniquename/"+pagename
  ```

  The String `portal:uniquename/` is required to obtain the object ID for the navigation element.
- Properties are created to be passed to the portlets on the page. The property names are `servername` and `appDeployed`.
- In addition to the request and response, you must include the String `isc.tasklaunch` as the configuration name on the getDynamicUICtrl() method.

- Use the addPage() method of the new DynamicUICtrl instance to create and instance of the page. Any properties that need to be passed to portlets on the new page must be provided in this method. The addPage() method does not launch the page, but it returns the object ID of the page instance. The next two methods, getURLGenerator() and createPageURL(), are used to construct the URL that opens the page. The URL contains property names and values if they have been provided.
- The URL to open the page is provided on the sendRedirect() method of the response.

Passing properties to other portlets

This topic describes how portlets can exchange properties. You should be familiar with the concepts for launching console pages before reading this topic.

Information can be provided to portlets on the same page using the addSharedPortlet() method. Property values are of the type com.ibm.portal.propertybroker.property.PropertyValue. The example in “Launching pages” on page 44 shows these properties passed to the page using an instance of the PropertyValue interface (`propertyValues`) and shows how that object was created using the PropertyFactory interface.

The SetServer.java source from the PortletContext sample shows how to pass properties to a portlet on the same page. It is assumed that the portlet has already performed a JNDI lookup to determine that the PropertyFactory, DynamicUIManagerFactoryService, and URLGeneratorFactoryService services are available. This is described in “Launching pages” on page 44

- In this example, the portlet that is the source of the properties needs the object ID of the target portlet. The object ID is determined by performing a JNDI lookup using the values from the portlet’s <resource-link/> element in the topology descriptor.
- The property, `cproperty`, is created using the PropertyController interface and the createProperty() method of the PropertyFactory interface. The data type for the property must be a String.
- Prior to sending the property, the source portlet creates an instance of the DynamicUICtrl interface, passing the string ″isc.tasklaunch″ as the configuration name.
- The addSharedPortlet() method sends the properties to the target portlet on the request. After the action phase, the target portlet updates the output for the response.

```
Context ctx = null;
ObjectID portletDefinitionID1 = null;
ObjectID portletDefinitionID2 = null;

String portletname1="com.ibm.isclite.samples.PortletContext/PortletGetPerformance";
String portletname2="com.ibm.isclite.samples.PortletContext/PortletGetApps";

PortletSession ps  = request.getPortletSession(false);

try
   {
      ctx = new InitialContext(com.ibm.portal.jndi.Constants.ENV);
      portletDefinitionID1 =
         (ObjectID)ctx.lookup("portal:config/portletdefinition/"+portletname1);
      portletDefinitionID2 =
         (ObjectID)ctx.lookup("portal:config/portletdefinition/"+portletname2);
   } catch (NamingException ne)
     {
        logger.log(Level.FINE, "portletdefinitionID not found - Naming exception:"+ne.getMessage());
        return;
     }
logger.log(Level.FINE, "portletdefinitionID="+portletDefinitionID1.toString());

try
{
   PropertyController cproperty =
      propertyFactoryService.createProperty(myconfig);
   cproperty.setType("String");

   PropertyValue[] propertyValues = new PropertyValue[1];
```

```
   propertyValues[0] =
      propertyFactoryService.createPropertyValue(request, cproperty, serverName);

   DynamicUICtrl dmanagerCtrl =
      dynamicUIManagerFactoryService.getDynamicUICtrl(request, response, "isc.tasklaunch");

   ObjectID newPortletID1 =
      dmanagerCtrl.addSharedPortlet(portletDefinitionID1, null, propertyValues);
   ObjectID newPortletID2 =
      dmanagerCtrl.addSharedPortlet(portletDefinitionID2, null, propertyValues);

      logger.log(Level.FINE, "portlet ID created:"+newPortletID1.getOID());
    } catch ...
```

In some cases, the target portlet might be on a separate page. In this case, the properties are passed using the addPage() method. The target portlet receives the properties only when the page is launched that contains the portlet. If a property value is set multiple times before the page is launched, the value that was set last for the property is passed to the portlets on the page. You can use the URLGeneratorFactoryService interface to redirect the console to the page for the target portlet. This is demonstrated in the GetApps portlet of the PortletContext sample.

```
ObjectID newPageID = dmanagerCtrl.addPage(pageDefinitionID, null, propertyValues);
RedirectURLGenerator urlGenerator = urlGeneratorFactoryService.getURLGenerator(request, response);
EngineURL redirectURL = urlGenerator.createPageURL(newPageID);
response.sendRedirect(redirectURL.toString());
```

To use the URLGeneratorFactoryService, the portlet must first perform a JNDI lookup for this portlet service. Refer to the GetApps.java source to see how to obtain a reference to the portlet service.

Receiving properties

To receive properties, the target portlet must provide the `com.ibm.portal.pagecontext.enable` preference parameter in the portlet.xml with a value of true. If the portlet should receive any subsequent updates, the `com.ibm.portal.context.enable` read-only preference should also be set to true. Only String property types are supported and the context is passed as parameters of the action request. The following example shows how the GetApps portlet receives properties passed by the SetServer portlet.

```
public void processAction(ActionRequest request, ActionResponse response)
       throws PortletException, IOException {

        PortletSession ps = request.getPortletSession(true);
        appDeployed = request.getParameter("appDeployed");
        serverName=request.getParameter("servername");

        ps.setAttribute("servername",serverName);
        ps.setAttribute("appDeployed",appDeployed);
        launchPage(request, response);

    }
```

A target portlet can check `com.ibm.portal.action` parameter of the request during action processing to determine if any properties have been passed. If properties are being passed to the portlet, the value of this parameter is `com.ibm.portal.pagecontext.receive`. For example:

```
 String action = req.getParameter(com.ibm.portal.action.name);
 if (action!=null && action.equalsIgnoreCase("com.ibm.portal.pagecontext.receive")) {

   // code to get the properties as a parameter on the request

 }
```

Launching Eclipse-based help

If your Integrated Solutions Console module contains a portlet that requires a help page, follow the instructions in this topic to implement help for the portlet.

Sample

The Modes sample (modes.war) demonstrates how a console module can provide user assistance or help. This sample is described in “Console module samples” on page 34.

Enabling portlet help

To implement portlet help, you use the portlet's doHelp() method, the EclipseHelp class, and the help setting in the portlet.xml file. Inside the doHelp() method, link to the target help topic in your Eclipse help plug-in by using the portletHelp() method of the EclipseHelp class. By including a doHelp() method in your portlet and adding the help setting to the portlet.xml file, you enable the portlet help mode. When portlet help mode is enabled, the ? icon is displayed on the portlet title bar. When a user clicks the icon, a separate browser window opens and displays only that help topic.

Adding the doHelp method

To add the doHelp() method to your portlet, perform the following procedure:

1. In the portlet Java file, include the following import statement:

```
import com.ibm.portal.help.EclipseHelp;
```

2. Implement the doHelp() method in the portlet.

```
public void doHelp(RenderRequest request, RenderResponse response)
       throws PortletException, IOException {

}
```

3. Inside the method, create a String to contain the help topic. The syntax for specifying a help topic is *pluginID*/*file_path*/*file_name*, where *pluginID* is the Eclipse help plug-in ID as specified in its plugin.xml file, *file_path* is the sub-directory path (if any), and *file_name* is the filename and extension of the help content file.

```
public void doHelp(RenderRequest request, RenderResponse response)
       throws PortletException, IOException {

   String topic = "com.ibm.isc.help.modes/help_mode.html";
}
```

4. Call the method EclipseHelp.portletHelp() passing in the request, response, and topic as shown in the code snippet below.

```
public void doHelp(RenderRequest request, RenderResponse response)
       throws PortletException, IOException{

   String topic = "com.ibm.isc.help.modes/help_mode.html";
  EclipseHelp.portletHelp(request, response, topic);

}
```

Setting help mode in the portlet.xml file

In addition to implementing the doHelp() method, you must set the help mode for that portlet in the portlet.xml file. To set help mode in the portlet.xml file, perform the following steps:

1. Use a text editor to open the portlet deployment descriptor (the portlet.xml file). See topics under Developing portlets for more information about creating a portlet.xml file.
2. Inside the <portlet-app> element, locate the <portlet> element for the portlet that implements the doHelp() method.

3. Inside the <supports> sub-element, add a <portlet-mode> element like the one shown in the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
             version="1.0" id="com.ibm.isclite.samples.Modes"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
                                 http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
   <portlet id="Modes_Portlet" >
      <portlet-name>ModesPortlet</portlet-name>
      <display-name>Modes portlet1 (JSR 168)</display-name>
      <display-name xml:lang="en">Modes portlet1 (JSR 168)</display-name>
      <portlet-class>com.ibm.isclite.samples.Modes</portlet-class>
      <expiration-cache>0</expiration-cache>
      <supports>
         <mime-type>text/html</mime-type>
         <portlet-mode>VIEW</portlet-mode>
         <portlet-mode>EDIT</portlet-mode>
         <portlet-mode>HELP</portlet-mode>
      </supports>
      <supported-locale>en</supported-locale>
      <resource-bundle>nls.HTML</resource-bundle>
   </portlet>


</portlet-app>
```

Creating Eclipse help

Refer to http://help.eclipse.org for information about providing HTML help as an Eclipse plugin.

Support for bidirectional characters

Integrated Solutions Console supports bidirectional text. If your console module provides support for Hebrew, it must provide two identical resource bundles for the translated Strings. The file name for one resource bundle must be appended with the `_iw` suffix and the other resource bundle must be appended with the `_he` suffix. For example:

```
portlet_iw.properties
portlet_he.properties
```

Console module schemas

Descriptors for a console module

Console modules are packaged as *portal application archives* (PAA). A portal application archive is similar to a standard JSR 168 portlet WAR file - it includes one or more portlets plus the servlet and portlet descriptors (web.xml and portlet.xml) However, the portal application archive includes two additional descriptors:

ibm-portal-topology.xml
: Defines the console module and its components, page layout, and location in the navigation. Based on the schema definitions in ibm-portal-topology.xsd.

ibm-portal-security.xml
: Describes portal application security, such as application roles and access control. Based on the schema definitions in ibm-portal-security.xsd.

When a console module is deployed, Integrated Solutions Console reads these descriptors and uses that information to add the console module to the installation. The XML descriptor files must be well-formed and validate with the associated schemas. If the file is not valid, deployment ends and an error message is generated.

A copy of the schema files (with descriptive comments embedded) is in the *app_server_root*/properties/schemas directory. Not all elements and attributes in the schemas are supported, however. Only elements and attributes that are documented in the following sections are supported:

- "Base schema"
- "Portal topology schema"
- "Application definition elements" on page 52
- "Portal security schema" on page 62

Base schema

The topology and security schemas reference the base schema for portal application archives, ibm-portal-base.xsd. The following complex type element definitions from the base schema are used most often in the topology and security descriptors. These elements can be used within any other element in the descriptor that requires a localized string to be displayed, for example, in the title or description elements.

NlsString

specifies localized strings to be displayed. This tag is intended to be used during development before the application has been updated to support multiple languages through resource bundles. The xml:lang attribute enables a programmer to get started with the code without having to define a resource bundle right at the start. The value of the xml:lang attribute should adhere to RFC 1766 (http://www.ietf.org/rfc/rfc1766.txt). The default value of the xml:lang attribute is English(en).

```
<base:nls-string lang="locale">string</base:nls-string>
```

NlsRef

specifies localized strings that are maintained in a Java resource bundle. You must include the location of the resource bundle and the name of the key for lookup.

```
<base:nls-ref key="key_name" locationName="path_to_resource_bundle"/>
```

where *path_to_resource_bundle* specifies a path relative to the /WEB-INF directory of the module's WAR file. For example, if the resource bundle is located in /WEB-INF/classes/nls/myresources.properties, specify `/classes/nls/myresources`.

If the resource bundle cannot be located, the key name is returned. That is, if the property string that could not be found is intended to be displayed in the console, the key name would be displayed instead.

Preference

Used to describe name/value pairs used by the application. The preference name is specified as an attribute and the preference value is specified as a subelement.

```
<preference name="preference_name">
   <base:value>
      <base:string>preference_value</base:string>
   </base:value>
</preference>
```

The preference value can be specified as a string, NlsString, or NlsRef.

Portal topology schema

The portal topology descriptor distinguishes the console module from other portlet applications by providing additional information about the portlets in the application. Whereas the portlet.xml provides a unique identifier for each portlet, it's fully-qualified class name, and supported modes, the portal topology descriptor describes a console module in a generic sense, including a list of component portlets, the page layout, and navigation information. Portlets in the application can be packaged in the WAR file or already deployed from another archive. The only mapping between the descriptors is a reference from the portal

application's <resource-link/> element, using the portlet application ID and portlet name from the portlet descriptor.

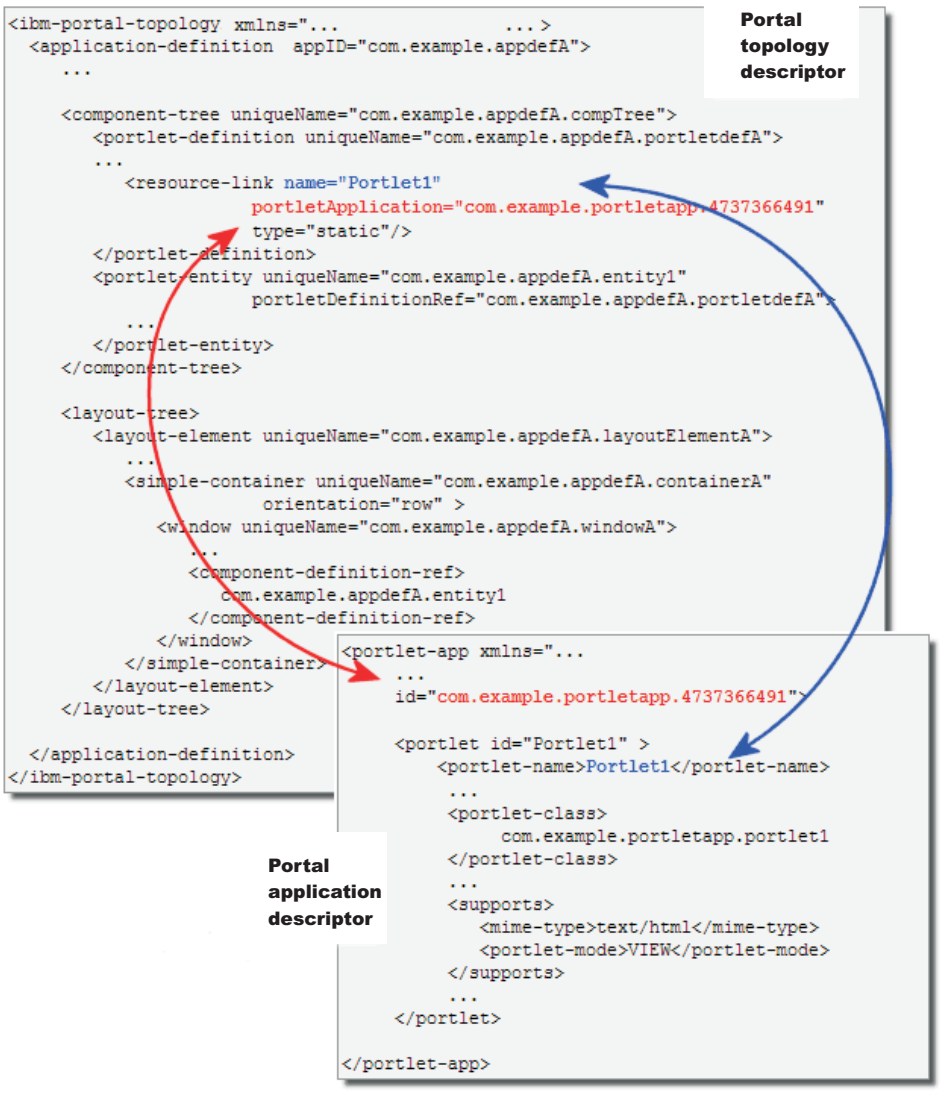


Figure 2. References from the portal topology descriptor to the portlet descriptor

Description of the top level topology elements

Only the following elements, along with those described in "Application definition elements" on page 52, are supported for this release.

<ibm-portal-topology/>
Required. the root element of the portal topology descriptor. The content of this tag is one or more <application-definition/> tags and, optionally, either an <about-page/> or a <PAA-ref/> element.

<application-definition/>
At least one is required. This element describes an application with its component portlets, page layout, and portal navigation structure. See Application definition elements.

After the application definition, only one of the following elements can be defined.

<about-page/>

Optional. The about page is provided by the lead application for a product and provides a description of the product and other resources for support. This content is specified using the unique name of the layout element in the application definition that contains the about page content. At runtime, the about page is accessed from the main welcome page of the console. Providing the <about-page/> element also causes the console module to appear in the View selection list of the navigation.

In the following example from the page layout sample, about page refers to a layout element with the unique name `com.ibm.isclite.pagelayout.layoutElement.F`.

```
<about-page>com.ibm.isclite.pagelayout.layoutElement.F</about-page>
```

The layout element provided in this sample includes a portlet with the about page content.

<PAA-ref/>

Optional. This element references the **appID** of the lead application for the product. The **appID** name should be unique in the install. This element is intended for subcomponents of an application that spans multiple archives.

Application definition elements

The <application-definition/> element describes the portal application, its components, page layout, and navigation. It is contained by the <ibm-portal-topology/> element in the portal topology descriptor.

Basic structure of the application definition

The following figure illustrates the elements of the application definition that are supported for this release.

```
<application-definition appID="" version="">

   <title/>

   <requires uniqueName="" version=""/>

   <component-tree uniqueName="">
      <portlet-definition uniqueName="">
         <title/>
         <resource-link name="" type="" portletApplication=""/>
      </portlet-definition>
      <portlet-entity uniqueName="" portletDefinitionRef="">
         <title/>
      </portlet-entity>
      <url-link uniqueName="" url="" absolute="" />
   </component-tree>

   <layout-tree>
      <layout-element uniqueName="">
         <title/>
         <simple-container uniqueName="" orientation="">
            <window uniqueName="">
               <title/>
               <component-definition-ref/>
            </window>
         </simple-container>
      </layout-element>
   </layout-tree>

   <navigation-element uniqueName="" layoutElementRef="">
      <title/>
      <preference>
         <value/>
      </preference>
      <url-mapping/>
      <parent-tree parentTreeRef="" ordinal=""/>
      <navigation-content uniqueName="" componentDefinitionRef="" windowRef=""/>
   </navigation-element>

</application-definition>
```

Figure 3. Elements in the portal application definition

Other elements of the application definition that can be found in the ibm-portal-topology.xsd schema are not supported for this release. At a minimum, the application definition requires a title, a component tree, and a layout tree. The following sections describe the <application-definition/> element, its attributes, and its subelements.

Application definition

The <application-definition/> element defines one or more applications that are part of the package. Exactly one is required.

The following attributes are used with this element.

appID Required. Specifies a unique identifier for this portal application. The appID has no relationship with IDs in the portlet.xml. To guarantee uniqueness, see “Guidelines for unique identifiers” on page 62.

version
Required. Specifies the numerical version of the application. The format of the version is defined in the ibm-portal-base.xsd schema under the simple type for VersionString.

Title

Exactly one is required. The <title/> element provides a title for the application. If this is the lead application for a product, this title is displayed on the Welcome page when the user logs into the console. A lead application is the one that includes the <about-page/> element in the topology definition. Subcomponents of the product use the <PAA-ref/> element to specify the lead application to which they belong.

The content of the <title/> element can be specified as a simple string element from the base schema. To provide a localized string, use the <base:nls-string/> or <base:nls-ref/> elements for the title content. For example:

```
<title>
  <base:nls-string lang="en">Integrated Solutions Console Sample</base:nls-string>
</title>
```

Requires

Optional, multiple allowed. The <requires/> element indicates the unique ID of a portal application that is a prerequisite for this application. Any prerequisites for the application must be already deployed. The application can have multiple prerequisites.

The following attributes are used with this element.

uniqueName
: Required. Specifies the unique identifier of another application that must already be deployed on the server. This value must match the appID attribute from the <application-definition/> element of the required application's topology descriptor. Otherwise, deployment fails.

version
: Required. Specifies the numerical version from the required application's <application-definition/> element. If the version is not identical to the required application's version string, deployment fails. Required.

The following example shows how the tree merge sample indicates the page layout sample as a prerequisite application.

```
<requires uniqueName="com.ibm.isclite.samples.PageLayout" version="6.1"/>
```

The following example shows the attributes for the <application-definition/> element of the page layout sample. These attributes are used as the values for attributes on the <requires/> element for the tree merge sample.

```
<application-definition appID="com.ibm.isclite.samples.PageLayout" version="6.1">
...
```

Component tree

Exactly one is required. The <component-tree/> element describes all portlets in the console module. The tree structure of the component definitions can be used for organizing the components in the portal application. There are no semantics associated with this organization and the portal may not respect this structure for organizing the actual portal resources corresponding to it.

The following attribute is used with this element.

uniqueName
: Required. Specifies a unique identifier for the component tree. To guarantee uniqueness, see "Guidelines for unique identifiers" on page 62.

The following elements make up the content of the <component-tree/> element.

<portlet-definition/>

Optional, multiple allowed. This element describes one portlet. The portlet can be included in the same archive or already deployed on the server.. A portlet definition refers to a portlet defined in a portlet.xml deployment descriptor in a portlet WAR file. A portlet can be associated with a window in a layout. In most cases, the application artifact is actually defined by its own deployment descriptor within the archive and the <resource-link/> element is used for referencing it so that it can be integrated into the portal application page layout and navigation.

The following attribute is used with this element.

uniqueName

Required. Specifies a unique identifier for the portlet definition. To guarantee uniqueness, see "Guidelines for unique identifiers" on page 62.

The following element makes up the content of the <portlet-definition/>.

<title/>

Required, exactly one allowed. This element provides a title for the portlet definition. This title is not displayed in the console for this release. To provide a localized string, use the <base:nls-string/> or <base:nls-ref/> elements for the title content.

<resource-link/>

Required, exactly one allowed. This element provides a link to a portlet definition that is included in the PAA package or is already deployed separately in the portal.

The following attributes are used with this element.

name Specifies the name of the portlet as defined by the <portlet-name/> element in the portlet.xml. If the name of the portlet is not unique within the PAA, an additional portlet-application attribute must be specified.

type The value `static` specifies that the resource must be available at the time the PAA package is deployed. This is the default and the only value that is supported for this release.

portletApplication

Specifies the ID attribute of the <portlet-app/> element in the portlet.xml. Since different PAA packages could have portlets with the same name, it is necessary to name space them.

The following example shows the <resource-link/> element for the page layout sample.

```
<resource-link name="PageLayoutPortlet1"
       portletApplication="com.ibm.isclite.samples.PageLayout" type="static"/>
```

The following example is taken from the portlet.xml for the page layout sample to show the values needed for the previous <resource-link/> example.

```
...
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
             version="1.0" id="com.ibm.isclite.samples.PageLayout"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
                                 http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
    <portlet id="PageLayout_Portlet1" >
       <portlet-name>PageLayoutPortlet1</portlet-name>
...
```

The next example is also from the page layout sample. In this example, the <resource-link/> uses values from a portlet that is already deployed to the portal. These values come from the portlet.xml of the portlet (TempPortlet) from the tree merge sample.

```
<portlet-definition uniqueName="com.ibm.isclite.TreeMerge.appElementDefinition.A">
   <title>
      <base:nls-string lang="en">TreeMergePortlet</base:nls-string>
   </title>
   <resource-link name="TempPortlet"
                  portletApplication="com.ibm.isclite.samples.TreeMerge" type="static"/>
</portlet-definition>
<portlet-entity uniqueName="com.ibm.isclite.TreeMerge.appElement.B"
                portletDefinitionRef="com.ibm.isclite.TreeMerge.appElementDefinition.A">
   <title>
      <base:nls-string lang="en">TempPortlet</base:nls-string>
   </title>
</portlet-entity>
```

<portlet-entity/>

Optional, multiple allowed. This element specifies an instance of a portlet definition with additional configuration values. The portlet entity's uniqueName is used by the <window/> element to place the portlet on the page. The portlet definition that is specified can reside in the current archive or already be deployed on the server.

The following attributes are used with this element.

uniqueName

Required. Specifies a unique identifier for the portlet entity. To guarantee uniqueness, see "Guidelines for unique identifiers" on page 62.

portletDefinitionRef

Required. References the unique name of the portlet definition within the same topology descriptor file.

The following element makes up the content of the portlet definition.

<title/>

Optional, no more than one allowed. This element provides a title for the portlet entity. To provide a localized string, use the <base:nls-string/> or <base:nls-ref/> elements for the title content. This title is rendered in the portlet window.

<url-link/>

Optional and multiple elements are allowed. This element is used to include links to external URLs that are added to the console navigation . The following attributes are used with this element.

uniqueName

Required. Specifies a unique identifier for the url link. To guarantee uniqueness, see "Guidelines for unique identifiers" on page 62.

absolute

Optional. Indicates whether the URL is an absolute value (true) or relative value (false). Only absolute values are supported for this release and the default value is true.

url Required. Specifies an external address or URL that will be added to the console navigation.

In the following example, the URL for a company support page is provided.

```
<url-link uniqueName="my.example.com.support.page" url="http://my.example.com/support" absolute="true"/>
```

The following example shows part of the <component-tree/> element for the page layout sample with the first portlet definition and corresponding portlet entity.

```
<component-tree uniqueName="com.ibm.isclite.pagelayout.appTree">

   <!-- PageLayoutPortlet 1-->
   <portlet-definition uniqueName="com.ibm.isclite.pagelayout.appElementDefinition.A">
      <title>
         <base:nls-string lang="en">RowPortlet</base:nls-string>
      </title>
```

```
      <resource-link name="PageLayoutPortlet1"
                     portletApplication="com.ibm.isclite.samples.PageLayout" type="static"/>
   </portlet-definition>
   <portlet-entity uniqueName="com.ibm.isclite.pagelayout.appElement.A"
                   portletDefinitionRef="com.ibm.isclite.pagelayout.appElementDefinition.A">
      <title>
         <base:nls-string lang="en">RowPortlet</base:nls-string>
      </title>
   </portlet-entity>

...
```

Layout tree

Exactly one is required. Each layout element describes the layout for a page that is opened by a navigation element. There is a one to one correspondence between a page and a layout element. The page layout is determined by row and column containers. Each row or column can contain any number of windows, which in turn contain one portlet each. The layout elements, containers, and windows can be fully defined, including their content, or they can be empty templates to be filled when content is added subsequently.

The following element makes up the content of the layout tree.

<layout-element/>
Optional and multiple allowed. The layout element describes the layout of a portal page. Provide one layout element for each page that you want to define in the application.

The following attribute is used with this element.

uniqueName
Required. Specifies a unique identifier for the layout element. To guarantee uniqueness, see “Guidelines for unique identifiers” on page 62.

The following elements make up the content of the layout element.

<title/>
Exactly one is required. This element provides a title for the layout element. To provide a localized string, use the <base:nls-string/> or <base:nls-ref/> elements for the title content. This title is displayed as the title for the corresponding page.

<simple-container/>
At least one is required. This element defines a row or column in the page layout. This element can be nested.

The following attributes are used with this element.

uniqueName
Required. Specifies a unique identifier for the layout element. To guarantee uniqueness, see “Guidelines for unique identifiers” on page 62.

orientation
Required. Specifies how the contents of the container are arranged. The value `row` arranges content from left to right. The value `column` arranges content from top to bottom.

The following element makes up the content of this element.

<window/>
Optional and multiple are allowed. A window describes a place in the page layout where portlet content can be inserted. The window can either contain an application artifact or a nested layout.

The following attribute is used for the window.

uniqueName
Required. Specifies a unique identifier for the window. To guarantee uniqueness, see “Guidelines for unique identifiers” on page 62.

The following elements make up the content of this element.

<title/>
Optional, no more than one allowed. To provide a localized string, use the <base:nls-string/> or <base:nls-ref/> elements for the title content. This string is rendered in the portlet title bar if the title for the <portlet-entity/> is not provided. If neither the portlet entity or the window provide a title, the title is obtained from the portlet.xml (<portlet-name/> element) of the portlet.

<component-definition-ref/>
Optional, but no more than one allowed. This element specifies a reference to the unique name of the component definition for the portlet to be included in this window.

The page layout sample contains six layout elements in its layout tree. This provides six pages showing different ways to arrange the content on the page. The following example shows the first layout element with its simple containers and windows. The inline comments are also shown.

```
<layout-tree>

   <!--PageLayout A contains Two Seperate Containers:
       The first Container has row orientation with one window inside the container
       The second Container has column orientation with one window inside the container

       This layout uses is-spot to specify the application for the window.
       Therefore you must specify the navigation-content in the navigation element
       below that uses this layout element.
    -->
    <layout-element uniqueName="com.ibm.isclite.pagelayout.layoutElement.A">
       <title>
          <base:nls-string lang="en">Two independent Containers</base:nls-string>
       </title>
       <simple-container orientation="row"
                         uniqueName="com.ibm.isclite.pagelayout.container.A1A">
          <simple-container orientation="row"
                            uniqueName="com.ibm.isclite.pagelayout.container.A1">
             <window uniqueName="com.ibm.isclite.pagelayout.window.A1">
                <title>
                   <base:nls-string lang="en">Row Portlet</base:nls-string>
                </title>
                <component-definition-ref>
                   com.ibm.isclite.pagelayout.appElement.A
                </component-definition-ref>
             </window>
          </simple-container>

          <simple-container orientation="column"
                            uniqueName="com.ibm.isclite.pagelayout.container.A2">
             <window uniqueName="com.ibm.isclite.pagelayout.window.A2">
                <title>
                   <base:nls-string lang="en">Column Portlet</base:nls-string>
                </title>
            </window>
        </simple-container>
       </simple-container>
    </layout-element>
```

Navigation element

Optional, and multiple are allowed. The <navigation-element/> element describes an element or *node* in the navigation tree. This element can be nested. Navigation elements also require unique names in the installation.

Navigation elements are arranged in a tree that reflects the structure of the navigation hierarchy that is created in the portal when the application is deployed. For example, a navigation tree with two primary nodes and two branch nodes each would exhibit the following structure:

```
<navigation-element>
   <navigation-element/>
   <navigation-element/>
</navigation-element>
<navigation-element>
   <navigation-element/>
   <navigation-element/>
</navigation-element>
```

In most cases, a navigation element references the layout element of the page that is displayed when the node is selected.

The following are the attributes of the <navigation-element/> element.

hidden

Optional. The default is false. When true, this attribute hides the navigation element from the console navigation tree. When this attribute hides a parent navigation element, it also hides any children. It is recommended that you associate a **parentTreeRef** attribute containing an empty string with each hidden navigation element. If you are planning to launch a navigation node, the navigation node must point to a layout element. In other words the navigation element must contain the **layoutElementRef** attribute.

uniqueName

Required. Specifies a unique identifier for the navigation element. To guarantee uniqueness, see “Guidelines for unique identifiers” on page 62.

You can also use this attribute to associate a navigation element from another product with this product. In this case, you must omit the title element from the navigation element. For example, if Product A has a navigation element titled ″Setting the background color″ that also works with Product B, then Product B would include a navigation element without a title that specifies the same unique name as this node from Product A. In addition, you can specify a wild card pattern to include multiple navigation elements from another products. The wild card is specified using two dashes (--). For example, the following pattern would include all of the navigation elements from a product that is already deployed and that have a unique name starting with `com.ibm`.

```
<navigation-element uniqueName="com.ibm.--">
</navigation-element>
```

layoutElementRef

Optional. Specifies the page to be displayed when the node is selected.

The following elements make up the content of the navigation element.

<title/>

Optional, no more than one allowed. This element provides a title for the node to be displayed in the navigation. This title is also displayed in the task tree for My Tasks, and in the View selection menu over the navigation . Because of the limited viewable width of the View selection menu, try to keep the character length of the title to a minimum. To provide a localized string, use the <base:nls-string/> or <base:nls-ref/> elements for the title content.

<parent-tree/>

Optional, no more than one allowed. Specifies the parent node and position under the parent node

where the current navigation element is to be located. The parent node need not be part of the current application definition. If a requested parent node does not exist, the node is appended to the bottom of the navigation tree and is merged into the parent when the parent application is deployed.

The following attributes are used with this element.

parentTreeRef
Required. Indicates the unique name of the navigation element under which this node should be placed.

ordinal
Optional. Provides a non negative integer to determine the relative order of nodes under a particular parent tree. If no ordinal is specified, or if two navigation elements specify the same ordinal for a navigation level, the node is located in the order in which it is deployed.

The following example from the tree merge sample shows how the <parent-tree/> element is used to merge the navigation elements into the navigation for the page layout sample. The parentTreeRef attribute specifies the unique name of the top level navigation element for the page layout sample. Child navigation elements are placed under the corresponding top level node for tree merge. Ordinal is not specified, so this navigation tree is appended to the page layout navigation tree.

```
<navigation-element uniqueName="com.ibm.isclite.samples.navigationElement.TreeMerge">

   ...
   <parent-tree parentTreeRef="com.ibm.isclite.samples.navigationElement.Topology"/>

   <navigation-element uniqueName="com.ibm.isclite.treemerge.navigationElement.A"
                       layoutElementRef="com.ibm.isclite.TreeMerge.layoutElement.A">
   ...
   </navigation-element>

   <navigation-element uniqueName="com.ibm.isclite.treemerge.navigationElement.B"
                       layoutElementRef="com.ibm.isclite.TreeMerge.layoutElement.B">

   ...
   </navigation-element>

</navigation-element>
```

<navigation-content/>
Optional, multiple allowed. This element defines the content of a window in a page layout. This can be used to supply the content for a window at runtime or to map different portlets to the same window, depending on which navigation element is selected.

The following attributes are used with this element.

uniqueName
Required. Specifies a unique identifier for the navigation content. To guarantee uniqueness, see “Guidelines for unique identifiers” on page 62.

componentDefinitionRef
Required. Indicates the unique name of the component definition that should be rendered in the window. The component definition can be a portlet entity or a url link. If the component definition specifies a url-link, omit the windowRef attribute. Windows cannot render external URLs.

windowRef
Optional. Indicates the unique name of the window where the component definition should be rendered. If this attribute is not specified, the component definition applies to the entire layout element. Multiple <navigation-content/> elements can specify the same window. Optional.

The following example from the page layout sample illustrates how the <navigation-content/> element can be used.

1. A portlet is defined in a component definition with the unique name `com.ibm.isclite.pagelayout.appElement.B` .

```
<portlet-entity uniqueName="com.ibm.isclite.pagelayout.appElement.B"
                portletDefinitionRef="com.ibm.isclite.pagelayout.appElementDefinition.B">
   <title>
      <base:nls-string lang="en">ColumnPortlet</base:nls-string>
   </title>
</portlet-entity>
```

2. A window is defined without a component definition reference, which means it has empty content.

```
<simple-container orientation="column"
                  uniqueName="com.ibm.isclite.pagelayout.container.A2">
   <window uniqueName="com.ibm.isclite.pagelayout.window.A2">
      <title>
         <base:nls-string lang="en">Column Portlet</base:nls-string>
      </title>
   </window>
</simple-container>
```

3. The navigation content specifies the unique name of the portlet and window for the componentDefinitionRef and windowRef attributes respectively.

```
<!--This Navigation Element is for Layout A -->
<navigation-element uniqueName="com.ibm.isclite.pagelayout.navigationElement.pagelayoutA"
                    layoutElementRef="com.ibm.isclite.pagelayout.layoutElement.A">
   <title>
      <base:nls-string lang="en">2 Separate Containers</base:nls-string>
   </title>
   <navigation-content uniqueName="com.ibm.isclite.pagelayout.navigationContent.A2"
                       componentDefinitionRef="com.ibm.isclite.pagelayout.appElement.B"
                       windowRef="com.ibm.isclite.pagelayout.window.A2"/>
</navigation-element>
```

<preference/>

Optional. Indicates the unique name of the application definition for a product. This is used to include the node in the navigation tree of the product when console users select the product title to filter the navigation. For example:

1. The page layout sample is deployed with the title `Integrated Solutions Console Sample` (as defined by the title tag for the application definition). This title is added to the product filter of the navigation **View** selection list.
2. The tree merge sample is subsequently deployed specifying page layout as a prerequisite application. To ensure that navigation nodes for tree merge are also displayed when the console user selects `Integrated Solutions Console Sample` for product filtering, the navigation elements includes this preference definition.

```
<preference name="ProductFilter">
   <base:value>
      <base:string>com.ibm.isclite.samples.PageLayout</base:string>
   </base:value>
</preference>
```

Note: The name attribute must be set to `ProductFilter`. The <base:string/> element must indicate the unique name of the application definition for the prerequisite product.

Organizing the navigation tree

The console navigation tree provides access to the pages for which the user has permission. The hierarchy of nodes and pages on the navigation tree is defined by the <navigation-element/> element in the portal topology descriptor. All pages accessible from the navigation tree must be defined within the content hierarchy.

Before you start authoring the <navigation-element/> section of the ibm-portal-topology.xml file, take time to plan the entries that you want to be displayed on the console navigation tree. Identify the pages that will be accessible from the console navigation tree. If possible, reuse existing primary organizational nodes and any appropriate sub-nodes in your console navigation tree.

Guidelines for unique identifiers

A console module consists of XML descriptors that must provide element identifiers that are unique throughout the application. This topic provides guidelines for setting the these identifiers to help ensure uniqueness. Some development tools, like WebSphere Application Server Toolkit, assign randomly-generated unique names for you. If you need to manage unique names yourself, follow these guidelines.

It is recommended that product or product suite teams define a naming convention for their console modules. For maximum effectiveness, the conventions should be defined before console modules are developed to allow time for refinement and avoid rework. The following convention provides one guideline that can help manage the unique names for an application.

```
namespace + module_name + element name + identifier
```

The identifier is necessary to distinguish different elements in the same descriptor. For example, the following unique names are used in the portal topology descriptor of the TreeMerge sample:

```
<component-tree         uniqueName="com.ibm.isclite.TreeMerge.appTree"                  ...
<portlet-definition     uniqueName="com.ibm.isclite.TreeMerge.appElementDefinition.A"   ...
<portlet-entity         uniqueName="com.ibm.isclite.TreeMerge.appElement.A"             ...
<portlet-definition     uniqueName="com.ibm.isclite.TreeMerge.appElementDefinition.B"   ...
<portlet-entity         uniqueName="com.ibm.isclite.TreeMerge.appElement.B"             ...
<layout-element         uniqueName="com.ibm.isclite.TreeMerge.layoutElement.A"          ...
<simple-container       uniqueName="com.ibm.isclite.TreeMerge.container.A"              ...
<window                 uniqueName="com.ibm.isclite.TreeMerge.window.A">                ...
<layout-element         uniqueName="com.ibm.isclite.TreeMerge.layoutElement.B"          ...
<simple-container       uniqueName="com.ibm.isclite.TreeMerge.container.B"              ...
<window                 uniqueName="com.ibm.isclite.TreeMerge.window.B"                 ...
<navigation-element     uniqueName="com.ibm.isclite.treemerge.navigationElement.A"      ...
<navigation-element     uniqueName="com.ibm.isclite.treemerge.navigationElement.B"      ...
<navigation-content     uniqueName="com.ibm.isclite.treemerge.navigationContent.B"      ...
```

Portal security schema

The portal security descriptor describes application roles for a portal application and maps those roles to actual users and groups of the console. The following example shows the ibm-portal-security.xml for the page layout sample.

Description of the portal security elements

<ibm-portal-security/>

Root element of the portal security descriptor. This element contains the following element.

<application-role/>

Optional and multiple allowed. Specifies an application role for the portal application. An application role maps a set of permissions, or role type, to a specific resource defined in the application's topology descriptor. The following attribute is used with this element.

uniqueName

Required. Specifies the administrative role for this application. The role can be one of the existing roles that are provided by the application server or you can specify a new role for the application that is created during deployment.

- When you specify one of the application server roles, make sure you use all lower case (for example, *administrator*, *operator*, *monitor*, or *configurator*) rather than the initial capitalization that appears in the console interface (*Administrator*, *Operator*, *Monitor*, or *Configurator*). If there is a mismatch in upper or lower casing (*MOnitor* instead of *monitor*), then the role cannot be created.

- When you specify a new role for an application, use lowercase characters as a guideline.
- The administrative console also provides the *all authenticated portal users* virtual role. Resources with *all authenticated portal users* permission are available to all authenticated users.
- When you specify a new role, it inherits the access rights of the *monitor* role when it is created. A user with the new administrative role has access to the same resources as a user with the *monitor* role.
- When you specify a resource in the topology descriptor without protecting it with an administrative role or virtual role, it becomes unavailable to any role.

To guarantee uniqueness, see "Guidelines for unique identifiers" on page 62.

The following element makes up the content of the application role.

<portal-role/>

Optional and multiple allowed. Associates a resource with a role type. The role type determines what actions a user can perform on that resource. The following attributes can be used with this element.

object-ref

Specifies the unique name of a resource in the topology descriptor. For this release, only portlet entities and navigation elements are supported.

role-type

Specifies the set of actions the user with this role type can perform on the resource. For this release, role-type has meaning only for portlet entities. The following values are allowed for this release (case-sensitive):

- **User**

  A user with this role type is allowed to view the resource and access help.
- **Privileged User**

  A user with this role type is allowed to view the resource, edit preferences, and access help.

These role mappings cannot be changed through the console. Instead, you can update the descriptor in its extracted location on the server. You must delete and redeploy the application to cause the changes to take effect.

The console module samples provide examples of how to develop the elements of the portal security descriptor.

Chapter 3. Using scripting (wsadmin)

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

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:

- 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.
- 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.
- Clustering servers Includes topics about clusters, such as creating clusters, creating cluster members, querying a cluster state, removing clusters, 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.

Getting started with scripting

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

Make sure that all WebSphere Application Server for z/OS server, administrator and client user IDs (any user IDs that run WebSphere Application Server for z/OS scripts) are run with environment variables LANG and LC_ALL both set to the same locale based on code page IBM-1047. Settings based on any other code page may cause the scripts to fail. See ″Changing the Locale in the Shell″ in *UNIX System Services User's Guide* for more information.

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:

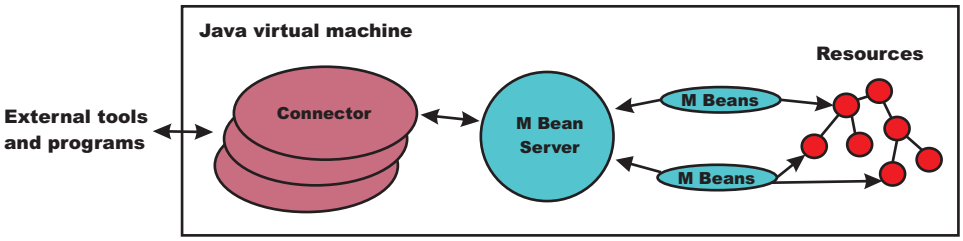


Figure 1: A WebSphere Application Server scripting solution

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.

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

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.

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 Scripting environment properties.

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.

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 Version 6 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
  - Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) JMX Connector
- 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:

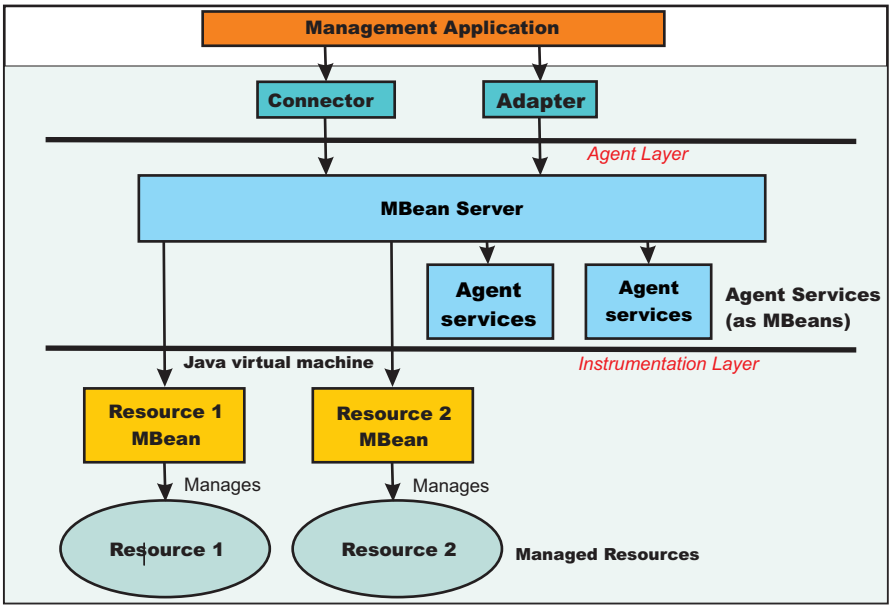


Figure 1: JMX architecture

JMX distributed administration

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

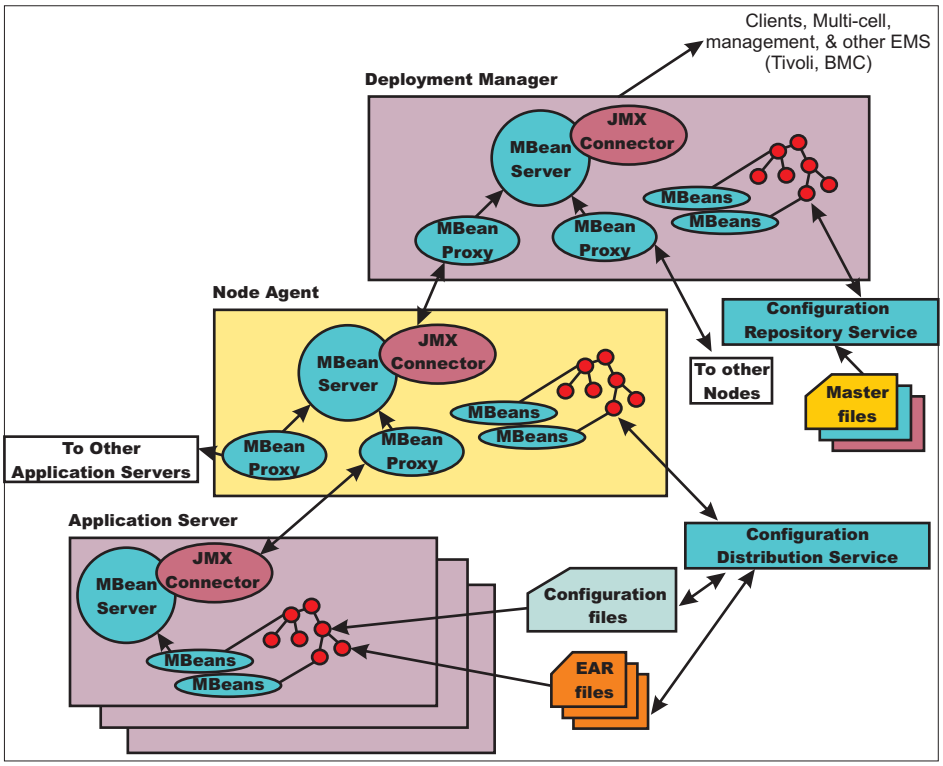


Figure 2: WebSphere Application Server distributed administration of JMX

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. Using the MBean proxy the local MBean server can pass the message to an external MBean server that is located on:
  - A node agent that has an MBean proxy for all the servers within its node. The MBean proxies for other nodes are not used.
  - The deployment manager has MBean proxies for all the node agents in the cell.

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 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 an 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 WebSphere Application Server is available in an HTML table that is installed in each copy of the WebSphere Application Server product. Under the main installation directory for the product, there is the `web` directory. Under the `web` directory there is another directory called `mbeanDocs`. In the `mbeanDocs` directory there are several HTML files; one HTML file for each MBean supplied with WebSphere Application Server. There is also an `index.html` file that ties all the individual MBean files together in a top-level navigation tree. Each MBean provides a summary of its attributes, operations, and notifications.

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

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.

Jacl

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

The wsadmin tool uses Jacl V1.3.2.

Deprecation of the Jacl syntax in the wsadmin tool

Deprecation of a product feature does not mean that the feature is removed from the product immediately. Deprecation is a process of announcing the intent to remove the feature at some future time. The WebSphere Application Server deprecation procedure calls for a feature to remain in the product for two full release cycles before the feature can be removed. For more information about deprecation of features, see the Deprecated and removed features article in the *Migrating, coexisting, and interoperating* PDF.

The wsadmin administrative scripting program supports two scripting languages, Jacl and Jython. The Version 6.1 release of WebSphere Application Server represents 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. In the V6.1 release, WebSphere Application Server contains significantly enhanced administrative functions and tooling that support product automation and the use of the Jython syntax. The following new Jython scripting-related enhancements are provided in WebSphere Application Server V6.1:

- The Eclipse-based WebSphere Application Server Tool (AST) now includes substantial support for automation scripting, providing a full function development environment for wsadmin scripts, including a Jython editor, color-coded keyword highlighting, statement completion assistance, and script debugging support.
- Administrative console command assist - A new feature of the WebSphere Application Server 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.

All future enhancements in the area of WebSphere Application Server scripting will focus on use of the Jython syntax. While Jacl will remain as a component that is shipped with WebSphere Application Server for at least two full releases, no new tooling or explicit enhancements will be created for the Jacl syntax.

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 ($) is 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 will give 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 a easy way to clear out a big data structure. The existence of a variable can be tested with the **info exists** command. You may have to test for the existence of the variable because the `incr` parameter requires that a variable exist first, for example:

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

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 foobar]
=> 6
```

In this example, the nested command is the following: `string length foobar`. The **string** command performs various operations on strings. In this case, the command asks for the length of the string `foobar`. 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 his 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 this to quote 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 backslashes that are 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 quotes:

Use double quotes and curly braces to group words together. Quotes allow 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 devide procedure:

```
divide 20 5
```

And it will give the result like below:

```
4
```

It is not really necessary to use the variable c in this example. The procedure body could 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 could 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 may 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 only exist 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.

- **global** command - 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`. Instead, it is put into the `argv0` variable. 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

String 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 may be additional arguments depending on the operation.

The following table includes a summary of the **string** command:

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 string index	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 character in 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 lower case.
string toupper string	Returns string in upper case.
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.
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  foo  z
=> z

append  foo 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 will be `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 a particular instance 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 quotes to group together words with white space into a single list element.

The following table includes commands that are related to lists:

Command	Description
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 foo $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:

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 indices defined for arr, or those that match the string match pattern.
array set arr list	Initializes the array arr from list, which should have the same form as the list returned by get.
array size arr	Returns the number of indices 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 determines 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 only works on the last pattern-body pair. If you use the default pattern on an earlier body, it will be 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 will be 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 will be like the following output, assuming that there is only one server. If there is more then one servers, it will print 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 will run. 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 will be like the following output, assuming that there is only one server in the environment. If there is more then one server, it will print 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 will occur 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 will contain 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 needs to be returned.

Namespaces

Jacl keeps track of 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*.

```
source /temp/script/testProcedure.jacl
printName Cathy Smith
```

Create a script called *testProcedure.jacl*.

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

Pass the following path as a script argument.

```
wsadmin -lang jacl -f '/temp/script/test1.jacl'
```

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

For more information about Jacl, see the Scripting: Resources for Learning article.

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:

Basic function

The function is either the name of a built-in function or a Jython function. For example:

```
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.

Variable

To assign objects to names, the target of an assignment should be 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 similar to the rules for the C language, for example:

- An underscore character (_) or a letter plus any number of letters, digits or underscores

The following reserved words can not be used for variable names:

```
and     assert   break    class   continue
def     del      elif     else    except
exec    inally     for      from    global
if     importin   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 x[:-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 needs to 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. For example:

```
text =  "This is a tests of a long lines" \
" continuing lines here."
print text
=> This is a tests 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):
  try:
     res = []
     for x in seq1:
     if x in seq2:
       res.append(x)
     return res
  except:
```

To call the function above, 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`. The name of the program, or script, is not part of sys.argv. sys.argv is an array, so you use the index command to extract items from the argument list, for example:

```
import sys
first  =  sys.argv[0]
second  = sys.argv[1]
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. 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 they appear on. When statements are too long to fit on a single line you can also add a back sash (\) 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.

If

The `if` statement selects actions to perform. The `if` statement may 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 test3
 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 executes a block of indented statements as long as a test at the top keeps evaluating a true value. The general format of an 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 aborts execution 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 executes 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 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_type, sys.exc_value

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...'
```

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:

```
execfile('/temp/script/testFunctions.py')
print printName('Cathy', 'Smith')
```

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:

```
wsadmin -lang jython -f '/temp/script/test1.py'
```

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

For more information about the Jython language, see the Scripting: Resources for Learning article.

Scripting objects

The wsadmin tool operates on configurations and running objects through the following set of management objects: AdminConfig, AdminControl, AdminApp, AdminTask, and Help.

Each of these objects has commands that you can use to perform administrative tasks. To use the scripting objects, specify the scripting object, a command, and command parameters. For example:

Using Jacl:

```
$AdminConfig attributes ApplicationServer
```

Using Jython:

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

where `AdminConfig` is the scripting object, `attributes` is the command, and `ApplicationServer` is the command parameter.

To find out more specific information about each of the scripting objects, including command and command parameter information, see AdminConfig, AdminApp, AdminControl, AdminTask, or Help.

WebSphere Application Server system management separates administrative functions into two categories: functions that work with the configuration of WebSphere Application Server installations, and functions that work with the currently running objects in WebSphere 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. You can create, query, change, or remove this configuration without starting an application server process. The **AdminConfig object**, the **AdminTask object**, and the **AdminApp object** handle configuration functionality. You can invoke configuration functions with or without being connected to a server.

The second entity represents 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 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 WebSphere Application Server scripting support provides functions to locate configuration objects, and running objects. Objects in the configuration do not always represent objects that are currently running. The **AdminControl object** manages running objects.

You can 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

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 article or you can also use the **Help** command, for example:

Using Jacl:

```
$Help help
```

Using Jython:

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

Using the AdminApp object for scripted administration

Use the AdminApp object to manage applications.

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.

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.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

To see a list of all available commands for the AdminApp object:
- See the Commands for the AdminApp object article.
- You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminApp help
  ```

  Using Jython:

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

Listing applications with the wsadmin tool:

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.
- Query the configuration and create a list of installed applications, for example:
  - Using Jacl:

    ```
    $AdminApp list
    ```

  - Using Jython:

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

  where:

`$`	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")
    ```

  where:

`$`	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 with the wsadmin tool:

Use the wsadmin tool to configure an application's settings.

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

1. 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]')
       ```

     where:

`$`	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')
    ```

  where:

`$`	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.

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Using the 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.

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.

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. If a scripting client is connected to a deployment manager, then all MBeans in all server processes are visible. If a scripting client is connected to a node agent, all MBeans in all server processes on that node are accessible. 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:

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

  Using Jacl:

  ```
  $AdminControl help
  ```

  Using Jython:

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

ObjectName, Attribute, and AttributeList classes:

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 WebSphere Application Server 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:

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:

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.
  - If the client attaches to a node agent, the list contains MBeans that run in the node agent and MBeans that run on all application servers on that node.
  - If the client attaches to a deployment manager, the list contains MBeans that run in the deployment manager, all of the node agents communicating with that deployment manager, and all application servers on the nodes served by those node agents.
- 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 tool:

Use scripting and the wsadmin tool to specify running objects.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to specify running objects:

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)
       ```

     where:

`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 retuns 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)
```

where:

`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
`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 Jacl syntax.

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 with the wsadmin tool:

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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

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)
     ```

where:

`$`	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
`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)
     ```

   where:

`$`	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 tool:

You can use scripting to invoke operations on running objects.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to perform operations on running objects:

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

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

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

   where:

`$`	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,*')
     ```

   where:

`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')
     ```

where:

`$`	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

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 with the wsadmin tool:

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to modify attributes on running objects:

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

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

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

   where:

`$`	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
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, for example:
   - Using Jacl:

     `set ts1 [$AdminControl completeObjectName` *name*`]`
   - Using Jython:

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

where:

`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, for example:
   - Using Jacl:

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

   - Using Jython:

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

   where:

`$`	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.

Synchronizing nodes with the wsadmin tool:

You can propagate node changes using scripting and the wsadmin tool. This article applies to network deployment installations only.

A node synchronization is necessary in order to propagate configuration changes to the affected node or nodes. By default, this situation occurs periodically, as long as the node can communicate with the deployment manager. You can propagate changes explicitly by performing the following steps:

1. Set the variable for node synchronization.
   - Using Jacl:

     ```
     set Sync1 [$AdminControl completeObjectName type=NodeSync,node=myNodeName,*]
     ```

   - Using Jython:

     ```
     Sync1 = AdminControl.completeObjectName('type=NodeSync,node=myNodeName,*')
     ```

   where:

`set`	is a Jacl command
`Sync1`	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=NodeSync,node=`*`myNodeName`*	is a fragment of the object name. The complete name is returned by this command. This fragment is used to find the matching object name which is the SyncNode object for the *myNodeName* node, where *myNodeName* is the name of the node that you use to synchronize configuration changes. For example: `type=Server, name=serv1`. It can be any valid combination of domain and key properties. For example, type, name, cell, node, process, and so on.

   Example output:

   ```
   WebSphere:platform=common,cell=myNetwork,version=5.0,name=node
   Sync,mbeanIdentifier=nodeSync,type=NodeSync,node=myBaseNode,
   process=nodeagent
   ```

2. Synchronize the node by issuing the following command:
   - Using Jacl:

     ```
     $AdminControl invoke $Sync1 sync
     ```

   - Using Jython:

     ```
     AdminControl.invoke(Sync1, 'sync')
     ```

   where:

`$`	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
`invoke`	is an AdminControl command
`Sync1`	evaluates the ID of the server that is specified in step number 1
`sync`	is an attribute of modify command

   Example output:

   ```
   true
   ```

   You receive an output value of `true`, if the synchronization completes.

When the synchronization is complete, the files created in the /WebSphere/DeploymentManager/config directory now exists on the *mynode* node in the /WebSphere/AppServer/config directory.

Using the AdminConfig object for scripted administration

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

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.

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 connnected mode, the availability of the AdminConfig commands depend on the type of server to which a scripting client is connected in a Network Deployment installation.

The AdminConfig commands are available only if a scripting client is connected to a deployment manager. When connected to a node agent or an application server, the AdminConfig commands will not be available because the configuration for these server processes are copies of the master configuration that resides in the deployment manager. The copies are created in a node machine when configuration synchronization occurs between the deployment manager and the node agent. You should make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

- The following steps provide a general method to 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 article. You can also use the **Help** command, for example:

  Using Jacl:

  ```
  $AdminConfig help
  ```

  Using Jython:

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

Creating configuration objects using the wsadmin tool:

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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.

1. Use the AdminConfig object **listTemplates** command to list available templates:
   - Using Jacl:
     ```
     $AdminConfig listTemplates JDBCProvider
     ```
   - Using Jython:
     ```
     AdminConfig.listTemplates('JDBCProvider')
     ```

   where:

`$`	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

2. 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/')
     ```

   where:

`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 host name of the node where the new object is added

3. Specify the template that you want to use:
   - Using Jacl:
     ```
     set t1 [$AdminConfig listTemplates JDBCProvider "DB2 Universal JDBC Driver Provider (XA)"]
     ```
   - Using Jython:
     ```
     t1 = AdminConfig.listTemplates('JDBCProvider', 'DB2 Universal JDBC Driver Provider (XA)')
     ```

   where:

`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.

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

     ```
     $AdminConfig createUsingTemplate JDBCProvider $n1 {{name newdriver}} $t1
     ```

   - Using Jython:

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

   where:

`$`	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.

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Interpreting the output of the AdminConfig attributes command using scripting:

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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.

- 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 tool:

Specify configuration objects with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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

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/')
       ```

     where:

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
`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')
```

where:

`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.
`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 Jacl syntax.

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 tool:

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

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to create a list of attributes of configuration objects:

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')
     ```

   where:

`$`	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')
     ```

   where:

`$`	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')
  ```

where:

`$`	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 with the wsadmin tool:

You can modify configuration objects using scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to modify a configuration object:

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/')
     ```

   where:

`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)
```

where:

`$`	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, for example:
   - Using Jacl:

     ```
     $AdminConfig modify $jdbcProvider1 {{description "This is my new description"}}
     ```

   - Using Jython list:

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

   - Using Jython string:

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

   where:

`$`	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

   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. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

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.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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

1. 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/')
   ```

   where:

`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

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

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

   where:

`$`	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

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

The WebSphere 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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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

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
```

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 WebSphere Application Server configuration using the wsadmin tool:

You can use the wsadmin AdminConfig and AdminApp objects to make changes to the WebSphere Application Server configuration.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information. For this task, the wsadmin scripting client must be connected to the deployment manager server in a network deployment environment.

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 WebSphere Application Server installation.

1. Set a variable for creating a server:
   - Using Jacl:

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

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

   where:

`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

2. 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]]')
     ```

where:

`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.

3. 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]]]]')
     ```

   where:

`$`	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.
4. Install an application on the server.
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Modifying nested attributes with the wsadmin tool:

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

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.

1. Invoke the AdminConfig object commands interactively, in a script, or use the **wsadmin -c** commands from an operating system command prompt.
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/')
   ```

   where:

`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]]]]')
   ```

   where:

`$`	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.

An alternative way to modify nested attributes is to modify the nested attribute directly, for example:

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.

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.

Use the following commands to save the configuration changes:

1. Using Jacl:

   ```
   $AdminConfig save
   ```

2. Using Jython:

   ```
   AdminConfig.save()
   ```

where:

`$`	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.

You can use the **reset** command of the AdminConfig object to undo changes that you made to your configuration since your last save.

Using the 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.

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.

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.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

In V6.1, you can use parameter name and parameter value pairs instead of using parameter value one-to-one mapping as in V6.x. This way, you can specify the parameters of a step in any order and you do not have to specify option parameters. This applies to all commands for the AdminTask object. For example:

Syntax for V6.x:

```
$AdminTask commandName {-stepName {{param1Value param2Value param3Value ...}}}
```

Syntax for V6.1:

```
$AdminTask commandName {-stepName {-param1Name param1Value -param3Name param3Value -param2Name param2Value ...}}
```

Example using Jacl in V6.x:

```
$AdminTask createCluster {-clusterConfig {{cluster1 true}}}
```

Example using Jython in V6.x:

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

Example using Jacl in V6.1:

```
$AdminTask createCluster {-clusterConfig {-clusterName cluster1 -preferLocal true}}
```

Example using Jython in V6.1:

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

To find out the names of the step parameters, use the following command: `$AdminTask help` *command_name step_name*. For example using Jacl:

```
$AdminTask help createCluster clusterConfig
```

For example using Jython:

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

- Read “Invoking an administrative command in batch mode” on page 122 to use administrative commands in batch mode.
- Read “Invoking an administrative command in interactive mode” on page 127 to use administrative commands in interactive mode.
- Read “Obtaining online help using scripting” to learn how to use scripting for online help.

Obtaining online help using scripting:

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

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.

- 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 adminstrative 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” on page 846 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” on page 846 for more information about specifying parameter options.

Invoking an administrative command in batch mode:

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

To invoke an administrative command in interactive mode, see “Invoking an administrative command in interactive mode” on page 127.

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

     where:

`$`	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)`

where:

`$`	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*`)`

  where:

`$`	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

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`*`)`

  where:

`$`	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 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.

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.

- 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:

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

To invoke an administrative command in batch mode, see “Invoking an administrative command in batch mode” on page 122.

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

where:

$	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]')
  ```

  where:

$	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]')
  ```

  where:

$	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

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]')`

where:

`$`	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 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 (*).

- 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:

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]
...
```

This screen is usually the first interactive screen that is displayed when an administrative command is invoked interactively unless the invoked command does not contain any target object and non-step command parameters. If a command does not have a target object, then the prompt for the target object is skipped. The number of parameters depends on the number of arguments in the Argument area of the command-specific help. If an input is required, then an asterisk (*) is placed in front of the title. The parameter name is displayed for information and is the name that is used to set this parameter in batch mode. If a parameter is restricted to a set of values, then the valid choices are displayed. If current or default value is available, it is displayed. You can accept the existing value by pressing the Enter key. To add or change an existing value, enter a new value and click Enter.

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.

Starting the wsadmin scripting client

You can use the wsadmin tool to manage a WebSphere Application Server Network Deployment V6.x installation, as well as configuration, application deployment, and server run-time operations.

The WebSphere Application Server wsadmin tool provides the ability to run scripts. The WebSphere Application Server wsadmin tool only supports the Jacl and Jython scripting languages.

Before starting the wsadmin tool with security enabled you must set up the following:

- SSL considerations for WebSphere Application Server administrators
- Defining SSL security for clients and servers

You must start the wsadmin scripting client before you perform any other task using scripting.

1. Locate the command that starts the wsadmin scripting client.

   The command for invoking a scripting process is located in the *app_server_root*/bin directory. Use the wsadmin.sh file.

2. Start the wsadmin scripting client. 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:

Option for starting the wsadmin scripting client:	Explanation:	Examples:
Run scripting commands interactively	Run wsadmin with an option other than -f or -c or without an option. An interactive shell is displayed 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.	Using Jacl on operating systems such as AIX or Linux: `wsadmin.sh` If security is enabled: `wsadmin.sh -user wsadmin -password wsadmin` Using Jython on operating systems such as AIX or Linux: `wsadmin.sh -lang jython` By default security is enabled: `wsadmin.sh -lang jython -user wsadmin -password wsadmin` Example output: `WASX7209I: Connected to process server1 on node myhost using SOAP connector; The type of process is: UnManagedProcess` `WASX7029I: For help, enter: "$Help help"` `wsadmin>$AdminApp list` `adminconsole` `DefaultApplication` `ivtApp` `wsadmin>exit`
Run scripting commands as individual commands	Run the wsadmin tool with the -c option.	Using Jacl on operating systems such as AIX or Linux: `wsadmin.sh -c "\$AdminApp list"` or `wsadmin.sh -c '$AdminApp list'` Using Jython on operating systems such as AIX or Linux: `wsadmin.sh -lang jython -c 'AdminApp.list()'` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector; The type of process is: UnManagedProcess` `adminconsole` `DefaultApplication` `ivtApp`

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. WebSphere Application Server for z/OS supports multiple encoding for the Jacl and Jython command files. The default encoding for the command files is ASCII. To run an EBCDIC encoded file, add the following Java virtual machine (JVM) argument to the `wsadmin.sh` file through the -javaoption flag: `-Dscript.encoding=Cp1047` For example: `wsadmin.sh -javaoption -Dprofile.encoding=Cp1047` You can alternatively have two versions of the wsadmin.sh file, one that references the ASCII version of the file and another that references the EBCDIC version of the file. For example, copy the `wsadmin.sh` file to the `wsadminE.sh` file. Then add `-Dscript.encoding=Cp1047` to the `wsadminE.sh` file.	Using Jacl on operating systems such as AIX or Linux: `wsadmin.sh -f al.jacl` where the `al.jacl` file contains the following commands: `set apps [$AdminApp list]` `puts $apps` Using Jython on operating systems such as AIX or Linux: `wsadmin.sh -lang jython -f al.py` where the `al.py` file contains the following commands: `apps = AdminApp.list()` `print apps` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector; The type of process is:` `UnManagedProcess` `adminconsole` `DefaultApplication` `ivtApp`

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. WebSphere Application Server for z/OS supports multiple encoding for Jacl and Jython profile scripts. The default encoding for the profile file is ASCII. To run an EBCDIC encoded profile script file, add the following Java virtual machine (JVM) argument to the `wsadmin.sh` file: `-Dprofile.encoding=Cp1047` For example: `wsadmin.sh -javaoption -Dprofile.encoding=Cp1047` You can alternatively have two versions of the wsadmin.sh file, one that references the ASCII version of the file and another that references the EBCDIC version of the file. For example, copy the wsadmin.sh file to the wsadminE.sh file. Then add `-Dprofile.encoding=Cp1047` to the wsadminE.sh file. 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.	Using Jacl on operating systems such as AIX or Linux: `wsadmin.sh -profile alprof.jacl` where the `alprof.jacl` file contains the following commands: `set apps [$AdminApp list]` `puts "Applications currently installed:\n$apps"` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector;` `The type of process is: UnManagedProcess` `Applications currently installed:` ` adminconsole` ` DefaultApplication` ` ivtApp` ` WASX7029I: For help, enter: "$Help help"` ` wsadmin>` Using Jython on operating systems such as AIX or Linux: `wsadmin.sh -lang jython -profile alprof.py` where the `alprof.py` file contains the following commands: `apps = AdminApp.list()` `print "Applications currently installed:\n " + apps` Example output: `WASX7209I: Connected to process "server1" on node myhost using SOAP connector;` `The type of process is: UnManagedProcess` `Applications currently installed:` ` adminconsole` ` DefaultApplication` ` ivtApp` ` WASX7029I: For help, enter: "Help.help()"` ` wsadmin>`

Scripting: Resources for learning

Use the following links to find relevant supplemental information about the Jacl and Jython scripting languages, and about using scripting with WebSphere Application Server.

The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming instructions and examples

- Java command language
- Jacl: A Tcl implementation in Java
- Charming Jython
- Jython
- Sample scripts for WebSphere Application Server

Deploying applications using scripting

Use these topics to learn more about deploying applications with scripting and the wsadmin tool.

This topic contains the following tasks:

Installing applications with the wsadmin tool

You can use the wsadmin tool and scripting to install an application into the run time.

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

On a single server installation, the server must be running before you install an application. See the "Starting servers using scripting" on page 223 article for more information.

On a network deployment installation, the deployment manager must be running before you install an application. See the "startManager command" on page 932 article for more information.

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 a set of options. See the "Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands" on page 769 article for a list of valid options for the **install** and **installinteractive** commands. 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:

```
AdminApp.options()
```

For more information for the **options**, **install**, or **installinteractive** commands, see the “Commands for the AdminApp object” on page 742 article.

The application that you install must be an enterprise archive file (EAR), a Web 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 be able to install it. The wsadmin tool uses these extensions to figure out the archive type. If the file is a WAR or JAR file, it will be automatically wrapped as an EAR file.

If you are installing an application that has the AdminApp useMetaDataFromBinary option specified, then you can only install this application on a WebSphere Application Server V6.x deployment target. This also applies to editing the application, using the AdminApp **edit** command, after you install it. If you use the V5.x wsadmin tool to install or edit an application on a WebSphere Application Server V6.x cell, only the steps available for the V5.x wsadmin tool will be shown.

Perform the following steps to install an application into the run time:

1. Install the application.
   - Using 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 Jacl:

         ```
         $AdminApp install MyStuff/application1.ear {-server serv2}
         ```

       - Using Jython list:

         ```
         AdminApp.install('MyStuff/application1.ear', ['-server', 'serv2'])
         ```

       - Using Jython string:

         ```
         AdminApp.install('MyStuff/application1.ear', '[-server serv2]')
         ```

       where:

`$`	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

     – For a network deployment installation only, the following command uses the EAR file and the command option information to install the application on a cluster:
       - Using Jacl:

         ```
         $AdminApp install MyStuff/application1.ear {-cluster cluster1}
         ```

       - Using Jython list:

         ```
         AdminApp.install('MyStuff/application1.ear', ['-cluster', 'cluster1'])
         ```

       - Using Jython string:

         ```
         AdminApp.install('MyStuff/application1.ear', '[-cluster cluster1]')
         ```

       where:

`$`	is a Jacl operator for substituting a variable name with its value
`AdminApp`	is an object allowing application objects to be managed
`install`	is an AdminApp command
MyStuff/application1.ear	is the name of the application to install

`cluster`	is an installation option
`cluster1`	the value of the cluster option which will be cluster name

- Using interactive mode, the following command changes the application information by prompting you through a series of installation tasks:
  - Using Jacl:

    ```
    $AdminApp installInteractive MyStuff/application1.ear
    ```

  - Using Jython:

    ```
    AdminApp.installInteractive('MyStuff/application1.ear')
    ```

  where:

`$`	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

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Uninstalling applications with the wsadmin tool

You can uninstall applications with the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See “Starting the wsadmin scripting client” on page 135 for more information.

Steps to uninstall an application follow:

1. 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')
     ```

   where:

`$`	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

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Uninstalling an application removes it from the WebSphere Application Server configuration and from all the servers that the application was installed on. The application binaries (EAR file contents) are deleted from the installation directory. This occurs when the configuration is saved for single server WebSphere Application Server editions or when the configuration changes are synchronized from deployment manager to the individual nodes for network deployment configurations.

Managing Integrated Solutions Console applications using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

The Integrated Solutions Console EAR file must be unarchived before you install it.

- To deploy a portlet based Integrated Solutions Console application into the Integrated Solutions Console EAR file, use the following examples:
  - Using Jacl:

```
$AdminApp update isclite modulefile {-operation add -contents
/WebSphere/AppServer/systemApps/isclite.ear/upzippedWarName
-contenturi upzippedWARName -usedefaultbindings -contextroot contextroot

$AdminConfig save
```

- To remove a portlet based Integrated Solutions Console WAR file, use the following examples:
  - Using Jacl:

```
$AdminApp update isclite modulefile "-operation delete -contenturi WarName"

$AdminConfig save
```

Managing deployed applications using scripting

Use these topics to learn more about managing deployed applications with scripting and the wsadmin tool.

This topic contains the following tasks:

Starting applications with scripting

Use scripting and the wsadmin tool to start applications.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

You must install the application before starting it. See the “Installing applications with the wsadmin tool” on page 139 article for more information.

Perform the following steps to start an application:

1. 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
     ```

   where:

`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 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
   ```

2. 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')
     ```

   where:

`$`	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

Updating installed applications with the wsadmin tool

Use the wsadmin tool and scripting to update installed applications on an application server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Before starting an application, it must be installed. See the “Installing applications with the wsadmin tool” on page 139 article for more information.

Both the **update** command and the **updateinteractive** command support a set of options. See the “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 769 article for a list of valid options for the **update** and **updateinteractive** commands. 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()
```

For more information for the **options**, **update**, or **updateinteractive** commands, see the “Commands for the AdminApp object” on page 742 article. Perform the following steps to update an application:

1. 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 /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:

`$`	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
       /apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb
       -BindJndiForEJBNonMessageBinding {{"Increment Enterprise Java Bean"
        Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
       ```

– Using Jython string:

```
AdminApp.update('app1', 'modulefile', '[-operation addupdate -contents
/apps/app1/Increment.jar -contenturi Increment.jar -nodeployejb
-BindJndiForEJBNonMessageBinding [["Increment Enterprise Java Bean
" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
```

– Using Jython list:

```
bindJndiForEJBValue = [["Increment Enterprise Java Bean",
"Increment", " Increment.jar,META-INF/ejb-jar.xml", "Inc"]]

AdminApp.update('app1', 'modulefile', ['-operation', 'addupdate', '-contents',
'/apps/app1/Increment.jar', '-contenturi','Increment.jar' '-nodeployejb',
`-BindJndiForEJBNonMessageBinding', bindJndiForEJBValue])
```

where:

`$`	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
`"Increment Enterprise Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc`	is the value of the BindJndiForEJBNonMessageBinding option
`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 /apps/app1/app1Partial.zip}
```

  – Using Jython string:

```
AdminApp.update('app1', 'partialapp', '[-contents /apps/app1/app1Partial.zip]')
```

  – Using Jython list:

```
AdminApp.update('app1', 'partialapp', ['-contents', '/apps/app1/app1Partial.zip'])
```

  where:

`$`	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

- The following command updates the entire deployed application:
  - Using Jacl:

    ```
    $AdminApp update app1 app {-operation update -contents /apps/app1/newApp1.jar
    -usedefaultbindings -nodeployejb -BindJndiForEJBNonMessageBinding
    {{"Increment Enterprise Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}
    ```

  - Using Jython string:

    ```
    AdminApp.update('app1', 'app', '[-operation update -contents /apps/app1/newApp1.ear
    -usedefaultbindings -nodeployejb -BindJndiForEJBNonMessageBinding
    [["Increment Enterprise Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc]]]')
    ```

  - Using Jython list:

    ```
    bindJndiForEJBValue = [["Increment Enterprise Java Bean", "Increment", " Increment.jar,META-INF/
    ejb-jar.xml", "Inc"]]

    AdminApp.update('app1', 'app', ['-operation', 'update', '-contents',
    '/apps/app1/NewApp1.ear', '-usedefaultbindings', '-nodeployejb',
    `-BindJndiForEJBNonMessageBinding', bindJndiForEJBValue])
    ```

    where:

$	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 Enterprise Java Bean" Increment Increment.jar,META-INF/ejb-jar.xml Inc	is the value of the BindJndiForEJBNonMessageBinding option
bindJndiForEJBValue	is a Jython variable containing the value of the BindJndiForEJBNonMessageBinding option

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a Network Deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Stopping applications with scripting

You can stop applications using the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

The following example stops all running applications on a server:

1. 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
     ```

   where:

`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
   ```

2. 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
     ```

   where:

`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
```

3. 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.

Once you complete the steps for this task, all running applications on the server are stopped.

Listing the modules in an installed application with scripting

Use the AdminApp object **listModules** command to list the modules in an installed application.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Use this example:

- Using Jacl:

  ```
  $AdminApp listModules DefaultApplication -server
  ```

- Using Jython:

  ```
  print AdminApp.listModules('DefaultApplication', '-server')
  ```

  where:

$	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: 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 scripting

Use the wsadmin tool and scripting to determine if an application is running.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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,*')
  ```

  where:

`$`	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=Application,name=`*`myApplication`*	is the hierarchical containment path of the configuration object
`print`	is a Jython command

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
```

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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

You can use the AdminApp object to set configurations in an application. Some configuration settings are not available through the AdminApp object.

1. 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:

`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)
   ```

2. 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
     ```

where:

`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

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#ApplicationDeployment_1)
```

3. 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:

`$`	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 and application for session management, it is recommended that you specify each attribute.

4. 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. 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:

`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

5. 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:

`$`	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').split(java.lang.System.getProperty('line.separator')
    appConfig = configs[1:len(configs)-1]
    SM = AdminConfig.showAttribute (appConfig, 'sessionManagement')
    AdminConfig.modify (SM, attrs)
    ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a Network Deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

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.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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.

1. 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:

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)
```

2. 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]
     ```

     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:

`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

3. 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:

`$`	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"
```

4. 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.

   - 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]
     ```

     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])
```

Example output using Jython:

```
[sessionManagement, [[enableSecurityIntegration, true], [maxWaitTime, 30], [sessionPersistenceMode, NONE]]
```

5. 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:

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

6. 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.
   - Using Jacl:

     ```
     foreach module $mod1 {
          if {[regexp WebModuleDeployment $module] == 1} {
             $AdminConfig create WebModuleConfig $module $webAttrs
       $AdminConfig save
          }
       }
     ```

   - Using Jython:

```
arrayModules = mod1[1:len(mod1)-1].split(" ")
for module in arrayModules:
 if module.find('WebModuleDeployment') != -1:
  AdminConfig.create('WebModuleConfig', module, webAttrs)
  Adminconfig.save()
```

Example output:

```
myWebModuleConfig(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#WebModuleConfiguration_1)
```

7. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
8. In a Network Deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Exporting applications using 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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Exporting applications enables you to back them up and preserve their binding information.

- Export an enterprise application to a location of your choice, for example:
  - Using Jacl:

    ```
    $AdminApp export app1 /mystuff/exported.ear
    ```

  - Using Jython:

    ```
    AdminApp.export('app1', '/mystuff/exported.ear')
    ```

  where:

`$`	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:

    ```
    $AdminApp exportDDL app1 /mystuff
    ```

  - Using Jython:

    ```
    AdminApp.exportDDL('app1', '/mystuff')
    ```

  where:

`$`	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

Configuring a shared library using scripting

You can use scripting to configure a shared library for application servers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure an application server to use a shared library.

1. 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
     ```

   where:

`set`	is a Jacl command
`serv`	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
`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)
   ```

2. Create the shared library in the server. For example:
   - Using Jacl:

     ```
     $AdminConfig create Library $serv {{name mySharedLibrary} {classPath /mySharedLibraryClasspath}}
     ```
   - Using Jython:

     ```
     print AdminConfig.create('Library', serv, [['name', 'mySharedLibrary'], ['classPath',
     '/mySharedLibraryClasspath']])
     ```

   where:

`$`	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)
```

3. 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
     ```

   where:

`set`	is a Jacl command
`appServer`	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
`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
```

4. 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]
       ```

     where:

`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']])
    ```

  where:

set	is a Jacl command
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
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)
  ```

5. 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} {sharedClassloader true}}
     ```
   - Using Jython:

     ```
     print AdminConfig.create('LibraryRef', classLoader1, [['libraryName', 'MyshareLibrary'],
     ['sharedClassloader', 'true']])
     ```

   where:

$	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

`sharedClassloader`	is an attribute
true	is the value of the attribute
`print`	is a Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#LibraryRef_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring a shared library for an application using scripting

This task uses the AdminConfig object to configure a shared library for an application.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

You can use the AdminApp object to set certain configurations in the application.

1. 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. Idenitfy 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
          ```

        where:

`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:

          ```
          set library [$AdminConfig create Library $n1 {{name mySharedLibrary}
          {classPath /mySharedLibraryClasspath}}]
          ```

        – Using Jython:

```
library = AdminConfig.create('Library', n1, [['name', 'mySharedLibrary'],
['classPath', '/mySharedLibraryClasspath']])
print library
```

where:

`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
`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
    ```

  where:

`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)
```

2. 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
     ```

where:

`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)
```

3. 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
     ```

   where:

`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)
   ```

4. 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
     ```

   where:

`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)
```

5. Associate the shared library in the application through the class loader. For example:
   - Using Jacl:

     ```
     $AdminConfig create LibraryRef $classLoad1 {{libraryName
     MyshareLibrary} {sharedClassloader true}}
     ```

   - Using Jython:

     ```
     print AdminConfig.create('LibraryRef', classLoad1, [['libraryName',
     'MyshareLibrary'], ['sharedClassloader', 'true']])
     ```

   where:

`$`	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
`libraryName`	is an attribute
`MyshareLibrary`	is the value of the libraryName attribute
`sharedClassloader`	is an attribute
`true`	is the value of the sharedClassloader attribute

Example output:

```
(cells/mycell/applications/myApp.ear/deployments/myApp|deployment.xml#LibraryRef_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Setting background applications using scripting

You can enable or disable a background application using scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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.

1. 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:/')
     ```

   where:

`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

2. Enable the background application. For example:
   - Using Jacl:
     ```
     $AdminConfig modify $applicationDeployment "{backgroundApplication true}"
     ```
   - Using Jython:
     ```
     AdminConfig.modify(applicationDeployment, ['backgroundApplication', 'true'])
     ```

   where:

`$`	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
`applicationDeployment`	is a variable name that was set in step 1
`backgroundApplication`	is an attribute
`true`	is the value of the backgroundApplication attribute

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a Network Deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Modifying WAR class loader policies for applications using scripting

You can use scripting and the wsadmin tool to modify WAR class loader policies for applications.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

To modify WAR class loader policies for an application, perform the following steps:

1. 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 [$AdminConfig getid /Deployment:MyApp/]
     ```
   - Using Jython:
     ```
     dep = AdminConfig.getid("/Deployment:MyApp/")
     ```
2. Identify the deployed object and set it to the deployedObject variable. For example:
   - Using Jacl:
     ```
     set deployedObject [$AdminConfig showAttribute $dep deployedObject]
     ```
   - Using Jython:
     ```
     deployedObject = AdminConfig.showAttribute(dep, "deployedObject")
     ```
3. 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]'
     ```
4. 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']])
     ```
5. 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 class loader modes for applications using scripting

You can modify class loader modes for an application with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

To modify class loader modes for an application, perform the following steps:

1. 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/')
     ```

2. Identify the deployed object and set it to the depObject variable. For example:
   - Using Jacl:
     ```
     set depObject [$AdminConfig showAttribute $dep deployedObject]
     ```
   - Using Jython:
     ```
     depObject = AdminConfig.showAttribute(dep, 'deployedObject')
     ```
3. Identify the class loader and set it to the classldr variable. For example:
   - Using Jacl:
     ```
     set classldr [$AdminConfig showAttribute $depObject classloader]
     ```
   - Using Jython:
     ```
     classldr = AdminConfig.showAttribute(depObject, 'classloader')
     ```
4. 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]
     ```
5. 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']])
     ```
6. 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 scripting

You can use the wsadmin tool and scripting to modify the starting weight of an application.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

To modify the starting weight of an application, perform the following steps:

1. 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/")
     ```

2. 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")
     ```
3. 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]
     ```
4. 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']])
     ```
5. 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]
     ```

Commands for the WSScheduleCommands group of the AdminTask object

Use the commands in the WSScheduleCommands group to create or delete a schedule. The commands for this group do not require a target object. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the WSScheduleCommands group of the AdminTask object:

Table 1.

Command name:	Description:	Parameters and return values:	Examples:
createWSSchedule	The **createWSSchedule** command creates the scheduler settings in the configuration.	• Parameters: **- 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, required) **- dayOfWeek** The day of the week to check for expired certificates. (Integer, required) **- hour** The hour of the day to check for expired certificates. (Integer, required) **- minute** The minute to check for expired certificates. Use this parameter with the hour parameter. (Integer, required) **- nextStartDate** The next time, in seconds, to check for expired certificate. (Long, optional) • Returns: The configuration object name of the scheduler object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createWSSchedule { -name testSchedule –frequency 30 –hour 1 –minute 30 }` • Using Jython string: `AdminTask.createWSSchedule ('[-name testSchedule –frequency 30 –hour 1 –minute 30]')` • Using Jython list: `AdminTask.createWSSchedule (['-name', 'testSchedule', '–frequency', '30', '–hour', '1', '–minute', '30'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSSchedule {-interactive}` • Using Jython string: `AdminTask.createWSSchedule ('[-interactive]')` • Using Jython list: `AdminTask.createWSSchedule (['-interactive'])`

Table 1. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteWSSchedule	The **deleteWSSchedule** command deletes the settings of a scheduler from the configuration.	• Parameters: **- name** The name that uniquely identifies the scheduler. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSSchedule {-name testSchedule}` • Using Jython string: `AdminTask.deleteWSSchedule ('[-name testSchedule]')` • Using Jython list: `AdminTask.deleteWSSchedule (['-name', 'testSchedule'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSSchedule {-interactive}` • Using Jython string: `AdminTask.deleteWSSchedule ('[-interactive]')` • Using Jython list: `AdminTask.deleteWSSchedule (['-interactive'])`
getWSSchedule	The **getWSSchedule** command returns the settings of the specified scheduler.	• Parameters: **- name** The name that uniquely identifies the scheduler. (String, required) • Returns: The settings of the scheduler that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getWSSchedule { -name testSchedule}` • Using Jython string: `AdminTask.getWSSchedule ('[-name testSchedule]')` • Using Jython list: `AdminTask.getWSSchedule (['-name', 'testSchedule'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getWSSchedule {-interactive}` • Using Jython string: `AdminTask.getWSSchedule ('[-interactive]')` • Using Jython list: `AdminTask.getWSSchedule (['-interactive'])`

Table 1. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listWSSchedules	The **listWSSchedules** command lists the scheduler.	• Parameters: **- 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) • Returns:	**Batch mode example usage:** • Using Jacl: `$AdminTask listWSSchedules {-displayObjectNames true}` • Using Jython string: `AdminTask.listWSSchedules ('[-displayObjectNames true]')` • Using Jython list: `AdminTask.listWSSchedules (['-displayObjectNames', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listWSSchedules {-interactive}` • Using Jython string: `AdminTask.listWSSchedules ('[-interactive]')` • Using Jython list: `AdminTask.listWSSchedules (['-interactive'])`

Table 1. (continued)

Command name:	Description:	Parameters and return values:	Examples:
modifyWSSchedule	The **modifyWSSchedule** command changes the settings of an existing scheduler.	• Parameters: **- 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) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyWSSchedule {-name testSchedule –frequencey 7}` • Using Jython string: `AdminTask.modifyWSSchedule ('[-name testSchedule –frequencey 7]')` • Using Jython list: `AdminTask.modifyWSSchedule (['-name', 'testSchedule', '–frequencey', '7'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyWSSchedule {-interactive}` • Using Jython string: `AdminTask.modifyWSSchedule ('[-interactive]')` • Using Jython list: `AdminTask.modifyWSSchedule (['-interactive'])`

Commands for the WSNotifierCommands group of the AdminTask object

Use the commands in the WSNotifierCommands group to create or delete a notification. The commands for this group do not require a target object. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the WSNotifierCommands group of the AdminTask object:

Table 2.

Command name:	Description:	Parameters and return values:	Examples:
createWSNotifier	The **createWSNotifier** command creates the notification settings in the configuration. Use this command to notify users when certificates are expired.	• Parameters: **- 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, required) **- sendEmail** Set the value of this parameter to true if you want to e-mail the certificate expiration information. If not, set the value of this parameter to false. (Boolean, required) **- emailList** The list of e-mail addresses where you want to send certificate expiration information. Separate the values in the list with colons (:). (String, optional) • Returns: The configuration object name of the notification object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createWSNotifier { -name testNotifier –logToSystemOut true –sendEmail false}` • Using Jython string: `AdminTask.createWSNotifier ('[-name testNotifier –logToSystemOut true –sendEmail false]')` • Using Jython list: `AdminTask.createWSNotifier (['-name', 'testNotifier', '–logToSystemOut', 'true', '–sendEmail', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSNotifier {-interactive}` • Using Jython string: `AdminTask.createWSNotifier ('[-interactive]')` • Using Jython list: `AdminTask.createWSNotifier (['-interactive'])`

Table 2. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteWSNotifier	The **deleteWSNotifier** command deletes the settings of a notification configuration.	• Parameters: **- name** The name that uniquely identifies the notification configuration. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSNotifier {-name testNotifier}` • Using Jython string: `AdminTask.deleteWSNotifier ('[-name testNotifier]')` • Using Jython list: `AdminTask.deleteWSNotifier (['-name', 'testNotifier'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSNotifier {-interactive}` • Using Jython string: `AdminTask.deleteWSNotifier ('[-interactive]')` • Using Jython list: `AdminTask.deleteWSNotifier (['-interactive'])`
getWSNotifier	The **getWSNotifier** command displays the settings of a particular notification configuration.	• Parameters: **- name** The name that uniquely identifies the notification configuration. (String, required) • Returns: The settings of the notification configuration that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getWSNotifier {-name testNotifier}` • Using Jython string: `AdminTask.getWSNotifier ('[-name testNotifier]')` • Using Jython list: `AdminTask.getWSNotifier (['-name', 'testNotifier'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getWSNotifier {-interactive}` • Using Jython string: `AdminTask.getWSNotifier ('[-interactive]')` • Using Jython list: `AdminTask.getWSNotifier (['-interactive'])`

Table 2. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listWSNotifier	The **listWSNotifier** command lists the notifier from the configuration.	• Parameters: **- 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) • Returns: The notifier information.	**Batch mode example usage:** • Using Jacl: `$AdminTask listWSNotifier {-displayObjectName true}` • Using Jython string: `AdminTask.listWSNotifier ('[-displayObjectName true]')` • Using Jython list: `AdminTask.listWSNotifier (['-displayObjectName', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listWSNotifier {-interactive}` • Using Jython string: `AdminTask.listWSNotifier ('[-interactive]')` • Using Jython list: `AdminTask.listWSNotifier (['-interactive'])`

Table 2. (continued)

Command name:	Description:	Parameters and return values:	Examples:
modifyWSNotifier	The **modifyWSNotifier** command changes the settings of an existing notification configuration.	• Parameters: **- 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 e-mail the certificate expiration information. If not, set the value of this parameter to false. (Boolean, optional) **- emailList** The list of e-mail addresses where you want to send certificate expiration information. Separate the values in the list with colons (:). (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyWSNotifier {-name testNotifier –logToSystemOut false –sendEmail true –emailList tester}` • Using Jython string: `AdminTask.modifyWSNotifier ('[-name testNotifier –logToSystemOut false –sendEmail true –emailList tester]')` • Using Jython list: `AdminTask.modifyWSNotifier (['-name', 'testNotifier', '–logToSystemOut', 'false', '–sendEmail', 'true', '–emailList', 'tester'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyWSNotifier {-interactive}` • Using Jython string: `AdminTask.modifyWSNotifier ('[-interactive]')` • Using Jython list: `AdminTask.modifyWSNotifier (['-interactive'])`

Commands for the WSGateway group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the WSGateway group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

addWSGW Target Service	The **addWSGW Target Service** command adds a target to a gateway service. You must specify the targetService parameter or the targetDestination parameter.	Object name of the GatewayService object	• Parameters: **name** The administrative name of the target service. (Required) **targetDestination** The name of the target destination. This can be within the same bus as the gateway destination or in a different bus. If the target destination is not within the same bus as the gateway destination, you must also specify the targetBus parameter. You must either specify the targetDestination parameter or the targetService parameter. (Conditional) **targetService** The name of the target outbound service. You must either specify the targetDestination parameter or the targetService parameter. (Conditional) **targetBus** The name of the WPM bus that contains the target. (Optional) • Returns: The object name of the target service object that you created.	**Batch mode example usage:** • Using Jacl: `set gwTarget [$AdminTask addWSGW TargetService $gwService {-name "AnotherTarget" -target Service "AnotherService"}]` • Using Jython string: `gwTarget=AdminTask.addWSGWTarget Service(gwService, '[-name AnotherTarget -targetService AnotherService]')` • Using Jython list: `gwTarget=AdminTask.addWSGWTarget Service(gwService, ['-name', 'AnotherTarget', '-target Service', 'AnotherService'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addWSGWTargetService {-interactive}` • Using Jython string: `AdminTask.addWSGWTargetService ('[-interactive]')` • Using Jython list: `AdminTask.addWSGWTargetService (['-interactive'])`

createWSGW Gateway Service	The **createWSGW Gateway Service** command creates a new GatewayService with associated InboundService and TargetService objects. Configuration of the InboundPort and OutboundService/Port that is associated with these objects is done using separate commands.	ObjectName of the gateway instance which the gateway service is created	• Parameters: **-name** Administrative name of the Gateway Service. (required) **-wsdlLocation** Location of the template WSDL. May be a URL or a UDDI business key (UUID). (conditional) **-wsdlServiceName** The name of the service in the WSDL. (conditional) **-wsdlServiceNamespace** The namespace of the service in the WSDL. (conditional) **-targetDestination** The name of the target destination. (conditional) **-targetService** The name of the target outbound service. (conditional) **-requestDestination** The name of the gateway destination. (optional) **-replyDestination** The name of the gateway reply destination. (optional) **-targetBus** The name of the WPM bus containing the target. (optional) **-uddiReference** The reference of the UDDI registry for the WSDL. (optional) **-userId** The user id to use to retrieve the WSDL. (optional) **-password** The password to use to retrieve the WSDL. (optional) • Returns: ObjectName of the created GatewayService object	**Batch mode example usage:** • Using Jacl: `set gwService [$AdminTask createWSGW GatewayService $wsgw {-name MyGatewayService -targetService MyService}]` • Using Jython string: `gwService = AdminTask.createWSGW GatewayService(wsgw, '[-name MyGatewayService -target Service MyService]')` • Using Jython list: `gwService = AdminTask.createWSGW GatewayService(wsgw, ['-name', 'MyGatewayService', '-target Service', 'MyService'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSGW GatewayService {-interactive}` • Using Jython string: `$AdminTask createWSGW GatewayService ('[-interactive]')` • Using Jython list: `$AdminTask createWSGW GatewayService (['-interactive'])`

createWSGW ProxyService	The **createWSGW ProxyService** command creates a new proxy service with an associated inbound service and a target service object with an associated outbound service. Configuration of the inbound port objects that are associated with the inbound service is done using separate commands.	The object name of the gateway instance within which the proxy service is created.	• Parameters: **name** The administrative name of the proxy service. (required) **node** The node where the destinations will be localized. (conditional) **server** The server where the destinations will be localized. (conditional) **cluster** Cluster where the destinations will be localized. (conditional) **-requestDestination** The name of the proxy request destination. (optional) **-replyDestination** The name of the proxy reply destination. (optional) **-wsdlLocation** The location of the proxy WSDL (URL). (optional) • Returns: The object name of the proxy service object that you created.	**Batch mode example usage:** • Using Jacl: `set proxyService [$AdminTask create WSGWProxyService $wsgw {-name MyProxyService -node MyNode -server server1}]` • Using Jython string: `proxyService = AdminTask.createWSGW ProxyService(wsgw, '[-name MyProxyService -node MyNode -server server1]')` • Using Jython list: `proxyService = AdminTask.createWSGW ProxyService(wsgw, ['-name', 'MyProxyService', '-node', 'MyNode', '-server', 'server1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSGWProxyService {-interactive}` • Using Jython string: `AdminTask.createWSGWProxyService ('[-interactive]')` • Using Jython list: `AdminTask.createWSGWProxyService '([-interactive'])`

deleteWSGW Gateway Service	The **deleteWSGW Gateway Service** command deletes a gateway service. It deletes the gateway destination, the corresponding reply destination, the inbound service, the inbound port enablement objects, and all of the associated target service objects. This command does not delete the destinations that are associated with the target services.	Object name of the gateway service object	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSGWGatewayService $gwService` • Using Jython string: `AdminTask.deleteWSGWGatewayService (gwService)` • Using Jython list: `AdminTask.deleteWSGWGatewayService (gwService)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSGWGatewayService {-interactive}` • Using Jython string: `AdminTask.deleteWSGWGatewayService ('[-interactive]')` • Using Jython list: `AdminTask.deleteWSGWGatewayService (['-interactive'])`
deleteWSGW Proxy Service	The **deleteWSGW Proxy Service** command deletes a proxy service that includes the proxy destinations, outbound service, outbound ports, inbound service, and inbound port enablement objects.	Object name of the ProxyService object	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSGWProxyService $proxyService` • Using Jython string: `AdminTask.deleteWSGWProxyService (proxyService)` • Using Jython list: `AdminTask.deleteWSGWProxyService (proxyService)` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSGWProxyService {-interactive}` • Using Jython string: `AdminTask.deleteWSGWProxyService ('[-interactive]')` • Using Jython list: `AdminTask.deleteWSGWProxyService (['-interactive'])`

removeWSGWTarget Service	The **removeWSGWTarget Service** command removes a target service from the gateway service. The destinations that are associated with the target service are not deleted. If the target service that you remove is the default target service, the default is set to the first target service in the set or cleared if none are left.	The object name of the TargetService object.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeWSGWTargetService $gwTarget` • Using Jython string: `AdminTask.removeWSGWTargetService (gwTarget)` • Using Jython list: `AdminTask.removeWSGWTargetService (gwTarget)` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeWSGWTargetService {-interactive}` • Using Jython string: `AdminTask.removeWSGWTargetService ('[-interactive]')` • Using Jython list: `AdminTask.removeWSGWTargetService (['-interactive'])`

Commands for the CoreGroupManagement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the CoreGroupManagement group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

createCore Group	The **createCore Group** command creates a new core group. The core group that you create contains no members.	None	• Parameters: **- coreGroupName** The name of the core group that you are creating. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createCoreGroup {-coreGroupName MyCoreGroup}` • Using Jython string: `AdminTask.createCoreGroup ('[-coreGroupName MyCoreGroup]')` • Using Jython list: `AdminTask.createCoreGroup (['-coreGroupName', 'MyCoreGroup'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCoreGroup {-interactive}` • Using Jython string: `AdminTask.createCoreGroup ('[-interactive]')` • Using Jython list: `AdminTask.createCoreGroup (['-interactive'])`
deleteCore Group	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.	None	• Parameters: **- coreGroupName** The name of the existing core group that will be deleted. (String required) • Returns: None	**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]')`

doesCore Group Exist	The **doesCore Group Exist** command returns a boolean value that indicates if the core group that you specify exists.	None	• Parameters: **coreGroupName** The name of the core group. (String, required) • Returns: A boolean value.	**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'])`
getAllCore Group Names	The **getAll CoreGroup Names** command returns a string that contains the names of all of the existing core groups	None	• Parameters: None • Returns: String array (String[ ])	**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'])`

getCore GroupName ForServer	The **getCore GroupName ForServer** command returns the name of the core group in which the server that you specify is currently a member.	None	• Parameters: **- nodeName** The name of the node that contains the server. (String, required) **- serverName** The name of the server. (String, required) • Returns: The name of the core group that currently contains the server that you specified. (String)	**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'])`
getDefault CoreGroup Name	The **getDefault CoreGroup Name** command returns the name of the default core group.	None	• Parameters: None • Returns: String	**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'])`

moveCluster ToCore Group	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.	None	• Parameters: **- 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) • Returns: None	**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'])`

moveServer ToCore Group	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.	None	• Parameters: **- 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) • Returns: None	**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'])`

Commands for the CoreGroupBridgeManagement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the CoreGroupBridgeManagement group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

createCore GroupAccess Point	The **createCore GroupAccess Point** 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.	Core group bridge settings object for the cell. (ObjectName, required).	• Parameters: **- coreGroupName** The name of the core group for which the core group access point will be created. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createCoreGroupAccessPoint (cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridgeSettings_1) "-coreGroupName DefaultCoreGroup"` • Using Jython string: `AdminTask.createCoreGroupAccessPoint ('cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridgeSettings_1', '[-coreGroupName DefaultCoreGroup]')` • Using Jython list: `AdminTask.createCoreGroupAccessPoint ('cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridgeSettings_1', ['-coreGroupName', 'DefaultCoreGroup']')` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCoreGroupAccess Point {-interactive}` • Using Jython string: `AdminTask.createCoreGroupAccess Point ('[-interactive]')` • Using Jython list: `AdminTask.createCoreGroupAccess Point (['-interactive'])`

deleteCore GroupAccess Points	The **deleteCore GroupAccess Points** command deletes all the core group access points that are associated with a group that you specify.	Core group bridge settings object for the cell. (ObjectName, required)	• Parameters: **- coreGroupName** The name of the core group whose core group access points will be deleted. (String required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteCoreGroupAccessPoints (cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridgeSettings_1) "-coreGroupName DefaultCoreGroup"` • Using Jython string: `AdminTask.deleteCoreGroupAccessPoints ('(cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridgeSettings_1)', '[-coreGroupName DefaultCoreGroup]')` • Using Jython list: `AdminTask.deleteCoreGroupAccessPoints ('(cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridgeSettings_1)', ['-coreGroupName', 'DefaultCoreGroup'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCoreGroup AccessPoints {-interactive}` • Using Jython string: `AdminTask.deleteCoreGroup AccessPoints ('[-interactive]')` • Using Jython list: `AdminTask.deleteCoreGroup AccessPoints (['-interactive'])`

getNamed TCPEnd Point	The **getNamed TCPEnd Point** 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.	The bridge interface object for which the port will be listed. (ObjectName, required)	• Parameters: None • Returns: The port (named end point) object name of the TCP inbound channel instance which resides on the DCS transport channel chain of the bridge interface.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNamedTCPEndPoint (cells/rohitbuildCell01\|coregroup bridge.xml#BridgeInterface_2)` • Using Jython string: `AdminTask.getNamedTCPEndPoint(' (cells/rohitbuildCell01\|core groupbridge.xml#BridgeInterface_2)')` • Using Jython list: `AdminTask.getNamedTCPEndPoint(' (cells/rohitbuildCell01\|core groupbridge.xml#BridgeInterface_2)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNamedTCPEndPoint {-interactive}` • Using Jython string: `AdminTask.getNamedTCPEndPoint ('[-interactive]')` • Using Jython list: `AdminTask.getNamedTCPEndPoint (['-interactive'])`
listCore Groups	The **listCore Groups** command returns a collection of core groups that are related to the core group that you specify.	The name of the core group for which the related core groups will be listed. (String, required)	• Parameters: **- cgBridgeSettings** The group bridge settings object for the cell. (ObjectName, required) • Returns: A set of core group names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listCoreGroups DefaultCore Group "-cgBridgeSettings (cells/rohitbuildCell01\|core groupbridge.xml#CoreGroupBridge Settings_1)"` • Using Jython string: `AdminTask.listCoreGroups('Default CoreGroup', '[-cgBridgeSetting (cells/ rohitbuildCell01\|coregroupbridge.xml# CoreGroupBridgeSettings_1)]')` • Using Jython list: `AdminTask.listCoreGroups('Default CoreGroup', ['-cgBridgeSetting', '(cells/rohitbuildCell01\|coregroup bridge.xml#CoreGroupBridge Settings_1)'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listCoreGroups {-interactive}` • Using Jython string: `AdminTask.listCoreGroups ('[-interactive]')` • Using Jython list: `AdminTask.listCoreGroups (['-interactive'])`

listEligible Bridge Interfaces	The **listEligible Bridge Interfaces** 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.	The core group access point object for which bridge interfaces will be listed. (ObjectName, required)	• Parameters: None • Returns: 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)`	**Batch mode example usage:** • Using Jacl: `$AdminTask listEligibleBridgeInterfaces CGAP_DCG_2(cells/rohitbuildCell01\| coregroupbridge.xml#CoreGroupAccess Point_1089636614062)` • Using Jython string: `AdminTask.listEligibleBridgeInterfaces ('CGAP_DCG_2(cells/rohitbuildCell01\| coregroupbridge.xml#CoreGroupAccess Point_1089636614062)')` • Using Jython list: `AdminTask.listEligibleBridgeInterfaces ('CGAP_DCG_2(cells/rohitbuildCell01\| coregroupbridge.xml#CoreGroupAccess Point_1089636614062)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listEligibleBridge Interfaces {-interactive}` • Using Jython string: `AdminTask.listEligibleBridge Interfaces ('[-interactive]')` • Using Jython list: `AdminTask.listEligibleBridge Interfaces (['-interactive'])`

Configuring servers with scripting

You can configure servers using the wsadmin tool and scripting.

This topic contains the following tasks:

Creating a server using scripting

You can use scripting to create an application server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Creating an application server involves a configuration command. Perform the following steps to create a server:

1. Obtain the configuration ID of the object and assign it to the *mynode* variable, for example:

   Using Jacl:

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

   Using Jython:

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

2. There are two ways to create the server using the node that you specified in the first step. Perform one of the following:
   - Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask createApplicationServer mynode
       {-name test1 –templateName default}
       ```

     - Using Jython:

       ```
       AdminTask.createApplicationServer(mynode,
       ['-name', 'test1', '-templateName', 'default'])
       ```

   - Using the AdminConfig object:
     - Using Jacl:

       ```
       $AdminConfig create Server $node {{name myserv}
       {outputStreamRedirect {{fileName myfile.out}}}}
       ```

     - Using Jython:

       ```
       AdminConfig.create('Server', node, [['name', 'myserv'],
       ['outputStreamRedirect', [['fileName', 'myfile.out']]]])
       ```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring the Java virtual machine using scripting

Use scripting to configure settings for a Java virtual machine.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

To modify the Java virtual machine (JVM) of a server to turn on debug, perform the following steps:

1. There are two ways to perform this task. Choose one of the following:
   - 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.
        - 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 belonging to this server and assign it to the jvm variable.

– 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)
(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaVirtualMachine_2)
```

c. Identify the controller JVM of the server and its servant region JVM.

– Using Jacl:

```
set cjvm [lindex $jvm 0]
set sjvm [lindex $jvm 1]
```

– Using Jython:

```
# get line separator
import java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayJVMs = jvm.split(lineSeparator)
cjvm = arrayJVMs[0]
sjvm = arrayJVMs[1]
```

d. Modify the JVM to turn on debugging.

Using Jacl:

```
$AdminConfig modify $cjvm {{debugMode true} {debugArgs "-Djava.compiler=NONE -Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"}}
$AdminConfig modify $sjvm {{debugMode true} {debugArgs "-Djava.compiler=NONE -Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"}}
```

Using Jython:

```
AdminConfig.modify(cjvm, [['debugMode', 'true'], ['debugArgs',  "-Djava.compiler=NONE -Xdebug
-Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"]])
AdminConfig.modify(sjvm, [['debugMode', 'true'], ['debugArgs',  "-Djava.compiler=NONE -Xdebug
-Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"]])
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring enterprise bean containers using scripting

You can configure enterprise bean containers using scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See “Starting the wsadmin scripting client” on page 135 for more information.

Perform the following steps to configure an enterprise bean container:

1. Identify the application server and assign it to the serv1 variable. For example:
   - 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
```

where:

`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 representing the WebSphere Application Server configuration
`getid`	is an AdminConfig command
`/Cell:mycell/Node:mynode/Server:server1/`	is the hierarchical containment path of the configuration object
`Cell`	is the object type
mycell	is the optional name of the object
`Node`	is the object type
mynode	is the optional name of the object
`Server`	is the object type
server1	is the optional name of the object

Example output:

```
server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
```

2. Identify the EJB container belonging to the server and assign it to the ejbc1 variable. For example:
   - Using Jacl:

     ```
     set ejbc1 [$AdminConfig list EJBContainer $serv1]
     ```

   - Using Jython:

     ```
     ejbc1 = AdminConfig.list('EJBContainer', serv1)
     print ejbc1
     ```

   where:

`set`	is a Jacl command
`ejbc1`	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
`list`	is an AdminConfig command
`EJBContainer`	is the object type **Note:** The name of the object type that you input here 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 specified in step number 1

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#EJBContainer_1)
   ```

3. View all the attributes of the enterprise bean container.
   - The following example command does not show 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)]
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`print`	is a Jython command
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`showall`	is an AdminConfig command
`ejbc1`	evaluates to the ID of the enterprise bean container specified in step number 2

- The following command example includes 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)]]]
```

where:

`$`	is a Jacl operator for substituting a variable name with its value
`print`	is a Jython command
`AdminConfig`	is an object representing the WebSphere Application Server configuration
`showall`	is an AdminConfig command
`ejbc1`	evaluates to the ID of the enterprise bean container specified in step number 2

4. 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]]]])
```

     where:

`$`	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
ejbc1	evaluates to the ID of the enterprise bean container specified in step number 2
cacheSettings	is an attribute of modify objects
cacheSize	is an attribute of modify objects
2500	is the value of the cacheSize attribute
cleanupInterval	is an attribute of modify objects
3500	is 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]])
    ```

  where:

$	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
ejbc1	evaluates to the ID of the enterprise bean container specified in step number 2
inactivePoolCleanupInterval	is an attribute of modify objects
15000	is the value of the inactivePoolCleanupInterval attribute

5. Save the changes. For example:
   - Using Jacl:

     ```
     $AdminConfig save
     ```
   - Using Jython:

     ```
     AdminConfig.save()
     ```

   where:

$	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

Configuring a Performance Manager Infrastructure service using scripting

You can use scripting to configure a Performance Manager Infrastructure service.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Use the following steps to configure the Performance Manager Infrastructure (PMI) service for an application server:

1. Identify the application server and assign it to the s1 variable, for example:
   - Using Jacl:

     ```
     set s1 [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
     ```

   - Using Jython:

     ```
     s1 = AdminConfig.getid('Cell:mycell/Node:mynode/Server:server1/')
     ```

   where:

`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 an attribute
`mycell`	is the value of the Cell attribute
`Node`	is an attribute
`mynode`	is the value of the Node attribute
`Server`	is an attribute
`server1`	is the value of the Server attribute

   Example output:

   ```
   server1(cells/mycell/nodes/mynode/servers/server1|server.xml#Server_1)
   ```

2. Identify the PMI service that belongs to the server and assign it to the `pmi` variable, for example:
   - Using Jacl:

     ```
     set pmi [$AdminConfig list PMIService $s1]
     ```

   - Using Jython:

     ```
     pmi = AdminConfig.list('PMIService', s1)
     print pmi
     ```

   where:

`set`	is a Jacl command
`pmi`	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
`list`	is an AdminConfig command
`PMIService`	is an AdminConfig object
`s1`	evaluates to the ID of the application server specified in step number 1

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#PMIService_1)
   ```

3. Modify the attributes, for example:
   - Using Jacl:

```
$AdminConfig modify $pmi {{enable true}
{initialSpecLevel beanModule=H:cacheModule=H:connectionPoolModule=
H:j2cModule=H:jvmRuntimeModule=H:orbPerfModule=H:servletSessionsModule
=H:systemModule=H:threadPoolModule=H:transactionModule=H:
webAppModule=H:webServicesModule=H:wlmModule=H:wsgwModule=H}}
```

- Using Jython:

```
AdminConfig.modify(pmi, [['enable', 'true'],
['initialSpecLevel', 'beanModule=H:cacheModule=H:connectionPoolModule=
H:j2cModule=H:jvmRuntimeModule=H:orbPerfModule=H:servletSessionsModule=
H:systemModule=H:threadPoolModule=H:transactionModule=H:webAppModule=H:
webServicesModule=H:wlmModule=H:wsgwModule=H']])
```

This example enables PMI service and sets the specification levels for all of components in the server. The following are the valid specification levels for the components:

N	represents **none**
L	represents **low**
M	represents **medium**
H	represents **high**
X	represents **maximum**

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring an ORB service using scripting

You can use scripting to configure an ORB service.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to modify the Object Request Broker (ORB) service for an application server:

1. Identify the application server and assign it to the server variable:
   - 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:

`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)
```

2. Identify the ORB belonging to the server and assign it to the orb variable:
   - Using Jacl:

     ```
     set orb [$AdminConfig list ObjectRequestBroker $s1]
     ```

   - Using Jython:

     ```
     orb = AdminConfig.list('ObjectRequestBroker', s1)
     print orb
     ```

   where:

`set`	is a Jacl command
`orb`	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
`list`	is an AdminConfig command
`ObjectRequestBroker`	is an AdminConfig object
`s1`	evaluates to the ID of server specified in step number 1
`print`	a Jython command

Example output:

```
(cells/mycell/nodes/mynode/servers/server1|server.xml#ObjectRequestBroker_1)
```

3. Modify the 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']])
     ```

   where:

`$`	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
`orb`	evaluates to the ID of ORB specified in step number 2
`connectionCacheMaximum`	is an attribute
`252`	is the value of the connectionCacheMaximum attribute
`noLocalCopies`	is an attribute
`true`	is the value of the noLocalCopies attribute

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring processes using scripting

You can use scripting and the wsadmin tool to configure processes.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a process:

1. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:

       ```
       $AdminTask setProcessDefinition {-interactive}
       ```

     - Using Jython:

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

   - Using the AdminConfig object:
     a. Identify the server and assign it to the s1 variable. For example:
        - 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:

`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)
        ```

     b. Identify the process definition belonging to this server and assign it to the processDef variable. For example:
        - Using Jacl:

```
set processDef [$AdminConfig list JavaProcessDef $s1]
set controllerProcessDef [lindex $processDefs 1]
set servantProcessDef [lidex $processDefs 1]
```

– Using Jython:

```
processDef = AdminConfig.list('JavaProcessDef', s1)
# get line separator
import java
lineSeparator = java.lang.System.getProperty('line.separator')
arrayPDs = processDefs.split(lineSeparator)
controllerProcessDef = arrayPDs[0]
servantProcessDef = arrayPDs[1]
print controllerProcessDef
print servantProcessDef
```

Example output:

```
(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaProcessDef_1)
(cells/mycell/nodes/mynode/servers/server1:server.xml#JavaProcessDef_2)
```

c. Change the attributes.

– On z/OS systems, the following example shows how to change the process definition of the servant region. You can change the process definition of the controller region by substituting controllerProcessDef for servantProcessDef .

- Using Jacl:

```
$AdminConfig modify $servantProcessDef {{workingDirectory /temp/user1}}
```

- Using Jython:

```
AdminConfig.modify(servantProcessDef, [['workingDirectory', '/temp/user1']])
```

– The following example modifies the stderr file name:

- Using Jacl:

```
set errFile [list stderrFilename \${LOG_ROOT}/server1/new_stderr.log]
set attr [list $errFile]
$AdminConfig modify $servantProcessDef [subst {{ioRedirect {$attr}}}]
```

- Using Jython:

```
errFile = ['stderrFilename', '${LOG_ROOT}/server1/new_stderr.log']
attr = [errFile]
AdminConfig.modify(servantProcessDef, [['ioRedirect', attr]])
```

– The following example modifies the process priority:

- Using Jacl:

```
$AdminConfig modify $processDef {{execution {{processPriority 15}}}}
```

- Using Jython:

```
AdminConfig.modify(processDef, [['execution', [['processPriority', 15]]]])
```

– The following example changes the maximum startup attempts. You can modify this example to change other attributes in the process definition object.

- Using Jacl:

```
$AdminConfig modify $processDef {{monitoringPolicy {{maximumStartupAttempts 1}}}}
```

- Using Jython:

```
AdminConfig.modify(processDef, [['monitoringPolicy', [['maximumStartupAttempts',  1]]]])
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring transaction properties for a server using scripting

Use scripting to configure transaction properties for servers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure the runtime transaction properties for an application server.

1. Identify the transaction service MBean for the application server. The following command returns the transaction service MBean for *server1*.
   - 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
     ```

   where:

`set`	is a Jacl command
`ts`	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
`cell=mycell,node=mynode,process=server1, type=TransactionService`	is 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, process, etc.

   Example output:

   ```
   WebSphere:cell=mycell,name=TransactionService,mbeanIdentifier=TransactionService,
   type=TransactionService,node=mynode,process=server1
   ```

2. Modify the attributes.
   - Using Jacl:

     ```
     $AdminControl setAttributes $ts {{clientInactivityTimeout 30} {totalTranLifetimeTimeout 180}}
     ```

   - Using Jython:

     ```
     AdminControl.setAttributes(ts, [['clientInactivityTimeout', 30],  ['totalTranLifetimeTimeout', 180]])
     ```

   where:

`$`	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
`setAttributes`	is an AdminControl command
`ts`	evaluates to the ID of the transaction service specified in step number 1
`clientInactivityTimeout`	is an attribute
`30`	is the value of the clientInactivityTimeout attribute specified in seconds. A value of 0 means that there is no timeout limit.
`totalTranLifetimeTimeout`	is an attribute

180	is the value of the totalTranLifetimeTimeout attribute specified in seconds. A value of 0 means that there is no timeout limit.

Setting port numbers kept in the serverindex.xml file using scripting

This topic provides reference information about modifying port numbers in the `serverindex.xml` file.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

The end points of the `serverindex.xml` file are part of different objects in the configuration.

Use the following attributes to modify the end point information of the end point attributes for a process:

- **BOOTSTRAP_ADDRESS** of server1 process

  An attribute of the NameServer object that exists inside the server. It is used by the naming client to specify the naming server to look up the initial context. Use one of the following examples:

  – 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 end point, obtain the ID of the NameServer object and issue a **modify** command.

    - 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]]]])
      ```

- **SOAP_CONNECTOR-ADDRESS** of server1 process

  An attribute of the SOAPConnector object that exists inside the server. It is the port that is used by HTTP transport for incoming SOAP requests. Use one of the following examples:

  – 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]')
      ```

  – Using the AdminConfig object. To modify its end point, obtain the ID of the SOAPConnector object and issue a **modify** command, for example:

    - 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]]]])
```

- **DRS_CLIENT_ADDRESS** of server1 process

  An attribute of the SystemMessageServer object that exists inside the server. It is the port used to configure the Data Replication Service (DRS) which is a JMS-based message broker system for dynamic caching. The DRS_CLIENT_ADDRESS attribute is not available if a replication domain and a replicator entry have not been added to the server. Use one of the following examples:

  – Using the AdminTask object:

    - Using Jacl:

      ```
      $AdminTask modifyServerPort server1 {-nodeName mynode -endPointName
      DRS_CLIENT_ADDRESS -host myhost -port 7874}
      ```

    - Using Jython:

      ```
      AdminTask.modifyServerPort ('server1', '[-nodeName mynode -endPointName
      DRS_CLIENT_ADDRESS -host myhost -port 7874]')
      ```

  – Using the AdminConfig object. To modify the end point of the DRS_CLIENT_ADDRESS attribute, obtain the ID of the SystemMessageServer object and issue a **modify** command, for example:

    - Using Jacl:

      ```
      set s [$AdminConfig getid /Cell:mycell/Node:mynode/Server:server1/]
      set sms [$AdminConfig list SystemMessageServer $s]
      $AdminConfig modify $sms {{DRS_CLIENT_ADDRESS {{host myhost} {port 7874}}}}
      ```

    - Using Jython:

      ```
      s = AdminConfig.getid('/Cell:mycell/Node:mynode/Server:server1/')
      sms = AdminConfig.list('SystemMessageServer', s)
      AdminConfig.modify(sms, [['DRS_CLIENT_ADDRESS', [['host', 'myhost'], ['port', 7874]]]])
      ```

- **JMSSERVER_QUEUED_ADDRESS and JMSSERVER_DIRECT_ADDRESS** of server1 process

  An attribute of the JMSServer object that exists inside the server. These are ports used to configure the WebSphere Application Server JMS provider topic connection factory settings. Use one of the following examples:

  – 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 end point, 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]]]])
      ```

- **NODE_DISCOVERY_ADDRESS** of nodeagent process An attribute of the NodeAgent object that exists inside the server. It is the port used to receive the incoming process discovery messages inside a node agent process. Use one of the following examples:
  - 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 end point, 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]]]])
      ```

  - 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]]]])
    ```

- **CELL_DISCOVERY_ADDRESS** of dmgr process An attribute of the deploymentManager object that exists inside the server. It is the port used to receive the incoming process discovery messages inside a deployment manager process. Use one of the following examples:
  - 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]')
      ```

  - Using the AdminConfig object. To modify its end point, 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]]]])
```

- **WC_defaulthost** of server1 process To modify a WC_defaulthost end point use one of the following examples:
  - Using the AdminTask object:
    - Using Jacl:

      ```
      $AdminTask modifyServerPort server1 {-nodeName myNode -endPointName WC_defaulthost -host myhost -port 5555}
      ```

    - Using Jython:

      ```
      AdminTask.modifyServerPort ('server1', '[-nodeName myNode -endPointName WC_defaulthost -host myhost -port 5555]'
      ```

  - 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:
           specialEndPoints [AdminConfig.showAttribute(serverEntry, "specialEndpoints")
          [1:len(specialEndPoints)-1].split(" ")
           for specialEndPoint in specialEndPoints:
              endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
              if endPointNm == "WC_defaulthost":
                 ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
                 AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5555]])
                 break
      ```

- **WC_defaulthost_secure** of server1 process To modify a WC_defaulthost_secure end point use one of the following examples:
  - 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:
     specialEndPoints [AdminConfig.showAttribute(serverEntry, "specialEndpoints")
    [1:len(specialEndPoints)-1].split(" ")
     for specialEndPoint in specialEndPoints:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_defaulthost_secure":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5544]])
           break
```

- **WC_adminhost** of server1 process

  To modify a WC_adminhost end point 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:
     specialEndPoints [AdminConfig.showAttribute(serverEntry, "specialEndpoints")
    [1:len(specialEndPoints)-1].split(" ")
     for specialEndPoint in specialEndPoints:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
        if endPointNm == "WC_adminhost":
           ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
           AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 6666]])
           break
```

- **WC_adminhost_secure** of server1 process

To modify a WC_adminhost_secure end point 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:
     specialEndPoints [AdminConfig.showAttribute(serverEntry, "specialEndpoints")
    [1:len(specialEndPoints)-1].split(" ")
     for specialEndPoint in specialEndPoints:
        endPointNm = AdminConfig.showAttribute(specialEndPoint, "endPointName")
```

```
if endPointNm == "WC_adminhost_secure":
   ePoint = AdminConfig.showAttribute(specialEndPoint, "endPoint")
   AdminConfig.modify(ePoint, [["host", "myhost"], ["port", 5566]])
   break
```

- Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Disabling components using scripting

You can disable components with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to disable the name server component of a configured server. You can modify this example to disable a different component.

1. 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)
   ```

2. List the components belonging to the server and assign them to the components variable.
   - 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)
   ```

3. Identify the name server 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 index 2.
   - 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)
   ```

4. Disable the name server component by changing the nested initialState attribute belonging to the stateManagement attribute. For example:

- Using Jacl:

  ```
  $AdminConfig modify $nameServer {{stateManagement {{initialState STOP}}}}
  ```

- Using Jython:

  ```
  AdminConfig.modify(nameServer, [['stateManagement', [['initialState', 'STOP']]]])
  ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Disabling services using scripting

You can disable the services of a configured server with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to disable the trace service of a configured server. You can modify this example to disable a different service.

1. 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)
   ```

2. 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#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)
   ```

3. Identify the trace service and assign it to the traceService variable.

   Since trace service is the 7th 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)
   ```

4. 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']])
  ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Dynamic caching with scripting

You can configure dynamic caching with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Important: If you use the wsadmin tool to enable servlet caching, you must make sure that portlet fragment caching is also enabled. Similarly if you use the wsadmin tool to disable servlet caching, you must make sure that portlet fragment caching is also disabled. The settings for these two caching functions must stay synchronized. If you enable or disable servlet caching using the administrative console, synchronization is automatically taken care of for you.

To see a list of parameters associated with dynamic caching, use the **attributes** command. For example:

```
$AdminConfig attributes DynamicCache
```

Perform the following steps to enable servlet caching:

1. Locate the server object. The following example selects the first server found:

   Using Jacl:

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

   Using Jython:

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

2. List the web containers and assign them to the `wc` variable, for example:

   Using Jacl:

   ```
   set wc [$AdminConfig list WebContainer $s1]
   ```

   Using Jython:

   ```
   wc = AdminConfig.list('WebContainer', s1)
   ```

3. Set the enableServletCaching attribute to `true` and assign it to the `serEnable` variable, for example:

   Using Jacl:

   ```
   set serEnable "{enableServletCaching true}"
   ```

   Using Jython:

   ```
   serEnable = [['enableServletCaching', 'true']]
   ```

4. Enable caching, for example:

   Using Jacl:

   ```
   $AdminConfig modify $wc $serEnable
   ```

   Using Jython:

   ```
   AdminConfig.modify(wc, serEnable)
   ```

Modifying variables using scripting

Use scripting and the wsadmin tool to modify variables in WebSphere Application Server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to modify a WebSphere Application Server variable:

1. 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
       ```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Increasing the Java virtual machine heap size using scripting

For some servers, it may be required to 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. If the JVM platform is 32-bit AIX and it is required that the application server, deployment manager, or node agent have a heap size greater, perform the following steps:

1. Set attributes that control the heap size for the JVM associated with the server. This can be done using the administrative console, the wsadmin tool, or a Java client.
2. Perform one of the following to increase the heap size of the JVM:

- Create a script to launch the server. Use the -script option with the command that you are using to start the target server, for example, the **startServer**, **startManager**, or **startNode** commands. This will generate a script for that you can use to launch the target server in the future.
- Manually set the environment variables before starting the target server with the **startServer**, **startManager**, or **startNode** commands. With the following settings, you can specify a maximum heap size for the JVM of 11 multiplied by 256 megabytes:

  ```
  export IBM_JVM_LDR_CNTRL_NEW_VALUE=MAXDATA=0XB0000000@DSA
  export LDR_CNTRL=MAXDATA=0XB0000000@DSA
  ```

- Use the AdminTask object. For example:
  - Using Jacl:

    ```
    $AdminTask setJVMMaxHeapSize {-interactive}
    ```

  - Using Jython:

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

For additional information on AIX address space management, view the following sources:
- IBM 32-bit SDK for AIX, Java 2 Technology Edition, Version 1.4 - User Guide
- Getting more memory in AIX for your Java applications

Commands for the PortManagement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the PortManagement group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listApplicationPorts	Use the **listApplicationPorts** command to list the ports in order to access a particular application.	The application name.	• Parameters: None • Returns: The ports that are used by the application that you specified.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listApplicationPorts {-interactive}` • Using Jython string: `AdminTask.listApplicationPorts ('[-interactive]')` • Using Jython list: `AdminTask.listApplicationPorts (['-interactive'])`

listServer Ports	Use the **listServer Ports** command to list the ports that are used by the server that you specify.	The server name (String)	• Parameters: **- nodeName** The name of the node. This parameter is only required when the server name is not unique in the cell. (String, required) • Returns: The ports that are used by the server that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask listServerPorts server1 {-nodeName myNode}` • Using Jython string: `AdminTask.listServerPorts ('server1', '[-nodeName myNode]')` • Using Jython list: `AdminTask.listServerPorts ('server1', ['-nodeName', 'myNode'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listServerPorts {-interactive}` • Using Jython string: `AdminTask.listServerPorts ('[-interactive]')` • Using Jython list: `AdminTask.listServerPorts (['-interactive'])`
modify Server Port	Use the **modifyServer Port** command to modify the port that is used by the server.	The server name.	• Parameters: **- nodeName** The name of the node. (String, required) **- portName** The name of the port. (String, required) **- portNumber** The port number. (Integer, required) • Returns: The ports that are used by the server that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyServerPort server1 {-nodeName myNode -portName port1 -portNumber 5566}` • Using Jython string: `AdminTask.modifyServerPort ('server1', '[-nodeName myNode -portName port1 -portNumber 5566]')` • Using Jython list: `AdminTask.modifyServerPort ('server1', ['-nodeName', 'myNode', '-portName', 'port1', '-portNumber', '5566'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyServerPort {-interactive}` • Using Jython string: `AdminTask.modifyServerPort ('[-interactive]')` • Using Jython list: `AdminTask.modifyServerPort (['-interactive'])`

Commands for the VariableConfiguration group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the VariableConfiguration group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
remove Variable	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.	None	• Parameters: **- 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` **- nodeName** The name of the node. This parameter is only needed for server scopes that do not have unique name across nodes. (String, optional) • Returns: None	**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.	None	• Parameters: **- 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` . **- nodeName** 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) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setVariable {-variableName varname1 -scopeName Cell=localhost Node01Cell,Node= localhostNode01}` • Using Jython string: `AdminTask.setVariable('[-variableName varname1 -scopeName Cell=local hostNode01Cell,Node= localhostNode01]')` • Using Jython list: `AdminTask.setVariable (['-variableName', 'varname1', '-scopeName', 'Cell=localhostNode 01Cell,Node=local hostNode01'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setVariable {-interactive}` • Using Jython string: `AdminTask.setVariable ('[-interactive]')` • Using Jython list: `AdminTask.setVariable (['-interactive'])`

show Variables	Use the **show Variables** command to list variable values under a scope.	None	• Parameters: **- scopeName** The name of the scope. This parameter is optional if the scope name is unique in scope. (String, required) **- scope** The scope of the variable definition. The default is Cell. Valid values include: Cell, Node, Server, Application, and Cluster. (String, optional) **- nodeName** 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) • Returns: The variable values in list format. Each element in the list is a variable name and value pair.	**Interactive mode example usage:** • Using Jacl: `$AdminTask showVariables {-interactive}` • Using Jython string: `AdminTask.showVariables ('[-interactive]')` • Using Jython list: `AdminTask.showVariables (['-interactive'])`

Configuring connections to Webservers with scripting

Use these topics to learn about configuring a connection to Webservers using scripting and the wsadmin tool.

This topic contains the following tasks:

Regenerating the node plug-in configuration using scripting

You can use scripting and the wsadmin tool to regenerate the node plug-in configuration.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to regenerate the node plug-in configuration:

1. Identify the plug-in and assign it to the `generator` variable, for example:

   Using Jacl:

   ```
   set generator [$AdminControl completeObjectName type=PluginCfgGenerator,node=mynode,*]
   ```

   Using Jython:

   ```
   generator = AdminControl.completeObjectName('type=PluginCfgGenerator,node=mynode,*')
   ```

2. Regenerate the node plug-in:

   Using Jacl:

   ```
   $AdminControl invoke $generator generate "app_server_root/config
   mycell mynode null plugin-cfg.xml"
   ```

   Using Jython:

   ```
   AdminControl.invoke(generator, 'generate', "app_server_root/config
   mycell mynode null plugin-cfg.xml")
   ```

Creating new virtual hosts using templates with scripting

Use scripting to create a new virtual host from a new or preexisting template.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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:

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, restart the deployment manager on a network deployment edition, or 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 “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Managing servers with scripting

Use these topics to learn about managing servers with scripting.

This topic contains the following tasks:

Stopping a node using scripting

Use scripting to stop a node agent.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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:

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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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 number of seconds that the process should wait before starting the server.
- 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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to query the server state:

1. 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.completObjectName('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
```

2. Query for the 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 scripting

Use the wsadmin tool and scripting to list all the running applications on all the running servers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Use the following example to list all the running applications on all the running servers on each node of each cell:

- 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"
```

Example output:

```
Number of running servers on node mynode: 2
mynode (NodeAgent) has pid 3592; state: STARTED; on Windows 2000
Number of applications running on mynode: 0
-----------------------------------------------------

server1 (ManagedProcess) has pid 3972; state: STARTED; on Windows 2000
Number of applications running on server1: 0
-----------------------------------------------------

Number of running servers on node mynodeManager: 1
dmgr (DeploymentManager) has pid 3308; state: STARTED; on Windows 2000
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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to start a listener port on an application server. The following example returns a list of listener port MBeans:

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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

A generic server is a server that the WebSphere Application Server manages but did not supply.

- To define a generic server, use the following example:
  - Using Jacl:

```
$AdminTask createGenericServer mynode {-name generic1 -ConfigProcDef
{{"/usr/bin/myStartCommand" "arg1 arg2" "" "" "/tmp/workingDirectory"
"/tmp/stopCommand" "argy argz"}}}
$AdminConfig save
```

  - Using Jython:

```
AdminTask.createGenericServer('mynode', '[-name generic1 -ConfigProcDef
[[\tmp\myStartCommand.exe "a b c" "" "" \tmp\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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to set the development mode for a server object:

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 “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Disabling parallel startup using scripting

You can use scripting to disable parallel startup of servers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to disable parallel startup:

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 “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Obtaining server version information with scripting

Use the wsadmin tool and scripting to obtain server version information.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to query the server version information:

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 $server1 serverVersion
     ```

   - Using Jython:

     ```
     print AdminControl.getAttribute(server1, 'serverVersion')
     ```

   Example output for a Network Deployment installation follows:

   ```
   IBM WebSphere Application Server Version Report
      ------------------------------------------------------------------------

           Platform Information
           ------------------------------------------------------------------------

                   Name: IBM WebSphere Application Server
                   Version: 5.0


           Product Information
           ------------------------------------------------------------------------

                   ID: BASE
                   Name: IBM WebSphere Application Server
                   Build Date: 9/11/02
                   Build Level: r0236.11
                   Version: 5.0.0


           Product Information
           ------------------------------------------------------------------------

                   ID: ND
                   Name: IBM WebSphere Application Server for Network Deployment
                   Build Date: 9/11/02
   ```

```
            Build Level: r0236.11
            Version: 5.0.0

--------------------------------------------------------------------------------
End Report
--------------------------------------------------------------------------------
```

Commands for the NodeGroupCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the NodeGroupCommands group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
addNode Group Member	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. • z/OS platform from different sysplexs are not combined into the same node group.	The target object is the node group where the member will be created. This target object is required.	• Parameters: **- nodeName** The name of the node that you want to add to a node group. This parameter is required. • Returns: Node group member object ID	**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'])`

createNode Group	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.	The node group name to be created. This target object is required.	• Parameters: **- shortName** The short name of the node group. This parameter is optional. **- description** The description of the node group. This parameter is optional. • Returns: The node group object ID.	**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'])`
createNode Group Property	The **createNode Group Property** command creates custom properties for a node group.	The name of the node group. This target object is required.	• Parameters: **- 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. • Returns: The properties object ID.	**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'])`

listNode Group Properties	The **listNode Group Properties** command displays all of the custom properties of a node group.	The target object is name of the node group. This target object is required.	• Parameters: None • Returns: A list of all of the custom properties of a node group.	**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'])`
listNode Groups	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.	The target object is name of the node. This target object is optional.	• Parameters: None • Returns: A list of the node groups in the cell.	**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.	The target object is name of the node group. This target object is optional.	• Parameters: None • Returns: A list of all the nodes in the cell	**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'])`
modify Node Group	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.	The target object is the node group name. This target object is required.	• Parameters: **- shortName** The short name of the node group. This parameter is optional. **- description** The description of the node group. This parameter is optional. • Returns: Node group object ID.	**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'])`

modify Node Group Property	The **modify Node Group Property** command modifies custom properties for a node group	The name of the node group. This target object is required.	• Parameters: **- 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. • Returns: Properties object ID	**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'])`
remove Node Group	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.	The name of the node group to be removed. This target object is required.	• Parameters: None • Returns: The node group object ID.	**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'])`

removeNode Group Member	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. • For the z/OS platform, you cannot remove nodes from sysplex node groups.	The target object is the node group containing the member to be removed. This target object is required.	• Parameters: **- nodeName** The name of the node to remove from a node group. This parameter is required. • Returns: Node group member object ID.	**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'])`

removeNode Group Property	The **removeNode Group Property** command removes custom properties of a node group.	The name of the node group. This target object is required.	• Parameters: **- name** The name of the custom property to remove. This parameter is required. • Returns: Properties object ID	**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'])`

Commands for the Utility group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the Utility group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
changeHost Name	Use the **changeHost Name** command to change the host name of a node.	None	• Parameters: **- hostName** The new host name. (String, required) **- nodeName** The name of the node whose host name will be changed. (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask changeHostName {-interactive}` • Using Jython string: `AdminTask.changeHostName ('[-interactive]')` • Using Jython list: `AdminTask.changeHostName (['-interactive'])`

getDmgr Properties	Use the **getDmgr Properties** command to return the name of the deployment manager.	None	• 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.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getDmgrProperties {-interactive}` • Using Jython string: `AdminTask.getDmgrProperties ('[-interactive]')` • Using Jython list: `AdminTask.getDmgrProperties (['-interactive'])`
isFederated	Use the **isFederated** command to check if the system is a single server or network deployment.	None	• Parameters: None • Returns: Boolean. `true` if the system is a network deployment system. Otherwise it returns `false`.	**Interactive mode example usage:** • Using Jacl: `$AdminTask isFederated {-interactive}` • Using Jython string: `AdminTask.isFederated ('[-interactive]')` • Using Jython list: `AdminTask.isFederated (['-interactive'])`

Commands for the ManagedObjectMetadata group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the ManagedObjectMetadata group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

compareNode Version	The **compareNode Version** command compares the WebSphere Application Server version given a node that you specify and an input version.	None	• Parameters: **- 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. • Returns: – 0 if node version matches the input version – -1 if node version is smaller than the input version – 1 is node version is higher than the input version	**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'])`
getMetadata Properties	The **getMetadata Properties** command obtains all metadata for the node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: The list of metadata properties.	**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 **getMetadataProperty** command obtains metadata with the specified key for the node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. **- propertyName** Metadata property key. • Returns: The requested property for the node that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getMetadataProperty {-nodeName node1 -propertyName com.ibm.websphere.baseProductVersion}` • Using Jython string: `AdminTask.getMetadataProperty ('[-nodeName node1 -propertyName com.ibm.websphere.baseProductVersion]')` • Using Jython list: `AdminTask.getMetadataProperty (['-nodeName', 'node1', '-propertyName', 'com.ibm.websphere.baseProductVersion'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getMetadataProperty {-interactive}` • Using Jython string: `AdminTask.getMetadataProperty ('[-interactive]')` • Using Jython list: `AdminTask.getMetadataProperty (['-interactive'])`
getNodeBaseProductVersion	The **getNodeBaseProductVersion** command returns the version of the WebSphere Application Server for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: WebSphere Application Server version for the node that you specify.	**Batch mode example usage:** • Using Jacl: `$AdminTask getNodeBaseProductVersion {-nodeName node1}` • Using Jython string: `AdminTask.getNodeBaseProductVersion('[-nodeName node1]')` • Using Jython list: `AdminTask.getNodeBaseProductVersion(['-nodeName', 'node1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getNodeBaseProductVersion {-interactive}` • Using Jython string: `AdminTask.getNodeBaseProductVersion ('[-interactive]')` • Using Jython list: `AdminTask.getNodeBaseProductVersion (['-interactive'])`

getNode Major Version	The **getNode Major Version** command returns the minor version of the WebSphere Application Server for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: WebSphere Application Server minor version for the node that you specified.	**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'])`
getNode Minor Version	The **getNode Minor Version** command returns the minor version of the WebSphere Application Server for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: WebSphere Application Server minor version for the node that you specified.	**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'])`

getNode Platform OS	The **getNode Platform OS** command returns the operating system name for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: The operating system name of the node that you specified.	**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'])`
getNode Sysplex Name	The **getNode Sysplex Name** command returns the sysplex name for a node that you specify.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: The sysplex name of the given node.	**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'])`

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.	None	• Parameters: **- nodeName** The name of the node associated with the metadata you want this command to return. • Returns: A `true` value if the node operating system is z/OS. A `false` value if the node operating system is not z/OS.	**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'])`

Commands for the ServerManagement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the ServerManagement group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

create Application Server	Use the **create Application Server** command to create a new application server.	Node name (optional)	• Parameters: **- name** The name of the server that you want to create. (String) **- templateName** The name of the template from which to base the server. (String) **- genUniquePorts** Specifies that unique ports should be created for the server. (boolean) **- templateLocation** The location of a template. (ObjectName) • Returns: The configuration ID of the server you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createApplication Server ndnode1 {-name test1 –templateName default}` • Using Jython string: `AdminTask.createApplication Server(ndnode1, '[-name test1 -templateName default]')` • Using Jython list: `AdminTask.createApplication Server(ndnode1, ['-name', 'test1', '-templateName', 'default'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createApplication Server {-interactive}` • Using Jython string: `AdminTask.createApplication Server ('[-interactive]')` • Using Jython list: `AdminTask.createApplication Server (['-interactive'])`

create Application Server Template	The **create Application Server Template** command creates a new application server template.	None	• Parameters: **- templateName** The name of the application server template that you want to create. (String) **- serverName** The name of the server from which to base the template. (String) **- nodeName** The node that corresponds to the server from which to base the template. (String) **- description** The description of the template. (String) **- templateLocation** The location where you want to place the template. (String) • Returns: The configuration ID of a new template.	**Batch mode example usage:** • Using Jacl: `$AdminTask createApplication ServerTemplate {-templateName newTemplate -serverName server1 -nodeName ndnode1 -description "This is my new template"}` • Using Jython string: `AdminTask.createApplication ServerTemplate('[-templateName newTemplate -serverName server1 -nodeName ndnode1 -description "This is my new template"]')` • Using Jython list: `AdminTask.createApplication ServerTemplate(['-templateName', 'newTemplate', '-serverName', 'server1', '-nodeName', 'ndnode1', '-description', "This is my new template"])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createApplication ServerTemplate {-interactive}` • Using Jython string: `AdminTask.createApplication ServerTemplate ('[-interactive]')` • Using Jython list: `AdminTask.createApplication ServerTemplate (['-interactive'])`

create Generic Server	Use the **create Generic Server** 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 **create Generic Server** command provides an additional step, ConfigProcDef, that you can use to configure the parameters that are specific to generic servers.	None	• Parameters: **- name** The name of the server that you want to create. **- 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** The port for the server. **- templateLocation** The location of the server template. **- 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)	**Batch mode example usage:** • Using Jacl: `$AdminTask createGeneric Server jim667BaseNode {-name jgeneric -ConfigProcDef {{"/usr/bin/myStartCommand" "arg1 arg2" "" "" "/tmp/ workingDirectory" "/tmp/ stopCommand" "argy argz"}}}` • Using Jython string: `AdminTask.createGeneric Server('jim667BaseNode', '[-name jgeneric -Config ProcDef [[/usr/bin/my StartCommand "arg1 arg2" "" "" /tmp/working Directory /tmp/StopCommand "argy argz"]]]')` • Using Jython list: `AdminTask.createGeneric Server('jim667BaseNode', ['-name', 'jgeneric', '-ConfigProcDef', [['/usr/ bin/myStartCommand', "arg1 arg2" "" "", '/tmp/working Directory', '/tmp/Stop Command', "argy argz"]]])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createGeneric Server {-interactive}` • Using Jython string: `AdminTask.createGeneric Server ('[-interactive]')` • Using Jython list: `AdminTask.createGeneric Server (['-interactive'])`

			- 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) • Returns: None	

createWeb Server	Use the **createWeb Server** command to create a Web server definition. This command is a two-step process. The first step creates a Web server definition using a template. The parameters of the second step configure 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 the Web server definitions allow you to administer and configure IBM HTTP Server Web servers using the administrative console. These functions include: Start, Stop, View logs, View and Edit configuration files.	None	• **Parameters for step one:** **nodeName** The name of the node. (String, required) **name** The name of the server. (String, required) **templateName** 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** Indicates that you want to generate unique ports. (optional) • **Parameters for step two:** **serverConfig** Create the Web server. (String, required) **webPort** The port for the Web server. (String, required) **configurationFile** The configuration file. The default is the path relative to the installation root, for example, `conf/httpd.conf`. (String, optional) **webInstallRoot** The installation path for the Web server. (String, required) **pluginInstallRoot** The plug-in installation path. (String, required) **serviceName** The service name. (String, optional) **errorLogfile** The error log for viewing. The default is the path relative to the installation root, for example, `logs/error_log`. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createWebServer {-name web1 -serverConfig {{webPort Webserver InstallRoot PluginInstallRoot Configuration_file_name Windows_Server_Name errorLogPath accessLogPath WebProtocol}} -remoteServerConfig {{AdminPort UserID Password AdminProtocol}}` • Using Jython list: `AdminTask.createWebServer(['-name', 'web1', '-serverConfig', [['webPort', 'WebserverInstall Root', 'PluginInstallRoot', 'Configuration_file_name', 'Windows_Server_Name', 'errorLog Path', 'accessLogPath', 'Web Protocol']], '-remoteServerConfig', [['AdminPort', 'UserID', 'Password', 'AdminProtocol']]])` • Using Jython string: `AdminTask.createWebServer('[-name web1 -serverConfig [[webPort WebserverInstallRoot PluginInstallRoot Configuration_ file_name Windows_Server_Name errorLogPath accessLogPath WebProtocol]] -remoteServerConfig [[AdminPort UserID Password AdminProtocol]]]')` where `-serverConfig` is second step of the command. – WebPort - is the port for the Webserver (required for all webservers) – WebserverInstallRoot - is the install path (directory) for webserver. necessary for IBM HTTP Server Admin Function. – Plugin Install Root - is install root where the plugin for the webserver is installed. Necessary for all webservers. – Configuration file name - is the file path for the IBM HTTP Server. This is necessary for View and edit of the IBM HTTP Server Configuration file only. – Windows Service Name - The windows service name on which IBM HTTP Server is to be started. This is necessary for Start and stop of the IBM HTTP Server Web server only. – ErrorLogPath - This is the path for the IBM HTTP Server error log (error.log)

			accessLogfile The access log for viewing. The default is the path relative to the installation root, for example, `logs/access_log`. (String, optional) **webProtocol** Parameters for 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) **adminPort** The port of the IBM HTTP Server administrative server. (String, required) **adminUserID** The user ID. This value should match the one for authentication in the `admin.conf`. (String, required) **adminPasswrd** The administrative password. (String, required) **adminProtocol** The administrative protocol title. Options include `HTTP` or `HTTPS`. The default is `HTTP`. (String, required)	• AccessLogPath - This is the path for the IBM HTTP Server access log (access.log) • WebServerProtocol - HTTP or HTTPS where -remoteServerConfig is 3rd step of the command These parameters are only necessary if the IBM HTTP Server Web server is installed on a machine remote from WebSphere. • Admin Server Port - This is the port for the ADministration server. The administration server is installed on the same machine as the IBM HTTP Server. The administrative server handles administrative requests to the IBM HTTP Server Web server. • UserID - This is the userID for authentication, if authentication is activated on the Administration server in the admin configuration file (admin.conf). • Passwd - This is the password for the specified authentication UserID. The password is generated by htpasswd utility in the admin.passwd file. • Admin ServerProtocol - HTTP or HTTPS **Interactive mode example usage:** • Using Jacl: `$AdminTask createWebServer -interactive` • Using Jython string: `AdminTask.createWebServer ('[-interactive]')` • Using Jython list: `AdminTask.createWebServer (['-interactive'])`

			• **Parameters for step three:** Parameters for IBM HTTP Server administration server running with an unmanaged or remote Web server (installed on machine different from WebSphere Application Server) **adminPortTitle (adminPort)** Port of IBM HTTP Server administration **adminUserIDTitle (adminUserID)** The user ID. This value should match the authentication in the admin.conf file. **adminPasswdTitle (adminPasswd)** password **AdminProtocolTitle (adminProtocol)** This parameter is required. The value is either HTTP or HTTPS. The default value is HTTP. • **Returns:** None	

deleteServer	Use the **deleteServer** command to delete a server.	None	• 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) • Returns: None	**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'])`

delete Server Template	Use the **delete Server Template** 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.	The name of the template to delete. (ObjectName, required)	• Parameters: • Returns: None	**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'])`

getServer Type	The **getServer Type** command returns the type of the server that you specify.	None	• Parameters: **- serverName** The name of the server. (String) **- nodeName** The name of the node. (String) • Returns: The type of the server.	**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.	None	• Parameters: **serverType** The type of the server. Used to filter the results. (String) **nodeName** The name of the node. Used to filter the results. (String) • Returns: A list of configuration IDs for the servers that match the criteria that you specified.	**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'])`

listServer Templates	Use the **listServer Templates** command to list server templates.	None	• 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/value pair that you can use to find templates by properties. For example, `com.ibm.websphere.nodeOperatingSystem=os390` (String[], optional) • Returns: A list of server template identifications that match with the criteria that you specify with the command parameters. If you do no specify any parameters, all server templates are returned.	**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'])`

listServer Types	Use the **listServer Types** command to display all the current server types. For example, `APPLICATION_ SERVER, WEB_ SERVER, GENERIC_ SERVER`	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: 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.	**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'])`
setJVM Debug Mode	Use the **setJVM Debug Mode** command to set the Java virtual machine (JVM) debug mode for the application server.	None	• 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) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask setJVMDebugMode {-interactive}` • Using Jython string: `AdminTask.setJVMDebugMode ('[-interactive]')` • Using Jython list: `AdminTask.setJVMDebugMode (['-interactive'])`

setJVM Generic JVM Arguments	Use the **setJVM Generic JVM Arguments** command to set the Java virtual machine (JVM) debug mode for the application server.	None	• Parameters: **- serverName** 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** The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required) **- processType** The process type of the server. Valid values include: Control, Servant, or Adjunct. (String, optional) **- genericJvmArguments** Specifies that the command line arguments pass to the Java virtual machine code that starts the application server process. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setJVMGenericJVM Arguments {-serverName server1 -nodeName node1}` • Using Jython string: `AdminTask.setJVMGenericJVM Arguments('[-serverName server1 -nodeName node1]')` • Using Jython list: `AdminTask.setJVMGenericJVM Arguments(['-serverName', 'server1', '-nodeName', 'node1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setJVMGenericJVM Arguments {-interactive}` • Using Jython string: `AdminTask.setJVMGenericJVM Arguments ('[-interactive]')` • Using Jython list: `AdminTask.setJVMGenericJVM Arguments (['-interactive'])`

setJVM Initial HeapSize	Use the **setJVM Initial HeapSize** command to set the Java Virtual Machine (JVM) initial heap size for the application server.	None	• Parameters: **- 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) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask setJVMInitialHeapSize {-interactive}` • Using Jython string: `AdminTask.setJVMInitialHeapSize ('[-interactive]')` • Using Jython list: `AdminTask.setJVMInitialHeapSize (['-interactive'])`
setJVM MaxHeap Size	Use the **setJVM MaxHeap Size** command to set the Java virtual machine (JVM) maximum heap size for the application server.	None	• Parameters: **- serverName** The name of the server whose JVM properties are modified. If there is only one server in the configuration, (String, required) **- nodeName** The node name where the server locates. If the server name is unique in the cell, this parameter is optional. (String, required) **- maximumHeapSize** Specifies the maximum heap size available to the JVM code, in megabytes. (Integer, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setJVMMaxHeapSize {-serverName server1 -nodeName node1 -maximum HeapSize 10}` • Using Jython string: `AdminTask.setJVMMaxHeapSize('[-serverName server1 -nodeName node1 -maximum HeapSize 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'])`

setJVM Properties	Use the **setJVM Properties** command to set the Java virtual machine (JVM) configuration for the application server.	None	• Parameters: **- serverName** 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** The node name where the server resides. If the server name is unique in the entire cell, this parameter is optional. (String, required) **- classpath** 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)	**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'])`

			- 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) • Returns: None	

setJVM System Properties	Use the **setJVM System Properties** command to set the Java virtual machine (JVM) system property for the process of the application server.	None	• 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** The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required) **- propertyName** The property name. (String, required) **- propertyValue** The property value. (String, required) • Returns: None	**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'])`

setProcess Definition	Use the **setProcess Definition** command to set the process definition of an application server.	None	• 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) **- executableName** The executable name that is invoked to start the process. This parameter is only applicable to WebSphere Application Server version. (String, optional) **- executableArguments** 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** The file system directory that the process uses for the current working directory. (String, optional) **- executableTargetKind** The type of the executable target. Valid values include JAVA_CLASS and EXECUTABLE JAR. (String, optional) **- executableTarget** 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) • Returns: None	**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'])`

setTrace Specification	Use the **setTrace Specification** 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.	None	• 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** The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required) **- traceSpecification** The trace specification. (String, required) • Returns: None	**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'])`

showJVM Properties	Use the **showJVM Properties** command to list the Java virtual machine (JVM) configuration for the server of the application process.	None	• Parameters: **- serverName** 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** 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, required) • Returns: The current JVM properties of the application server in list format. Each element in the list is a property name and value pair.	**Batch mode example usage:** • Using Jacl: `$AdminTask showJVMProperties {-serverName server1 -nodeName node1 -property Name test.property}` • Using Jython string: `AdminTask.showJVMProperties('[-serverName server1 -nodeName node1 -property Name test.property]')` • Using Jython list: `AdminTask.showJVMProperties(['-serverName', 'server1', '-nodeName', 'node1', '-propertyName', 'test.property'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask showJVMProperties {-interactive}` • Using Jython string: `AdminTask.showJVMProperties ('[-interactive]')` • Using Jython list: `AdminTask.showJVMProperties (['-interactive'])`

showJVM System Properties	Use the **showJVM System Properties** command to show the Java virtual machine (JVM) system properties for the process of the application server.	None	• 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** The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required) **- 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) • Returns: The current JVM properties for the process of the application server in list format. Each element in the list is a property name and value pair.	**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'])`

show Process Definition	Use the **show Process Definition** command to show the process definition of the server.	None	• Parameters: **- serverName** 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** The node name where the server resides. If the server name is unique in the cell, this parameter is optional. (String, required) **- 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) • Returns: The current process definition of the server in list format. Each element in the list is a property name and value pair.	**Batch mode example usage:** • Using Jacl: `$AdminTask showProcessDefinition {-serverName server1 -nodeName node1 -propertyName test.property}` • Using Jython string: `AdminTask.showProcessDefinition ('[-serverName server1 -nodeName node1 -property Name test.property]')` • Using Jython list: `AdminTask.showProcessDefinition (['-serverName', 'server1', '-nodeName', 'node1', '-propertyName', 'test. property'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask showProcessDefinition {-interactive}` • Using Jython string: `AdminTask.showProcessDefinition ('[-interactive]')` • Using Jython list: `AdminTask.showProcessDefinition (['-interactive'])`

show Server Info	The **show Server Info** command returns the information for a server that you specify.	The configuration ID of the server. (required)	• Parameters: None • Returns: A list of metadata.	**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'])`

showServer TypeInfo	The **showServer TypeInfo** command displays information about a specific server type.	A server type. For example: `APPLIC ATION _SERVER`(String, required)	• Parameters: **- version** Specify the version of the templates that you want to list. For example, 6.0.0.0. (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) • Returns: A list of information about the server type.	**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'])`

show Template Info	Use the **show Template Info** command to display the metadata information for a specific server template.	The server type, for example: `APPLICATION_SERVER` (String, required)	• Parameters: None • Returns: The metadata information regarding a specific template.	**Batch mode example usage:** • Using Jacl: `$AdminTask showTemplateInfo default(templates/servertypes /APPLICATION_SERVER/servers/ default\|server.xml) {isSystem Template true} {name default} {com.ibm.websphere.baseProd uctVersion 6.0.0} {description {The WebSphere Default Server Template}} {com.ibm.websphere. baseProductMinorVersion 0.0} {com.ibm.websphere.baseProduct MajorVersion 6} {com.ibm. websphere.nodeOperatingSystem {}} {isDefaultTemplate true}` • Using Jython string: `AdminTask.showTemplateInfo (default(templates/serverTypes /APPLICATION_SERVER/servers/ default\|server.xml)) '[[is SystemTemplate true] [com. ibm.websphere.baseProduct Version 6.0.0] [name default] [com.ibm.websphere.basePro ductMinorVersion 0.0] [description The WebSphere Default Server Template] [isDefaultTemplate true] [com.ibm.websphere.node OperatingSystem] [com.ibm. websphere.baseProductMajor Version 6]]'` • Using Jython list: `AdminTask.showTemplateInfo (default(templates/server Types/APPLICATION_SERVER/ servers/default\|server.xml)) [['isSystemTemplate', 'true'], ['com.ibm.websphere.base ProductVersion', '6.0.0'], ['name', 'default'] ['com. ibm.websphere.baseProduct MinorVersion', '0.0'], ['description', 'The WebSphere Default Server Template'] ['isDefaultTemplate', 'true'], ['com.ibm.websphere.node OperatingSystem'], ['com. ibm.websphere.baseProduct MajorVersion', '6']]` **Interactive mode example usage:** • Using Jacl: `$AdminTask showTemplateInfo {-interactive}` • Using Jython string: `AdminTask.showTemplateInfo ('[-interactive]')` • Using Jython list: `AdminTask.showTemplateInfo (['-interactive'])`

Commands for the UnmanagedNodeCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the UnmanagedNodeCommands group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
create Unmanaged Node	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.	None	• Parameters: **- 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) • Returns: null	**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'])`
listManaged Nodes	Use the **listManaged Nodes** command to list the managed nodes, nodes that have a node agent defined, in a configuration.	None	• Parameters: None • Returns: List	**Batch mode example usage:** • Using Jacl: `$AdminTask listManagedNodes` • Using Jython string: `AdminTask.listManagedNodes()` • Using Jython list: `AdminTask.listManagedNodes()`

list Unmanaged Nodes	Use the **list Unmanaged Nodes** command to list the unmanaged nodes in a configuration.	None	• Parameters: None • Returns: List	**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'])`
remove Unmanaged Node	Use the **remove Unmanaged Node** command to remove an unmanaged node from the configuration.	None	• Parameters: **- nodeName** The name of the unmanaged node. (String, required) • Returns: null	**Batch mode example usage:** • Using Jacl: `$AdminTask removeUnmanaged Node {-nodeName myNode }` • Using Jython string: `AdminTask.removeUnmanaged Node('[-nodeName myNode]')` • Using Jython list: `AdminTask.removeUnmanaged Node(['-nodeName', 'myNode'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeUnmanaged Node {-interactive}` • Using Jython string: `AdminTask.createUnmanaged Node ('[-interactive]')` • Using Jython list: `AdminTask.createUnmanaged Node (['-interactive'])`

Commands for the ConfigurationArchiveOperations group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the ConfigurationArchiveOperations group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
exportServer	Use the **exportServer** command to export the server configuration to a node that is defined in the configuration archive. 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 membership.	None	• Parameters: **-archive** The fully qualified path of the exported configuration archive. (String, required) **-nodeName** The node name of the server. This parameter is only required when the server name is not unique across the cell. (String, optional) **-serverName** The server name. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask exportServer {-archive c:\myServer.ear -nodeName node1 -serverName server1}` • Using Jython string: `AdminTask.exportServer ('[-archive c:\myServer.ear -nodeName node1 -serverName server1]')` • Using Jython list: `AdminTask.exportServer (['-archive', 'c:\ myServer.ear', '-nodeName', 'node1', '-serverName', 'server1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask exportServer {-interactive}` • Using Jython string: `AdminTask.exportServer ('[-interactive]')` • Using Jython list: `AdminTask.exportServer (['-interactive'])`

	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.			

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.	None	• Parameters: **-archive** The fully qualified path of the configuration archive. (String, required) **-nodeInArchive** 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** 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** The node name where the server is imported. (String, optional if there is only one node) **-serverName** 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** The core group name to which the server should belong. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask importServer {-archive c:\myServer.ear -nodeInArchive node1 -serverInArchive server1}` • Using Jython string: `AdminTask.importServer ('[-archive c:\myServer.ear -nodeInArchive node1 -serverInArchive server1]')` • Using Jython list: `AdminTask.importServer (['-archive', 'c:\myServer.ear', '-nodeInArchive', 'node1', '-serverInArchive', 'server1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask importServer {-interactive}` • Using Jython string: `AdminTask.importServer ('[-interactive]')` • Using Jython list: `AdminTask.importServer (['-interactive'])`

Clustering servers with scripting

You can use scripting and the wsadmin tool to cluster servers.

This topic contains the following tasks:

- “Creating clusters using scripting”
- “Modifying cluster member templates using scripting”
- “Creating cluster members using scripting” on page 276
- “Creating clusters without cluster members using scripting” on page 278
- “Starting clusters using scripting” on page 278
- “Querying cluster state using scripting” on page 279
- “Stopping clusters using scripting” on page 280

Creating clusters using scripting

You can create clusters using scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to create a cluster:

1. Identify the server to convert to a cluster 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/')
   ```

2. Convert the existing server to a cluster by using the **convertToCluster** command passing in the existing server and the cluster name:

   Using Jacl:

   ```
   $AdminConfig convertToCluster $server myCluster1
   ```

   This command converts a cluster named myCluster with server1 as its member.

   Using Jython:

   ```
   print AdminConfig.convertToCluster(server, 'myCluster1')
   ```

   Example output:

   ```
   myCluster1(cells/mycell/cluster/myCluster1|cluster.xml#ClusterMember_1)
   ```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Modifying cluster member templates using scripting

You can create cluster members using the AdminConfig object and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

A copy of the first cluster member that you create is stored in the cluster scope as a template. You can create the first cluster member using any existing server as a template or a default server template. You

can also create a first cluster member when you create the cluster by converting a server to a cluster. When you create a first cluster member, the template of the cluster member is stored under the scope of the cluster. Additional cluster members are created using the cluster member template stored in the cluster scope.

A cluster can be either homogeneous or heterogeneous in nature. A homogeneous cluster spans nodes that are of the same WebSphere Application Server version. A heterogeneous cluster spans nodes of different WebSphere Application Server versions. Since a cluster can contain members from nodes that run on different versions of WebSphere Application Server, one template will be stored for each version of the WebSphere Application Server node that is configured as a cluster member. The cluster member template will not exist for a given node version until you create a first member in a node of the same version. For example, if a cluster contains several V6.1 nodes and several V6 nodes, there will be one cluster member template for the V6.1 node and one for the V6 node, such as the following:

- The *$WAS_HOME*/config/clusters/clusterName/servers/V6.1MemberTemplate template will be used as the template for any member that is created in a V6.1 node.
- The *$WAS_HOME*/config/clusters/clusterName/servers/V6MemberTemplate template will be used as the template for any member that is created in a V6 node.
- The *$WAS_HOME*/config/clusters/clusterName/servers/V5MemberTemplate template will be used as the template for any member that is created in a V5 node.

Therefore, when you make a configuration change to cluster members, you should make the same configuration change to the corresponding template that is stored in the cluster scope in order to keep the template in synch with the existing members. Similarly, when you make a configuration change to the template, you should make the same configuration change to existing cluster members.

You can modify a cluster member template using the wsadmin tool similar to how you modify a server. You cannot modify a cluster member template using the administrative console. Perform the following steps to modify a cluster member template using the wsadmin tool:

1. Obtain the cluster template under the cluster scope. For example, the following example obtains the version 6.1 cluster member template for the cluster1 cluster:
   - Using Jacl:

     ```
     set c [$AdminConfig getid /ServerCluster:Cluster1]

     set s [$AdminConfig list Server $c]
     ```

     Using Jython:

     ```
     c = AdminConfig.listTemplates('Server','cluster1/servers/V6.1')

     print AdminConfig.showall(c)
     ```
2. Modify the attributes of the template. For example:
   - Using Jacl:

     ```
     $AdminConfig modify $s {{attrName attrVal}}
     ```

     Using Jython:

     ```
     AdminConfig.modify(c, [[attrName, attrval]])
     ```
3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Creating cluster members using scripting

You can create cluster members using the AdminConfig object and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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.

A template is stored in the cluster scope that you must use to create additional cluster members. For more information about this template, see the “Modifying cluster member templates using scripting” on page 275 article.

To create cluster members using the AdminConfig object, perform the following steps:

1. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:
       ```
       $AdminTask createClusterMember {-interactive}
       ```
     - Using Jython:
       ```
       AdminTask.createClusterMember ('[-interactive]')
       ```
   - Using the AdminConfig object:
     a. Identify the existing cluster and assign it to the cluster variable:
        - Using Jacl:
          ```
          set cluster [$AdminConfig getid /ServerCluster:myCluster1/]
          ```
        - Using Jython:
          ```
          cluster = AdminConfig.getid('/ServerCluster:myCluster1/')
          print cluster
          ```
          Example output:
          ```
          myCluster1(cells/mycell/cluster/myCluster1|cluster.xml#ServerCluster_1)
          ```
     b. Identify the node to create the new server and assign it to the node variable:
        - Using Jacl:
          ```
          set node [$AdminConfig getid /Node:mynode/]
          ```
        - Using Jython:
          ```
          node = AdminConfig.getid('/Node:mynode/')
          print node
          ```
          Example output:
          ```
          mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
          ```
     c. (Optional) Identify the cluster member template and assign it to the serverTemplate variable:
        - Using Jacl:
          ```
          set serverTemplate [$AdminConfig listTemplates Server]
          ```
        - Using Jython:
          ```
          serverTemplate = AdminConfig.listTemplates('Server')
          print serverTemplate
          ```
          Example output:
          ```
          server1(templates/default/nodes/servers/server1|server.xml#Server_1)
          ```
     d. Create the new cluster member, by using the **createClusterMember** command.
        - The following example creates the new cluster member, passing in the existing cluster configuration ID, existing node configuration ID, and the new member attributes:
          - Using Jacl:
            ```
            $AdminConfig createClusterMember $cluster $node {{memberName clusterMember1}}
            ```
          - Using Jython:
            ```
            AdminConfig.createClusterMember(cluster, node, [['memberName', 'clusterMember1']])
            ```
        - The following example creates the new cluster member with a template, passing in the existing cluster configuration ID, existing node configuration ID, the new member attributes, and the template ID:

- Using Jacl:

```
$AdminConfig createClusterMember $cluster $node
{{memberName clusterMember1}} $serverTemplate
```

- Using Jython:

```
print AdminConfig.createClusterMember(cluster, node,
[['memberName',  'clusterMember1']], serverTemplate)
```

Example output:

```
clusterMember1(cells/mycell/clusters/myCluster1|cluster.xml$ClusterMember_2)
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Creating clusters without cluster members using scripting

You can use scripting to create clusters without cluster members.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to create a cluster without a cluster member:

1. There are two ways to perform this task. Choose one of the following:
   - Using the AdminTask object:
     - Using Jacl:

```
$AdminTask createCluster {-interactive}
```

     - Using Jython:

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

   - Using the AdminConfig object:
     a. Identify the cell configuration ID and set it to the s1 variable:
        - Using Jacl:

```
set s1 [$AdminConfig getid /Cell:mycell/]
```

        - Using Jython:

```
s1 = AdminConfig.getid('/Cell:mycell/')
```

     b. Create a new cluster without a cluster member:
        - Using Jacl:

```
$AdminConfig create ServerCluster $s1 {{name ClusterName}}
```

        - Using Jython:

```
print AdminConfig.create('ServerCluster', s1, '[[name ClusterName]]')
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Starting clusters using scripting

You can use scripting and the wsadmin tool to start clusters in an application server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to start a cluster:

1. Identify the ClusterMgr MBean and assign it to the clusterMgr variable.
   - Using Jacl:
     ```
     set clusterMgr [$AdminControl completeObjectName cell=mycell,type=ClusterMgr,*]
     ```
   - Using Jython:
     ```
     clusterMgr = AdminControl.completeObjectName('cell=mycell,type=ClusterMgr,*')
     print clusterMgr
     ```

   This command returns the ClusterMgr MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=ClusterMgr,mbeanIdentifier=ClusterMgr,
   type=ClusterMgr,process=dmgr
   ```
2. Refresh the list of clusters.
   - Using Jacl:
     ```
     $AdminControl invoke $clusterMgr retrieveClusters
     ```
   - Using Jython:
     ```
     AdminControl.invoke(clusterMgr, 'retrieveClusters')
     ```

   This command calls the retrieveClusters operation on the ClusterMgr MBean.
3. Identify the Cluster MBean and assign it to the cluster variable.
   - Using Jacl:
     ```
     set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*]
     ```
   - Using Jython:
     ```
     cluster = AdminControl.completeObjectName('cell=mycell,type=Cluster,name=cluster1,*')
     print cluster
     ```

   This command returns the Cluster MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=cluster1,mbeanIdentifier=Cluster,type=Cluster,process=cluster1
   ```
4. Start or RippleStart the cluster.
   - To start a cluster, use the following example. These commands invoke the start operation on the cluster MBean:
     –
     Using Jacl:
     ```
     $AdminControl invoke $cluster start
     ```
     – Using Jython:
     ```
     AdminControl.invoke(cluster, 'start')
     ```
   - Use the following example to RippleStart a cluster. RippleStart combines stopping and starting operations. It first stops and then restarts each member of the cluster. For example, your cluster contains 3 cluster members named server_1, server_2 and server_3. When you click RippleStart, server_1 stops and restarts, then server_2 stops and restarts, and finally server_3 stops and restarts. Use the RippleStart option instead of manually stopping and then starting all of the application servers in the cluster. The following commands invoke the rippleStart operation on the cluster MBean:
     –
     Using Jacl:
     ```
     $AdminControl invoke $cluster rippleStart
     ```
     – Using Jython:
     ```
     AdminControl.invoke(cluster, 'rippleStart')
     ```

Querying cluster state using scripting

You can query cluster states using the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to query cluster state:

1. Identify the Cluster MBean and assign it to the cluster variable.
   - Using Jacl:
     ```
     set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*]
     ```
   - Using Jython:
     ```
     cluster = AdminControl.completeObjectName('cell=mycell,type=Cluster,name=cluster1,*')
     print cluster
     ```

   This command returns the Cluster MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=cluster1,mbeanIdentifier=Cluster,type=Cluster,process=cluster1
   ```
2. Query the cluster state.
   - Using Jacl:
     ```
     $AdminControl getAttribute $cluster state
     ```
   - Using Jython:
     ```
     AdminControl.getAttribute(cluster, 'state')
     ```

   This command returns the value of the run-time state attribute.

Stopping clusters using scripting

Use scripting and the wsadmin tool to stop a cluster.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to stop a cluster:

1. Identify the Cluster MBean and assign it to the cluster variable.
   - Using Jacl:
     ```
     set cluster [$AdminControl completeObjectName cell=mycell,type=Cluster,name=cluster1,*]
     ```
   - Using Jython:
     ```
     cluster = AdminControl.completeObjectName('cell=mycell,type=Cluster,name=cluster1,*')
     print cluster
     ```

   This command returns the Cluster MBean.

   Example output:

   ```
   WebSphere:cell=mycell,name=cluster1,mbeanIdentifier=Cluster,type=Cluster,process=cluster1
   ```
2. Stop the cluster.
   - Using Jacl:
     ```
     $AdminControl invoke $cluster stop
     ```
   - Using Jython:
     ```
     AdminControl.invoke(cluster, 'stop')
     ```

   This command invokes the stop operation on the Cluster MBean.

Commands for the ClusterConfigCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the ClusterConfigCommands group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

createCluster	The **createCluster** command creates a new server cluster. A server cluster consists of a group of application servers that are referred to as *cluster members*. Optionally, a replication domain can be created for the new cluster, and an existing server can be included as the first cluster member.	None	• **Parameters for step one:** ***-clusterConfig*** Specifies the configuration of the new server cluster. This command step is required. The following parameters can be specified for this step: **clusterName** The name of the new server cluster. (Required) **clusterType** The type of the server cluster that you are creating. The default type is APPLICATION_SERVER. Valid values for this parameter include: APPLICATION_SERVER, PROXY_SERVER, AND ONDEMAND_ROUTER. (Optional) **preferLocal** Enables or disables node scoped routing optimization within this cluster. The value is true or false. It not specified, the default value is true. (Optional) • **Parameters for step two:** ***-replicationDomain*** Specifies the configuration of a replication domain for this cluster. A replication domain is used to support HTTP session data replication. This command step is optional. The following parameters can be specified for this step: **createDomain** Creates a replication domain with a name set to the name of the new cluster. The value is true or false. It not specified, the default value is false. (Optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createCluster {-clusterConfig {{cluster1 true}}}` `$AdminTask createCluster {-clusterConfig {{cluster1 true}} -replicationDomain {{true}}}` `$AdminTask createCluster {-clusterConfig {{cluster1 true}} -convertServer {{node1 server1 "" "" ""}}}` • Using Jython string: `AdminTask.createCluster('[-clusterConfig [[cluster1 true]]]')` `AdminTask.createCluster('[-clusterConfig [[cluster1 true]] -replicationDomain [[true]]]')` `AdminTask.createCluster('[-clusterConfig [[cluster1 true]] -convertServer [[node1 server1 "" "" ""]]]')` • Using Jython list: `AdminTask.createCluster (['-clusterConfig', [[' cluster1', 'true']]])` `AdminTask.createCluster (['-clusterConfig', [[' cluster1', 'true']], '-replicationDomain', [['true']]])` `AdminTask.createCluster (['-clusterConfig', [[' cluster1', 'true']], '-convertServer', [['node1', 'server1', "" "" ""]]])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCluster {-interactive}` • Using Jython string: `AdminTask.createCluster ('[-interactive]')` • Using Jython list: `AdminTask.createCluster (['-interactive'])`

			• **Parameters for step three:** ***-convertServer*** Specifies information about an existing application server to convert to be the first member of the cluster. This command step is optional. The following parameters can be specified for this step: **serverNode** The name of the node with the server to be converted to the first cluster member. You must also specify the serverName parameter. (Required) **serverName** The name of the application server to be converted to the first cluster member. You must also specify the serverNode parameter. (Required) **memberWeight** The weight of the cluster member. The weight controls the amount of work directed to the application server. If the weight is greater than the weight assigned to other cluster members, the server will receive a larger share of the workload. The value is a number between 0 and 100. If none is specified, the default is 2. (Optional)	

			nodeGroup The name of the node group which this cluster member's node, and all future cluster members' nodes, must belong to. All cluster members must reside on nodes in the same node group. If specified, it must be one of the node groups which this member's node belongs to. If not specified, the default value will be the first node group listed for this member's node. (Optional) **replicatorEntry** Specifies a replicator entry for the converted member will be created in the cluster's replication domain. A replicator entry is used to provide HTTP session data replication. The value is true or false which indicates whether the replicator entry will be created. The default value is false. You can specify this parameter only if the createDomain parameter was set to true in the replicationDomain command step. (Optional) • **Returns:** The object name of cluster that was created.	

createClusterMember	The **createClusterMember** command creates a member of a server cluster. A cluster member is an application server that belongs to a cluster. If this is the first member of the cluster, you must specify a template to use as the model for the cluster member. The template can be either a default server template, or an existing application server. The first cluster member is used as a template to create subsequent members in the cluster. When you create a first cluster member, the template of the cluster member is stored under the scope of the cluster. Since a cluster can contain members from nodes that run different versions of WebSphere Application Server,	cluster Object ID - The configuration object ID of the cluster to which the new member will belong. If you do not specify the configuration ID, you must specify the clusterName parameter. You can obtain the object name programmatically through Java using the WebSphere Application Server ConfigService API or through wsadmin scripting using the AdminConfig object.	• **Parameters:** **-clusterName** The name of the cluster to which the new member will belong. If you do not specify this parameter, you must specify the cluster object ID in the command target. • **Parameters for step one:** ***-memberConfig*** Specifies the attributes of the new cluster member to be created in the cluster. This command step is required. The following parameters can be specified for this step: **memberNode** The name of the node where the new cluster member will be created. This parameter is required. **memberName** The name of the server to be created for the new cluster member. This parameter is required. **memberWeight** The weight of the new cluster member. This controls the amount of work directed to the application server. If the weight is greater than the weight assigned to other cluster members, the server will receive a larger share of the workload. The value is a number between 0 and 100. The default value is 2.	**Batch mode example usage:** • Using Jacl: First member creation using template name: `$AdminTask createClusterMember {-clusterName cluster1 -memberConfig {{node1 member1 "" "" true false}} -firstmember {{serverTemplateName "" "" "" ""}}}` First member creation using server and node for template: `$AdminTask createClusterMember {-clusterName cluster1 -memberConfig {{node1 member1 "" "" true false}} -firstmember {{ "" node1 server1 "" ""}}}` Second member creation: `$AdminTask createClusterMember {-clusterName cluster1 -memberConfig {{node1 member2 "" "" true false}}}`

	one template will be stored for each version of the WebSphere Application Server node that is already configured as cluster members. The cluster member template will not exist for a given version of node until a first member is created in any node of the same version. For example, if a cluster contains some V6.1 nodes and some V6.0.x nodes, there will be one cluster member template for the V6.1 node and one cluster member template for the V6.0.x node. The following template will be used for members that are created for a V6.1 node: *$WAS_HOME*/config/templates/clusters/clusterName/servers/V6.1MemberTemplate.		**genUniquePorts** Generates unique port numbers for each HTTP transport defined in the server. The new server will not have HTTP transports which conflict with any other servers defined on the same node. The value is true or false. The default value is true . **replicatorEntry** Specifies a replicator entry for the new cluster member will be created in the cluster's replication domain. A replicator entry is used to provide HTTP session data replication. This command parameter is optional. The value is true or false which indicates whether the entry will be created. The default value is false. You can specify this parameter only if a replication domain has been created for the cluster. • **Parameters for step two:** ***-firstMember*** Specifies additional information necessary to create the first cluster member. This command step is required when creating the first member of the cluster, and is executable only when creating the first member of the cluster. The target of this command step is a Boolean value indicating whether or not to perform this step. The default value is true if any of the step parameters are specified; otherwise the default value is false. The following parameters can be specified for this step:	• Using Jython string: First member creation using template name: `AdminTask.createClusterMember ('[-clusterName cluster1 -memberConfig [[node1 member1 "" "" true false]] -firstMember [[serverTemplateName "" "" "" ""]]]')` First member creation using server and node for template: `AdminTask.createClusterMember ('[-clusterName cluster1 -memberConfig [[node1 member1 "" "" true false]] -firstMember [["" node1 server1 "" ""]]]')` Second member creation: `AdminTask.createClusterMember ('[-clusterName cluster1 -memberConfig [[node1 member2 "" "" true false]]]')` • Using Jython list: First member creation using template name: `AdminTask.createClusterMember (['-clusterName', 'cluster1', '-memberConfig', [['node1', 'member1', "" "", 'true', 'false']], '-firstMember', [['serverTemplateName', "" "" "" ""]]])`

	The following template will be used for members that are created for a V6.0.x node: *$WAS_HOME*/config/templates/clusters/clusterName/servers/V6MemberTemplate. The following template will be used for members that are created for a V5.x node: *$WAS_HOME*/config//clusters/clusterName/servers/V5MemberTemplate. When you make a configuration change to members in a cluster, you must make the same configuration change to the template that is stored in the cluster scope that corresponds.		**templateName** The name of an application server template to use when creating the new cluster member. If you specify a template, you cannot specify the templateServerNode and templateServerName parameters to use an existing application server as a template. You are required to specify either the templateName parameter, or the templateServerNode and templateServerName parameters in this step. **templateServerNode** The name of the node with an existing application server to use as the template when creating the new cluster member. If you specify the templateServerNode parameter, you must also specify the templateServerName parameter, and you cannot specify the templateName parameter. You are required to specify either the templateName parameter, or the templateServerNode and templateServerName parameters, in this step. **templateServerName** The name of the existing application server to use as the model when creating the new cluster member. If you specify the templateServerName parameter, you must also specify the templateServerNode parameter, and you cannot specify the templateName parameter. You are required to specify either the templateName parameter, or the templateServerNode and templateServerName parameters, in this command step.	

			nodeGroup The name of the node group which this cluster member's node, and all future cluster members' nodes, must belong to. All cluster members must reside on nodes in the same node group. This parameter is optional. If specified, it must be one of the node groups which this member's node belongs to. If not specified, the default value will be the first node group listed for this member's node. **coreGroup** The name of the core group this cluster member, and all future cluster members, must belong to. All cluster members must belong to the same core group. This parameter is optional. If not specified, the default value is the default core group defined in the cell. • **Returns:** The object name of cluster member that was created.	

				First member creation using server and node for template: `AdminTask.createClusterMember (['-clusterName', 'cluster1', '-memberConfig', [['node1', 'member1', "" "", 'true', 'false']], '-firstMember', [["", 'node1', 'server1', "" ""]]])` Second member creation: `AdminTask.createClusterMember (['-clusterName', 'cluster1', '-memberConfig', [['node1', 'member2', "" "", 'true', 'false']]])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createClusterMember {-interactive}` • Using Jython string: `AdminTask.createClusterMember ('[-interactive]')` • Using Jython list: `AdminTask.createClusterMember (['-interactive'])`

deleteCluster	The **deleteCluster** command deletes the configuration of a server cluster. A server cluster consists of a group of application servers that are referred to as *cluster members*. When a server cluster is deleted, all of its members are deleted. Use the **deleteClusterMember** command to delete the configuration of an individual cluster member.	cluster Object ID - The configuration object ID of the cluster to be deleted. If the cluster's object ID is not specified, then the clusterName parameter must be specified. The object name can be obtained programmatically through Java using the WebSphere ConfigService API, or through wsadmin scripting using the AdminConfig object.	• **Parameters:** **-clusterName** The name of the cluster to be deleted. If this parameter is not specified, then the cluster object ID must be specified in the command target. • **Parameters for step one:** ***-replicationDomain*** Specifies the removal of the replication domain for this cluster. This command step is optional. The following parameters can be specified for this step: **deleteDomain** Deletes the replication domain for this cluster. This parameter is optional. The value is true or false which indicates whether the domain will be deleted. The default value is false. . Deleting the replication domain deletes all replicator entries defined in the domain. • **Returns:** None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteCluster {-clusterName cluster1 }` `$AdminTask deleteCluster {-clusterName cluster1 -replicationDomain {{true}}}` • Using Jython string: `AdminTask.deleteCluster('[-clusterName cluster1]')` `AdminTask.deleteCluster('[-clusterName cluster1 -replicationDomain [[true]]]')` • Using Jython list: `AdminTask.deleteCluster(['-clusterName', 'cluster1'])` `AdminTask.deleteCluster(['-clusterName', 'cluster1', '-replicationDomain', [['true']]])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCluster -interactive` • Using Jython string: `AdminTask.deleteCluster ('[-interactive]')` • Using Jython list: `AdminTask.deleteCluster (['-interactive'])`

deleteClusterMember	The **deleteClusterMember** command deletes the configuration of a cluster member. A cluster member is an application server that belongs to a server cluster. Use the **deleteCluster** command to delete the configuration of a cluster.	member Object ID - The configuration object ID of the cluster member to be deleted. If this is not specified, then the clusterName, memberNode and memberName parameters must be specified. The object name can be obtained programmatically via Java using the WebSphere ConfigService API, or via wsadmin scripting using the AdminConfig command.	• **Parameters:** **-clusterName** The name of the cluster which the member to be deleted belongs to. If this parameter is specified, then the memberName and memberNode parameters must also be specified. If this is not specified, then the member object ID must be specified in the command target. **-memberName** The server name of the member to be deleted from the cluster. If this parameter is specified, then the clusterName and memberNode parameters must also be specified. If this is not specified, then the member object ID must be specified in the command target. **-memberNode** The name of the node having the cluster member to be deleted. If this parameter is specified, then the memberName and clusterName parameters must also be specified. If this is not specified, then the cluster member object ID must be specified in the command target.	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteCluster Member {-clusterName cluster1 -memberNode node1 -member Name member1}` `$AdminTask deleteCluster Member {-clusterName cluster1 -memberNode node1 -member Name member2 -replication Entry {{true}}}` • Using Jython string: `AdminTask.deleteCluster Member('[-clusterName cluster1 -memberNode node1 -member Name member1]')` `AdminTask.deleteClusterMember ('[-clusterName cluster1 -memberNode node1 -memberName member2 -replicationEntry [[true]]]')` • Using Jython list: `AdminTask.deleteClusterMember (['-clusterName', 'cluster1', '-memberNode', 'node1', '-memberName', 'member1'])` `AdminTask.deleteClusterMember (['-clusterName', 'cluster1', '-memberNode', 'node1', '-memberName', 'member2', '-replicationEntry', [['true']]])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteClusterMember -interactive` • Using Jython string: `AdminTask.deleteClusterMember ('[-interactive]')` • Using Jython list: `AdminTask.deleteClusterMember (['-interactive'])`

			• **Parameters for step one:** ***-replicatorEntry*** Specifies the removal of a replicator entry for this cluster member. This command step is optional. The following parameters can be specified for this step: **deleteEntry** Delete the replicator entry having this cluster member's name from the cluster's replication domain. This parameter is optional. The value is true or false which indicates whether to delete the replicator entry. The default value is false. • **Returns:** None	

Configuring security with scripting

You can configure security with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

If you enable security for a WebSphere Application Server cell, supply authentication information to communicate with servers.

The `sas.client.props` and the `soap.client.props` files are located in the properties directory for each WebSphere Application Server profile, *profilePath*`/properties`.

- The nature of the properties file updates required for running in secure mode depend on whether you connect with a Remote Method Invocation (RMI) connector, or a SOAP connector:
  - If you use a Remote Method Invocation (RMI) connector, set the following properties in the `sas.client.props` file with the appropriate values:

    ```
    com.ibm.CORBA.loginUserid=
    com.ibm.CORBA.loginPassword=
    ```

    Also, set the following property:

    ```
    com.ibm.CORBA.loginSource=properties
    ```

    The default value for this property is `prompt` in the `sas.client.props` file. If you leave the default value, a dialog box appears with a password prompt. If the script is running unattended, it appears to hang.
  - If you use a SOAP connector, set the following properties in the `soap.client.props` file with the appropriate values:

    ```
    com.ibm.SOAP.securityEnabled=true
    com.ibm.SOAP.loginUserid=
    com.ibm.SOAP.loginPassword=
    ```

    Optionally, set the following property:

```
com.ibm.SOAP.loginSource=none
```

The default value for this property is `prompt` in the `soap.client.props` file. If you leave the default value, a dialog box appears with a password prompt. If the script is running unattended, it appears to hang.

- To specify user and password information, choose one of the following methods:
  - Specify user name and password on a command line, using the **-user** and **-password** commands. For example:

    ```
    wsadmin.sh -conntype RMI -port 2809 -user u1 -password secret1
    ```
  - Specify user name and password in the `sas.client.props` file for a RMI connector or the `soap.client.props` file for a SOAP connector.

  If you specify user and password information on a command line and in the `sas.client.props` file or the `soap.client.props` file, the command line information overrides the information in the `props` file.

  Note: On UNIX system, the use of -password option may result in security exposure as the password information becomes visible to the system status program such as ps command which can be invoked by other user to display all the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the soap.client.props file for SOAP connector or sas.client.props file for RMI connector. The soap.client.props and sas.client.props files are located in the properties directory of your WebSphere Application Server profile.

Enabling and disabling administrative security using scripting

You can use scripting to enable or disable administrative security.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

The default profile sets up procedures so that you can enable and disable administrative security based on LocalOS registry.

- To determine if application security is enabled or disabled by looking at the value of the appEnabled field in the WCCM security model, use the **isAppEnabled** command, for example:
  - Using Jacl:

    ```
    $AdminTask isAppSecurityEnabled {-interactive}
    ```
  - Using Jython:

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

  This command returns a value of `true` if appEnabled is set to `true`. Otherwise, returns a value of `false`.
- To determine if administrative security is enabled or disabled by looking at the value of the enabled field in the WCCM security model, use the **isGlobalSecurity** command, for example:
  - Using Jacl:

    ```
    $AdminTask isGlobalSecurity {-interactive}
    ```
  - Using Jython:

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

  Returns a value of `true` if enabled is set to `true`. Otherwise, returns a value of `false`.
- To set administrative security based on the passed in value, use the **setGlobalSecurity** command. For example:
  - Using Jacl:

    ```
    $AdminTask setGlobalSecurity {-interactive}
    ```
  - Using Jython:

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

Returns a value of `true` if the enabled field in the WCCM security model is successfully updated. Otherwise, returns a value of `false`.

- You can use the **help** command to find out the arguments that you need to provide with this call, for example:
  - Using Jacl:

    ```
    securityon help
    ```

    Example output:

    ```
    Syntax: securityon user password
    ```

  - Using Jython:

    ```
    securityon()
    ```

    Example output:

    ```
    Syntax: securityon(user, password)
    ```

- To enable administrative security based on the LocalOS registry, use the following procedure call and arguments:
  - Using Jacl:

    ```
    securityon user1 password1
    ```

  - Using Jython:

    ```
    securityon('user1', 'password1')
    ```

- To disable administrative security based on the LocalOS registry, use the following procedure call:
  - Using Jacl:

    ```
    securityoff
    ```

  - Using Jython:

    ```
    securityoff()
    ```

Enabling and disabling LTPA authentication

There are sample scripts located in the `<WAS_ROOT>/bin` directory on how to enable and disable LTPA authentication. The scripts are:

- LTPA_LDAPSecurityProcs.py (python script)
- LTPA_LDAPSecurityProcs.jacl (jacl script)

Note: The scripts hard code the type of LDAP server and base distinguished name (baseDN). The LDAP server type is hardcoded as IBM_DIRECTORY_SERVER and the baseDN is hardcoded as o=ibm,cn=us.

Enabling and disabling Java 2 security using scripting

You can enable or disable Java 2 security with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to enable or disable Java 2 security:

1. Identify the security configuration object and assign it to the security variable:
   - Using Jacl:

     ```
     set security [$AdminConfig list Security]
     ```

   - Using Jython:

     ```
     security = AdminConfig.list('Security')
     print security
     ```

   Example output:

   ```
   (cells/mycell|security.xml#Security_1)
   ```

2. Modify the enforceJava2Security attribute to enable or disable Java 2 security. For example:

- To enable Java 2 security:
  - Using Jacl:
    ```
    $AdminConfig modify $security {{enforceJava2Security true}}
    ```
  - Using Jython:
    ```
    AdminConfig.modify(security, [['enforceJava2Security', 'true']])
    ```
- To disable Java 2 security:
  - Using Jacl:
    ```
    $AdminConfig modify $security {{enforceJava2Security false}}
    ```
  - Using Jython:
    ```
    AdminConfig.modify(security, [['enforceJava2Security', 'false']])
    ```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Propagating security policy of installed applications to a JACC provider using wsadmin scripting

It is possible that you have applications installed prior to enabling the Java Authorization Contract for Containers (JACC)-based authorization. You can start with default authorization and then move to an external provider-based authorization using JACC later.

Also, during application installation or modification you might have had problems propagating the security policy information to the JACC provider. For example, network problems might occur, the JACC provider might not be available, and so on. For these cases, the security policy of the previously installed applications does not exist in the JACC provider to make the access decisions. One choice is to reinstall the applications involved. However, you can avoid reinstalling by using the wsadmin scripting tool. Use this tool to propagate information to the JACC provider independent of the application installation process. The tool eliminates the need for reinstalling the applications.

The tool uses the SecurityAdmin MBean to propagate the policy information in the deployment descriptor of any installed application to the JACC provider. You can invoke this tool using wsadmin at the base application server for base and deployment manager level for Network Deployment. Note that the SecurityAdmin MBean is available only when the server is running.

Use `propagatePolicyToJACCProvider(String appNames)` to propagate the policy information in the deployment descriptor of the enterprise archive (EAR) files to the JACC provider. If the RoleConfigurationFactory and the RoleConfiguration interfaces are implemented by the JACC provider, the authorization table information in the binding file of the EAR files is also propagated to the provider. See the *Securing applications and their environment* PDF for more information about these interfaces.

The `appNames String` contains the list of application names, delimited by a colon (:), whose policy information must be stored in the provider. If a null value is passed, the policy information of the deployed applications is propagated to the provider.

Also, be aware of the following items:

- Before migrating applications to the Tivoli Access Manager JACC provider, create or import the users and groups that are in the applications to Tivoli Access Manager.
- Depending on the application or the number of applications that are propagated, you might have to increase the request time-out period either in the `soap.client.props` file in the directory *profile_root*`/properties` (if using SOAP) or in the `sas.client.props` file (if using RMI) for the command to complete. You can set the request time-out value to 0 to avoid the timeout problem, and change it back to the original value after the command is run.

1. Configure your JACC provider in WebSphere Application Server.
   See the *Securing applications and their environment* PDF for more information.
2. Restart the server.
3. Enter the following commands:

```
//use the SecurityAdmin MBean at the Deployment Manager or the unmanaged base
//application server connect to the appropriate process (Deployment Manager or
//base application server)
wsadmin -user serverID -password serverPWD
// To get the SecurityAdmin MBean for Deployment Manager
wsadmin> set secadm [$AdminControl queryNames type=SecurityAdmin,process=dmgr,*]
// or to get the SecurityAdmin MBean for a unmanaged base application server
//(replace the process name to match your configuration)
wsadmin> set secadm [$AdminControl queryNames
         type=SecurityAdmin,process=server1,*]
// to propagate specific applications security policy information
wsadmin>set appNames [list app1:app2]
// or to propagate all applications installed
wsadmin>set appNames [list null]

// Run the command to propagate
wsadmin>$AdminControl invoke $secadm propagatePolicyToJACCProvider $appNames
```

Configuring the JACC provider for Tivoli Access Manager using the wsadmin utility

You can use the wsadmin utility to configure Tivoli Access Manager security for WebSphere Application Server.

Verify that all the managed servers, including node agents, are started. The following configuration is performed once on the deployment manager server. The configuration parameters are forwarded to managed servers, including node agents, when a synchronization is performed. The managed servers require their own restart for the configuration changes to take effect.

1. Start WebSphere Application Server.
2. Start the **wsadmin** command-line utility.
   Run the **wsadmin** command from the *app_server_root*/bin directory.
3. At the **wsadmin** prompt, enter the following command:

   $AdminTask configureTAM -interactive

   You are prompted to enter the following information:

Option	Description
WebSphere Application Server node name	Specify a single node or enter an asterisk (*) to choose all nodes including the deployment manager.
Tivoli Access Manager Policy Server	Enter the name of the Tivoli Access Manager policy server and the connection port. Use the format, *policy_server* **:** *port*. The policy server communication port is set at the time of Tivoli Access Manager configuration. The default port is 7135.

Option	Description
Tivoli Access Manager Authorization Server	Enter the name of the Tivoli Access Manager authorization server. Use the format *auth_server* **:** *port* **:** *priority*. The authorization server communication port is set at the time of Tivoli Access Manager configuration. The default port is 7136. More than one authorization server can be specified by separating the entries with commas. Having more than one authorization server configured is useful for failover and performance. The priority value is the order of authorization server use. For example: auth_server1**:**7136**:**1,auth_server2**:**7137**:**2. A priority of 1 is still required when configuring against a single authorization server.
WebSphere Application Server administrator's distinguished name	Enter the full distinguished name of the WebSphere Application Server security administrator ID, as created in the ″Creating the security administrative user″ topic in the *Securing applications and their environment* PDF. For example: cn=wasadmin,o=*organization*,c=*country*
Tivoli Access Manager user registry distinguished name suffix	For example: o=*organization*,c=*country*
Tivoli Access Manager administrator's user name	Enter the Tivoli Access Manager administration user ID, as created at the time of Tivoli Access Manager configuration. This ID is usually, sec_master.
Tivoli Access Manager administrator's user password	Enter the password for the Tivoli Access Manager administrator.
Tivoli Access Manager security domain	Enter the name of the Tivoli Access Manager security domain that is used to store users and groups. If a security domain is not already established at the time of Tivoli Access Manager configuration, click **Return** to accept the default.
Embedded Tivoli Access Manager listening port set	WebSphere Application Server needs to listen on a TCP/IP port for authorization database updates from the policy server. More than one process can run on a particular node and machine so a list of ports is required for the processes. Enter the ports that are used as listening ports by Tivoli Access Manager clients, separated by a comma. If you specify a range of ports, separate the lower and higher values by a colon. For example, 7999, 9990:9999.
Defer	Set to *yes*, this option defers the configuration of the management server until the next restart. Set to *no*, configuration of the management server occurs immediately. Managed servers are configured on their next restart.

4. When all information is entered, select **F** to save the configuration properties or **C** to cancel from the configuration process and discard entered information.

Now enable the JACC provider for Tivoli Access Manager- Enabling the JACC provider for Tivoli Access Manager topic in the *Securing applications and their environment* PDF.

Disabling embedded Tivoli Access Manager client using wsadmin

Follow these steps to unconfigure the Java Authorization Contract for Containers (JACC) provider for Tivoli Access Manager.

In a Network Deployment architecture, ensure that all the managed servers, including node agents, are started. Perform the following process once on the deployment management server. Details of the unconfiguration are forwarded to managed servers, including node agents, when a synchronization is performed. The managed servers require their own reboot for the configuration changes to take effect.

1. Restart the deployment manager process.
2. Start the wsadmin command-line utility. The wsadmin command is found in the*install_dir*/bin directory
3. From the **wsadmin** prompt, enter the following command:

   ```
   WSADMIN>$AdminTask unconfigureTAM -interactive
   ```

   You are prompted to enter the following information:

Option	Description
WebSphere Application Server node name	Enter an asterisk (*) to select all nodes.
Tivoli Access Manager administrator user name	Enter the Tivoli Access Manager administration user ID, as created at the time of Tivoli Access Manager configuration. This name is usually, `sec_master`.
Tivoli Access Manager administrator user password	Enter the password for the Tivoli Access Manager administrator.
Force	Enter *yes,* if you want to ignore errors when unconfiguring the JACC provider for Tivoli Access Manager. Enter this option as *yes* only when the Tivoli Access Manager domain is in an irreparable state.
Defer	Enter `no`, to force the unconfiguration of the connected server. Enter `No` for the unconfiguration to proceed correctly.

4. When all information is entered, enter `F` to save the properties or `C` to cancel from the unconfiguration process and discard the entered information.
5. **Optional:** Synchronize all nodes.
6. Restart all WebSphere Application Server instances for the changes to take effect.

Creating an SSL configuration at the node scope using scripting

An Secure Socket Layer (SSL) configuration references many other configuration objects. To help you make valid selections for the new SSL configuration before you create it, view information about existing configuration objects. Information about existing objects is also useful when you create a node scoped SSL configuration using the **createSSLConfig** command of the AdminTask object.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

To use the information in this task effectively, familiarize yourself with the instructions in the Creating a Secure Sockets Layer configuration topic in the *Securing applications and their environment* PDF. Perform the following task to create an Secure Socket Layer (SSL) configuration at the node scope:

1. List the existing configuration objects. Perform any of the following:
   - List some of the configuration objects that you may need when you create a new SSL configuration.

     For example, you want to see which management scopes have already been defined. If the one you need does not exist you will need to create it.
     - Using Jacl:

       ```
       $AdminTask listManagementScopes {-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02}
       ```
     - Using Jython:

```
AdminTask.listManagementScopes ('[-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02]')
```

This shows an existing cell scope and existing node scope that you can use. If you want to create a different scope, use the **createManagementScope** command of the AdminTask object to define a different one. The valid scope parameters are cell, nodegroup, node, server, cluster, and endpoint. See the Central management of Secure Sockets Layer configurations topic in the *Securing applications and their environment* PDF for more information on scope definitions.

- List the key stores that exist in the configuration including key stores and trust stores.
  - Using Jacl:

    ```
    $AdminTask listKeyStores
    ```

  - Using Jython:

    ```
    AdminTask.listKeyStores()
    ```

  Example output:

  ```
  CellDefaultKeyStore(cells/BIRKT40Cell02|security.xml#KeyStore_1)
  CellDefaultTrustStore(cells/BIRKT40Cell02|security.xml#KeyStore_2)
  CellLTPAKeys(cells/BIRKT40Cell02|security.xml#KeyStore_3)
  ```

  The previous example only lists the key stores for the default management scope which is also known as the cell scope. To obtain key stores for other scopes, specify the `scopeName` parameter, for example:

  - Using Jacl:

    ```
    $AdminTask listKeyStores {-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 }
    ```

  - Using Jython:

    ```
    $AdminTask listKeyStores ('[-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02]')
    ```

  Example output:

  ```
  CellDefaultKeyStore(cells/BIRKT40Cell02|security.xml#KeyStore_1)
  CellDefaultTrustStore(cells/BIRKT40Cell02|security.xml#KeyStore_2)
  CellLTPAKeys(cells/BIRKT40Cell02|security.xml#KeyStore_3)
  NodeDefaultKeyStore(cells/BIRKT40Cell02|security.xml#KeyStore_1134610924357)
  NodeDefaultTrustStore(cells/BIRKT40Cell02|security.xml#KeyStore_1134610924377)
  ```

- List specific trust or key managers. Be sure to display the object name for the trust managers. You will need the object name for the SSL configuration because you can specify multiple trust manager instances.
  - Using Jacl:

    ```
    $AdminTask listTrustManagers {-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -displayObjectName true }
    ```

  - Using Jython:

    ```
    AdminTask.listTrustManagers ('[-scopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -displayObjectName true]')
    ```

  Example output:

  ```
  IbmX509(cells/BIRKT40Cell02|security.xml#TrustManager_1)
  IbmPKIX(cells/BIRKT40Cell02|security.xml#TrustManager_2)
  IbmX509(cells/BIRKT40Cell02|security.xml#TrustManager_1134610924357)
  IbmPKIX(cells/BIRKT40Cell02|security.xml#TrustManager_1134610924377)
  ```

2. Create the node-scoped SSL configuration in interactive mode. Now that we have the information we need to choose from, we need to decide if these objects are sufficient or if we need to create new ones. For now, we will reuse what we've already got in the configuration and save creating new instances to task documents specific to those objects.
   - Using Jacl:

     ```
     $AdminTask createSSLConfig -interactive
     ```

   - Using Jython:

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

   Example output:

```
Create a SSL Configuration.

*SSL Configuration Alias (alias): BIRKT40Node02SSLConfig
Management Scope Name (scopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02
Client Key Alias (clientKeyAlias): default
Server Key Alias (serverKeyAlias): default
SSL Type (type): [JSSE]
Client Authentication (clientAuthentication): [false]
Security Level of the SSL Configuration (securityLevel): [HIGH]
Enabled Ciphers SSL Configuration (enabledCiphers):
JSSE Provider (jsseProvider): [IBMJSSE2]
Client Authentication Support (clientAuthenticationSupported): [false]
SSL Protocol (sslProtocol): [SSL_TLS]
Trust Manager Object Names (trustManagerObjectNames): (cells/BIRKT40Cell02|security.xml#TrustManager_1)
*Trust Store Name (trustStoreName): NodeDefaultTrustStore
Trust Store Scope (trustStoreScopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02
*Key Store Name (keyStoreName): NodeDefaultKeyStore
Key Store Scope Name (keyStoreScopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02
Key Manager Name (keyManagerName): IbmX509
Key Manager Scope Name (keyManagerScopeName): (cell):BIRKT40Cell02:(node):BIRKT40Node02

Create SSL Configuration

F (Finish)
C (Cancel)

Select [F, C]: [F] F
WASX7278I: Generated command line: $AdminTask createSSLConfig {-alias BIRKT40Node02SSLConfig -scopeName
(cell):BIRKT40Cell02:(node):BIRKT40Node02 -clientKeyAlias default -serverKeyAlias default
-trustManagerObjectNames (cells/BIRKT40Cell02|security.xml#TrustManager_1) -trustStoreName
NodeDefaultTrustStore -trustStoreScopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -keyStoreName
NodeDefaultKeyStore -keyStoreScopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 -keyManagerName
IbmX509 -keyManagerScopeName (cell):BIRKT40Cell02:(node):BIRKT40Node02 }
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

The name of the SSL configuration object that you created, for example, (`cells/BIRKT40Cell02|security.xml#SSLConfig_1136652770753`), appears in the `security.xml` file.

Example `security.xml` file output:

```
<repertoire xmi:id="SSLConfig_1136652770753" alias="BIRKT40Node02SSLConfig" type="JSSE"
managementScope="ManagementScope_1134610924357">
<setting xmi:id="SecureSocketLayer_1136652770924" clientKeyAlias="default" serverKeyAlias="default"
clientAuthentication="false" securityLevel="HIGH"  jsseProvider="IBMJSSE2" sslProtocol="SSL_TLS"
keyStore="KeyStore_1134610924357" trustStore="KeyStore_1134610924377" trustManager="TrustManager_1"
keyManager="KeyManager_1134610924357"/>
</repertoire>
```

Once you create the SSL configuration object, the next step is to use it. There are several different ways that you can associate SSL configurations with protocols, for example:

- Set the SSL configuration on the thread programmatically.
- Associate the SSL configuration with an outbound protocol or a target host and port.
- Directly associating the SSL configuration using the alias.
- Centrally managing the SSL configurations by associating them with SSL configuration groups or zones so that they are used based upon the group from where the end point exists.

Creating self-signed certificates using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

You can create self-signed certificates using the scripting and the AdminTask object. You can run the commands in interactive or batch mode. Interactive mode provides a way to discover the flags that you need to run the task in batch mode.

Certificates reside inside of key stores. To run the commands, you will need the name of the key store to be supplied. Use the **listKeyStore** command of the AdminTask object to get a list of key stores. If you need a new key store, use the **createKeyStore** command of the AdminTask object.

To create a personal key store, use the following examples:

- Interactive mode:
  - Using Jacl:

    ```
    $AdminTask createSelfSignedCertificate -interactive
    ```

  - Using Jython:

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

    Example output:

    ```
    *Key Store Name (keyStoreName): keyStore
    Key Store Scope Name (keyStoreScope):
    *Certificate Alias (certificateAlias): newCert
    "Certificate Version" (certificateVersion): 3
    *Key Size (certificateSize): [1024]
    *Common Name (certificateCommonName): localhost
    *Organization (certificateOrganization): workgroup
    Organizational Unit (certificateOrganizationalUnit): testing
    certLocality (certificateLocality): austin
    State (certificateState): Texas
    Zip (certificateZip): 78757
    Country (certificateCountry): [US]
    Validity Period (certificateValidDays): [365]
    Create Self-Signed Certificate

    F (Finish)
    C (Cancel)

    Select [F, C]: [F]

    WASX7278I: Generated command line: $AdminTask createSelfSignedCertificate
    {-keyStoreName keyStore -certificateAlias newCert -certificateVersion 3
    -certificateCommonName localhost -certificateOrganization ibm
    -certificateOrganizationalUnit testing -certificateLocality austin
    -certificateState Texas -certificateZip 78757 }
    true
    ```

    At the end of the output, the batch mode parameters are provided.

- Batch mode:
  - Using Jacl:

    ```
    $AdminTask createSelfSignedCertificate {-keyStoreName keyStore
    -certificateAlias newCert -certificateVersion 3 -certificateCommonName localhost
    -certificateOrganization ibm -certificateOrganizationalUnit testing
    -certificateLocality austin -certificateState Texas -certificateZip 78757 }
    ```

  - Using Jython:

    ```
    AdminTask.createSelfSignedCertificate ('[-keyStoreName keyStore
    -certificateAlias newCert -certificateVersion 3 -certificateCommonName localhost
    -certificateOrganization ibm -certificateOrganizationalUnit testing
    -certificateLocality austin -certificateState Texas -certificateZip 78757]')
    ```

Automating SSL configurations using scripting

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

SSL configuration is needed for WebSphere to perform SSL connections with other servers. A SSL configuration can be configured through the Admin Console. But if an automated way to create a SSL configuration is desired then AdminTask should be used.

AdminTask can be used in a interactive mode and batch mode. For automation the batch mode options should be used. AdminTask batch mode can be called in a JACL or Python script. Interactive mode will step through all the parameter the task needs, requires ones are marked with a ‘*’. Before the interactive task executes the task it echoes the batch mode syntax of the task to the screen. This can be helpful when writing batch mode scripts.

There attributes needed to create an ssl configurations:

- A key store
- Default client certificate alias
- Default server certificate alias
- Trust store
- The handshake protocol
- The ciphers needed during handshake
- Supporting client authentication or not

If automating the creation of a SSL Configuration it may be needed to create some of the attribute values needed like the key store, trust store, key manager, and trust managers.

- To create a SSL configuration the createSSLConfig AdminTask can be used. To make changes to the SSL configurations use the modifySSLConfig AdminTask.
  - Interactive mode:

    Interactive mode steps you through all attributes and tell you the default value of the attribute if there is one. The default value is in ‘[]’ on the prompt line. The actual flag used in batch mode is in ‘()’ on each prompt line. If you are using the default value then the flag will not show up on the batch command line.

    Using Jacl:

    ```
    $AdminTask createSSLConfig -interactive
    ```

  - Using Jython:

    ```
    AdminTask.createSSLConfig ('[interactive]')
    ```

  Example output:

```
*SSL Configuration Alias (alias): testSSLConfig
Management Scope Name (scopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
Client Key Alias (clientKeyAlias): clientCert
Server Key Alias (serverKeyAlias): serverCert
SSL Type (type): [JSSE]
Client Authentication (clientAuthentication): [false]
Security Level of the SSL Configuration (securityLevel): [HIGH] HIGH
Enabled Ciphers SSL Configuration (enabledCiphers):
JSSE Provider (jsseProvider): [IBMJSSE2]
Client Authentication Support (clientAuthenticationSupported): [false]
SSL Protocol (sslProtocol): [SSL_TLS] SSL_TLS
Trust Manager Object Names (trustManagerObjectNames):
*Trust Store Name (trustStoreName): testTrustStore
Trust Store Scope (trustStoreScopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
*Key Store Name (keyStoreName): testKeyStore
Key Store Scope Name (keyStoreScopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
Key Manager Name (keyManagerName): IbmX509
Key Manager Scope Name (keyManagerScopeName): (cell):HOSTNode01Cell:(node):HOSTNode01
```

```
Create SSL Configuration

F (Finish)
C (Cancel)

Select [F, C]: [F]
WASX7278I: Generated command line: $AdminTask createSSLConfig {-alias testSSLConfig
-scopeName (cell):HOSTNode01Cell:(node):HOSTNode01 -clientKeyAlias clientCert
-serverKeyAlias serverCert -trustStoreName testTrustStore
-trustStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyStoreName testKeyStore -keyStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyManagerName IbmX509 -keyManagerScopeName (cell):HOSTNode01Cell:(node):HOSTNode01 }
(cells/HOSTNode01Cell|security.xml#SSLConfig_1137687301834)
```

At the end of the output, the batch mode parameters are provided.

– Batch mode:

  Using Jacl:

```
$AdminTask createSSLConfig {-alias testSSLConfig
-scopeName (cell):HOSTNode01Cell:(node):HOSTNode01 -clientKeyAlias clientCert
-serverKeyAlias serverCert -trustStoreName testTrustStore
-trustStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyStoreName testKeyStore -keyStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyManagerName IbmX509 -keyManagerScopeName (cell):HOSTNode01Cell:(node):HOSTNode01}
```

– Using Jython:

```
AdminTask.createSSLConfig ('[-alias testSSLConfig
-scopeName (cell):HOSTNode01Cell:(node):HOSTNode01 -clientKeyAlias clientCert
-serverKeyAlias serverCert -trustStoreName testTrustStore
-trustStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyStoreName testKeyStore -keyStoreScopeName (cell):HOSTNode01Cell:(node):HOSTNode01
-keyManagerName IbmX509 -keyManagerScopeName (cell):HOSTNode01Cell:(node):HOSTNode01]')
```

Example output:

```
(cells/HOSTNode01Cell|security.xml#SSLConfig_1137687301834)
```

- Key Stores and Trust Stores The key store and trust store may already exist or a new one may need to be created. To create a new key store or trust store use the createKeyStore AdminTask. It will create a key store file and store the configuration object in the system configuration. A trust store is just a key store that usually only has signer certificates in it. To create a key store enter:

  – Using Jacl:

```
$AdminTask createKeyStore {-keyStoreName testKeyStore  -keyStoreType PKCS12
-keyStoreLocation $(USER_INSTALL_ROOT)\testKeyStore.p12 -keyStorePassword abcd
-keyStorePasswordVerify abcd  -keyStoreIsFileBased true -keyStoreReadOnly false}
```

  – Using Jython:

```
AdminTask.createKeyStore ('[-keyStoreName testKeyStore  -keyStoreType PKCS12
-keyStoreLocation $(USER_INSTALL_ROOT)\testKeyStore.p12 -keyStorePassword abcd
-keyStorePasswordVerify abcd  -keyStoreIsFileBased true -keyStoreReadOnly false]')
```

  To populate the key store with certificates see “Managing Certificates using AdminConsole and Admin Task” The key store and trust store are required to create a SSL configuration. Use the ‘-keyStoreName’ and ‘-trustStoreName’ flags on the createSSLConfig. There scopes can be added with the ‘-keyStoreScope’ flag and ‘-trustStoreScope’ flags.

- Key Manager Key manager are used to determine how a certificate is selected. The IbmX509 key manager is in the security configuration by default. If a different key manager is needed then use createKeyManager AdminTask to create it. To create a key manager enter:

  – Using Jacl:

```
$AdminTask createKeyManager {-name testKeyManager
-scopeName  (cell):HOSTNode01Cell:(node):HOSTNode01
-provider IBMJSSE2 -algorithm specialAlgorithm }
```

  – Using Jython:

```
AdminTask.createKeyManager ('[-name testKeyManager
-scopeName  (cell):HOSTNode01Cell:(node):HOSTNode01
-provider IBMJSSE2 -algorithm specialAlgorithm]')
```

To supply a key manager on the createSSLConfig AdminTask use the '-keyManagerName' along with the '-keyManagerScope' flag.

- Trust Manager Trust managers are use to determine how trust is established during ssl communication. The IbmX509 and IbmPKIX are trust managers are in the security configuration by default. If a different or additional trust manager is needed then use the createTrustManger AdminTask to create it. To create a trust manager enter:
  - Using Jacl:

    ```
    $AdminTask createTrustManager {-name testTrustManager
    -scopeName (cell):HOSTNode01Cell:(node):HOSTNode01
    -provider IBMJSSE2 -algorithm specialAlgorithm }
    ```

  - Using Jython:

    ```
    AdminTask.createTrustManager ('[-name testTrustManager
    -scopeName (cell):HOSTNode01Cell:(node):HOSTNode01
    -provider IBMJSSE2 -algorithm specialAlgorithm]')
    ```

  The SSL Configuration can have multiple trust managers. To supply multiple trust managers give a comma separated list of the trust managers configuration IDs with the `-trustManagerObjectNames` flag. When you create a trust manager the configuration object ID is returned. To get a list of trust managers object IDs use the **listTrustManagers** command of the AdminTask object with the `-displayObjectName true` flag. For example:

```
wsadmin>$AdminTask listTrustManagers -interactive
List Trust Managers

List trust managers.

Management Scope Name (scopeName):
Display list in ObjectName Format (displayObjectName): [false] true

List Trust Managers

F (Finish)
C (Cancel)

Select [F, C]: [F]
Inside generate script command
WASX7278I: Generated command line: $AdminTask listTrustManagers {-displayObjectName true }
IbmX509(cells/IBM-0AF8DABCF16Node01Cell|security.xml#TrustManager_IBM-0AF8DABCF16Node01_1)
IbmPKIX(cells/IBM-0AF8DABCF16Node01Cell|security.xml#TrustManager_IBM-0AF8DABCF16Node01_2)
```

Updating default key store passwords using scripting

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

When you install WebSphere Application Server, each server creates a key store and trust store for the default SSL configuration with the default password `WebAS`. To protect the security of the key store files and the SSL configuration, you must change the password. The following examples update the default password:

- Change multiple key stores passwords. The **changeMultipleKeyStorePasswords** command updates all of the key stores that have the same password. For example:
  - Using Jacl:

    ```
    $AdminTask changeMultipleKeyStorePasswords  {-keyStorePassword WebAS
    -newKeyStorePassword  secretPwd   -newKeyStorePasswordVerify secretPwd}
    ```

  - Using Jython:

```
AdminTask.changeMultipleKeyStorePasswords  ['(-keyStorePassword WebAS
-newKeyStorePassword secretPwd -newKeyStorePasswordVerify secretPwd]')
```

- Change the password of a single key store. The **changeKeyStorePassword** command updates the password of an individual key store. For example:
  - Using Jacl:

    ```
    $AdminTask changeKeyStorePassword {-keyStoreName testKS
    -keyStoreScope (cell):localhost:(server):server1
    -keyStorePassword WebAS -newKeyStorePassword secretPwd
    -newKeyStorePasswordVerify secretPwd}
    ```

  - Using Jython:

    ```
    AdminTask.changeKeyStorePassword ('[-keyStoreName testKS
    -keyStoreScope (cell):localhost:(server):server1
    -keyStorePassword WebAS -newKeyStorePassword secretPwd
    -newKeyStorePasswordVerify secretPwd]')
    ```

Commands for the IdMgrConfig group of the AdminTask object

Use the commands in the IdMgrConfig group to configure the virtual member manager. The commands for this group do not require a target object. To see the additional commands related to the virtual member manager, see the Commands for the IdMgrRepositoryConfig group of the AdminTask object and the Commands for the IdMgrRealmConfig group of the AdminTask object articles.

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the IdMgrConfig group of the AdminTask object:

Table 3.

Command name:	Description:	Parameters and return values:	Examples:
createIdMgr Supported EntityType	The **createIdMgr Supported EntityType** command creates a supported entity type configuration. To validate the result, check for duplicate entity type names.	• Parameters: **- name** The name of the supported entity type. (String, required) **- defaultParent** The default parent node for the supported entity type. (String, required) **- rdnProperties** The RDN attribute name for the supported entity type in the entity domain name. Separate the RDN properties with semicolons (;). (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createIdMgrSupported EntityType {-name entity1 -defaultParent node1}` • Using Jython string: `AdminTask.createIdMgrSupported EntityType ('[-name entity1 -defaultParent node1]')` • Using Jython list: `AdminTask.createIdMgrSupported EntityType (['-name', 'entity1', '-defaultParent', 'node1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createIdMgrSupported EntityType {-interactive}` • Using Jython string: `AdminTask.createIdMgrSupported EntityType ('[-interactive]')` • Using Jython list: `AdminTask.createIdMgrSupported EntityType (['-interactive'])`

Table 3. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteIdMgr Supported EntityType	The **deleteIdMgr Supported EntityType** command deletes the supported entity type configuration that you specify.	• Parameters: **- name** The name of the supported entity type. The value of this parameter must be one of the supported entity types. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrSupported EntityType {-name entity1}` • Using Jython string: `AdminTask.deleteIdMgrSupported EntityType ('[-name entity1]')` • Using Jython list: `AdminTask.deleteIdMgr SupportedEntityType (['-name', 'entity1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgr SupportedEntityType {-interactive}` • Using Jython string: `AdminTask.deleteIdMgr SupportedEntityType ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgr SupportedEntityType (['-interactive'])`
getIdMgr Supported EntityType	The **getIdMgr Supported EntityType** command returns the configuration of the supported entity type that you specify.	• Parameters: **- name** The name of the supported entity type. The value of this parameter must be one of the supported entity types. (String, required) • Returns: A hash map that has the parameters of the **createIdMgrSupportedEntityType** command as keys. Since the rdnProperties parameter has multiple values, its values are returned in a list.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrSupported EntityType {-name entity1}` • Using Jython string: `AdminTask.getIdMgrSupported EntityType ('[-name entity1]')` • Using Jython list: `AdminTask.getIdMgrSupported EntityType (['-name', 'entity1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrSupported EntityType {-interactive}` • Using Jython string: `AdminTask.getIdMgrSupported EntityType ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrSupported EntityType (['-interactive'])`

Table 3. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMgr Supported EntityTypes	The **listIdMgr Supported EntityTypes** command lists all of the supported entity types that are configured.	• Parameters: None • Returns: A list that contains the names of the supported entity types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgr SupportedEntityTypes` • Using Jython string: `AdminTask.listIdMgr SupportedEntityTypes()` • Using Jython list: `AdminTask.listIdMgr SupportedEntityTypes()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupported EntityTypes {-interactive}` • Using Jython string: `AdminTask.listIdMgrSupported EntityTypes ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrSupported EntityTypes (['-interactive'])`
resetId MgrConfig	The **resetId MgrConfig** command resets the current configuration to the last configuration that was saved.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask resetIdMgrConfig` • Using Jython string: `AdminTask.resetIdMgrConfig()` • Using Jython list: `AdminTask.resetIdMgrConfig()` **Interactive mode example usage:** • Using Jacl: `$AdminTask resetIdMgrConfig {-interactive}` • Using Jython string: `AdminTask.resetIdMgrConfig ('[-interactive]')` • Using Jython list: `AdminTask.resetIdMgrConfig (['-interactive'])`

Table 3. (continued)

Command name:	Description:	Parameters and return values:	Examples:
showId MgrConfig	The **showId MgrConfig** command returns the current configuration XML in string format.	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask showIdMgrConfig` • Using Jython string: `AdminTask.showIdMgrConfig()` • Using Jython list: `AdminTask.showIdMgrConfig()` **Interactive mode example usage:** • Using Jacl: `$AdminTask showIdMgrConfig {-interactive}` • Using Jython string: `AdminTask.showIdMgrConfig ('[-interactive]')` • Using Jython list: `AdminTask.showIdMgrConfig (['-interactive'])`
updateIdMgr Supported EntityType	The **updateIdMgr Supported EntityType** command updates the configuration that you specify for a supported entity type.	• Parameters: **- name** The name of the supported entity type. The value of this parameter must be one of the supported entity types. (String, required) **- defaultParent** The default parent node for the supported entity type. (String, optional) **- rdnProperties** The RDN attribute name for the supported entity type in the entity domain name. To reset all the values of the rdnProperties parameter, specify a blank string (""). (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrSupported EntityType {-name entity1}` • Using Jython string: `AdminTask.updateIdMgrSupported EntityType ('[-name entity1]')` • Using Jython list: `AdminTask.updateIdMgrSupported EntityType (['-name', 'entity1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrSupported EntityType {-interactive}` • Using Jython string: `AdminTask.updateIdMgrSupported EntityType ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrSupported EntityType (['-interactive'])`

Commands for the IdMgrRepositoryConfig group of the AdminTask object

Use the commands in the IdMgrRepositoryConfig to configure the virtual member manager. The commands for this group do not require a target object. To see the additional commands related to the virtual member manager, see the “Commands for the IdMgrConfig group of the AdminTask object” on page 305 and the “Commands for the IdMgrRealmConfig group of the AdminTask object” on page 398 articles.

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the IdMgrRepositoryConfig group of the AdminTask object:

Table 4.

Command name:	Description:	Parameters and return values:	Examples:
addIdMgrLDAP BackupServer	The **addIdMgrLDAP BackupServer** command adds or updates backup LDAP servers.	• Parameters: **- id** The ID of the repository. (String, required) **- primary_host** The host name for the primary LDAP server. (String, required) **- host** The host name for the LDAP server. (String, required) **- port** The port number for the LDAP server. (Integer, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPBackup Server {-id id1 -primary_ host myprimaryhost -host myhost.ibm.com}` • Using Jython string: `AdminTask.addIdMgrLDAPBackup Server ('[-id id1 -primary_host myprimaryhost -host myhost.ibm.com]')` • Using Jython list: `AdminTask.addIdMgrLDAPBackup Server (['-id', 'id1', '-primary_host', 'myprimary host', '-host', 'myhost. ibm.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAP BackupServer {-interactive}` • Using Jython string: `AdminTask.addIdMgrLDAPBack upServer ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrLDAPBack upServer (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
addIdMgr LDAPEntity Type	The **addIdMgr LDAPEntity Type** command adds an LDAP entity type definition.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the entity type. (String, required) **- searchFilter** The search filter that you want to use to search the entity type. (String, optional) **- objectClasses** One or more object classes for the entity type. (String, required) **- objectClassesForCreate** The object class to use when an entity type is created. If the value of this parameter is the same as the objectClass parameter, you do not need to specify this parameter. (String, optional) **- searchBases** The search base or bases to use while searching the entity type. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPEntity Type {-id id1 -name name1 -objectClasses objectclass}` • Using Jython string: `AdminTask.addIdMgrLDAPEntity Type ('[-id id1 -name name1 -objectClasses objectclass]')` • Using Jython list: `AdminTask.addIdMgrLDAPEntity Type (['-id', 'id1', '-name', 'name1', '-obje ctClasses', 'objectclass'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAP EntityType {-interactive}` • Using Jython string: `AdminTask.addIdMgrLDAP EntityType ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrLDAP EntityType (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
addId MgrLDAP EntityType RDNAttr	The **addId MgrLDAP EntityType RDNAttr** command adds RDN attribute configuration to an LDAP entity type definition.	• Parameters: **- id** The ID of the repository. (String, required) **- entityTypeName** The name of the entity type. (String, required) **- name** The attribute name that is used to build the relative distinguished name (RDN) for the entity type. (String, required) **- objectClass** The object class to use for the entity type for the relative distinguished name (RDN) attribute name that you specify. Use this parameter to map one entity type to multiple structural object classes. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPEntity TypeRDNAttr {-id id1 -entityTypeName entitytype -name name1}` • Using Jython string: `AdminTask.addIdMgrLDAPEntity TypeRDNAttr ('[-id id1 -entityTypeName entitytype -name name1]')` • Using Jython list: `AdminTask.addIdMgrLDAPEntity TypeRDNAttr (['-id', 'id1', '-entityTypeName', 'entity type', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPEntity TypeRDNAttr {-interactive}` • Using Jython string: `AdminTask.addIdMgrLDAPEntity TypeRDNAttr ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrLDAPEntity TypeRDNAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
addIdMgr LDAPGroup Dynamic Member Attr	The **addIdMgr LDAPGroup Dynamic Member Attr** command adds a dynamic member attribute configuration to an LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the LDAP attribute that is used as the group member attribute. For example, `member` or `uniqueMember`. (String, required) **- objectClass** The group object class that contains the member attribute. For example, `groupOfNames` or `groupOfUnqiueNames`. If you do not define this parameter, the member attribute applies to all group object classes. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPGroup DynamicMemberAttr {-id id1 -name name1 -objectClass objectclass}` • Using Jython string: `AdminTask.addIdMgrLDAPGroup DynamicMemberAttr ('[-id id1 -name name1 -objectClass objectclass]')` • Using Jython list: `AdminTask.addIdMgrLDAPGroup DynamicMemberAttr (['-id', 'id1', '-name', 'name1', '-objectClass', 'objectclass'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPGroup DynamicMemberAttr {-interactive}` • Using Jython string: `AdminTask.addIdMgrLDAPGroup DynamicMemberAttr ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrLDAPGroup DynamicMemberAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- scope The scope of the member attribute. The valid values for this parameter include the following: • direct - The member attribute only contains direct members, therefore, this value refers to the member directly contained by the group and not contained through the nested group. For example, if Group1 contains Group2 and Group2 contains User1, then Group2 is a direct member of Group1 but User1 is not a direct member of Group1. Both `member` and `uniqueMember` are direct member attributes. • nested - The member attribute that contains the direct members and the nested members.	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		• all - The member attribute that contains the direct members, the nested members, and the dynamic members. For example, the ibm-allMembers attribute which is supported by the IBM Tivoli Directory Server. (String, optional) **- dummyMember** Indicates that if you create a group without specifying a member, a dummy member will be filled in to avoid creating an exception about missing a mandatory attribute. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
addIdMgr LDAPGroup MemberAttr	The **addIdMgr LDAPGroup MemberAttr** command adds a member attribute configuration to an LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the LDAP attribute that is used as the group member attribute. For example, `member` or `uniqueMember`. (String, required) **- objectClass** The group object class that contains the member attribute. For example, `groupOfNames` or `groupOfUnqiueNames`. If you do not define this parameter, the member attribute applies to all group object classes. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPGroup MemberAttr {-id id1 -name name1}` • Using Jython string: `AdminTask.addIdMgrLDAPGroup MemberAttr ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.addIdMgrLDAPGroup MemberAttr (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPGroup MemberAttr {-interactive}` • Using Jython string: `AdminTask.addIdMgrLDAPGroup MemberAttr ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrLDAPGroup MemberAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- scope The scope of the member attribute. The valid values for this parameter include the following: • direct - The member attribute only contains direct members, therefore, this value refers to the member directly contained by the group and not contained through the nested group. For example, if Group1 contains Group2 and Group2 contains User1, then Group2 is a direct member of Group1 but User1 is not a direct member of Group1. Both `member` and `uniqueMember` are direct member attributes. • nested - The member attribute that contains the direct members and the nested members.	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		• all - The member attribute that contains the direct members, the nested members, and the dynamic members. For example, the ibm-allMembers attribute which is supported by the IBM Tivoli Directory Server. (String, optional) **- dummyMember** Indicates that if you create a group without specifying a member, a dummy member will be filled in to avoid creating an exception about missing a mandatory attribute. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
addId MgrLDAP Server	The **addId MgrLDAP Server** command adds an LDAP server to the LDAP repository ID that you specify.	• Parameters: **- id** The ID of the repository. (String, required) **- host** The host name for the primary LDAP server. (String, required) **- port** The port number for the LDAP server. (Integer, optional) **- bindDN** The binding distinguished name for the LDAP server. (String, optional) **- bindPassword** The binding password. (String, optional) **- authentication** Indicates the authentication method to use. The default value is `simple`. Valid values include: `none` or `strong`. (String, optional) **- referal** The LDAP referral. The default value is `ignore`. Valid values include: `follow`, `throw`, or `false`. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAPServer {-id id1 -host myhost.ibm.com}` • Using Jython string: `AdminTask.addIdMgrLDAPServer ('[-id id1 -host myhost.ibm.com]')` • Using Jython list: `AdminTask.addIdMgrLDAPServer (['-id', 'id1', '-host', 'myhost.ibm.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrLDAP Server {-interactive}` • Using Jython string: `AdminTask.addIdMgrLDAP Server ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrLDAP Server (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- derefAliases Controls how aliases are dereferenced. The default value is `always`. Valid values include: • `never` - never deference aliases • `finding` - deferences aliases only during name resolution • `searching` - deferences aliases only after name resolution (String, optional) **- sslEnabled** Indicates to enable SSL or not. The default value is `false`. (Boolean, optional) **- connectionPool** The connection pool. The default value is `false`. (Boolean, optional) **- connectTimeout** The connection timeout in seconds. The default value is `0`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- ldapServerType The type of LDAP server being used. The default value is `IDS51`. (String, optional) **- sslConfiguration** The SSL configuration. (String, optional) **- certificateMapMode** Specifies whether to map X.509 certificates into a LDAP directory by exact distinguished name or by certificate filter. The default value is `exactdn`. To use the certificate filter for the mapping, specify `certificatefilter`. (String, optional) **- certificateFilter** If `certificateMapMode` has the value `certificatefilter`, then this property specifies the LDAP filter which maps attributes in the client certificate to entries in LDAP. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
addIdMgr Repository BaseEntry	The **addIdMgr Repository BaseEntry**command adds a base entry to the specified repository.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The distinguished name of a base entry. (String, required) **- nameInRepository** The distinguished name in the repository that uniquely identifies the base entry name. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrRepository BaseEntry {-id id1 -name name1}` • Using Jython string: `AdminTask.addIdMgrRepository BaseEntry ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.addIdMgrRepository BaseEntry (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrRepository BaseEntry {-interactive}` • Using Jython string: `AdminTask.addIdMgrRepository BaseEntry ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrRepository BaseEntry (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createId MgrDB Repository	The **createId MgrDB Repository** command creates a database repository configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- dataSourceName** The name of the data source. The default value is `jdbc/wimDS`. (String, required) **- databaseType** The type of the database. The default value is `DB2`. (String, required) **- dbURL** The URL of the database. (String, required) **- dbAdminId** The database administrator ID. (String, required if database type is not Cloudscape.) **- dbAdminPassword** The database administrator password. (String, required if database type is not Cloudscape.) **- adapterClassName** The default value is `com.ibm.ws.wim.adapter.db.DBAdapter`. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createIdMgrDB Repository {-id id1 -dataSourceName datasource name -databaseType DB2}` • Using Jython string: `AdminTask.createIdMgrDB Repository ('[-id id1 -dataSourceName data sourcename -databaseType DB2]')` • Using Jython list: `AdminTask.createIdMgrDB Repository (['-id', 'id1', '-dataSourceName', 'data sourcename', '-databaseType', 'DB2'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createIdMgrDB Repository {-interactive}` • Using Jython string: `AdminTask.createIdMgrDB Repository ('[-interactive]')` • Using Jython list: `AdminTask.createIdMgrDB Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- JDBCDriverClass The JDBC driver class name. (String, optional) **- supportSorting** Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional) **- supportTransaction** Indicates if transactions are supported or not. The default value is `false`. (Boolean, optional) **- isExtIdUnique** Specifies if the external ID is unique. The default value is `true`. (Boolean, optional) **- supportExternalName** Indicates if external names are supported or not. The default value is `false`. (Boolean, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- entityRetrievalLimit Indicates the value of the retrieval limit on database entries. The default value is `200`. (Integer, optional) **- saltLength** The salt length in bits. The default value is `12`. (Integer, optional) **- encryptionKey** The default value is `rZ15ws0ely9yHk3zCs3sTMv/ho8fY17s`. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createId MgrFile Repository	The **createId MgrFile Repository** command creates a file repository configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- messageDigest Algorithm** The message digest algorithm that will be used for hashing the password. The default value is `SHA-1`. Valid values include the following: `SHA-245`, `SHA-384`, or `SHA-512`.(String, required) **- adapterClassName** The default value is `com.ibm.ws.wim.adapter.file.was.FileAdapter`. (String, optional) **- supportPaging** Indicates if paging is supported or not. The default value is `false`. (Boolean, optional) **- supportSorting** Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createIdMgrFile Repository {-id id1 -messageDigestAlgorithm SHA-245}` • Using Jython string: `AdminTask.createIdMgrFile Repository ('[-id id1 -messageDigestAlgorithm SHA-245]')` • Using Jython list: `AdminTask.createIdMgrFile Repository (['-id', 'id1', '-messageDigestAlgorithm', 'SHA-245'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createIdMgrFile Repository {-interactive}` • Using Jython string: `AdminTask.createIdMgrFile Repository ('[-interactive]')` • Using Jython list: `AdminTask.createIdMgrFile Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- supportTransaction Indicates if transaction is supported or not. The default value is `false`. (Boolean, optional) **- isExtIdUnique** Specifies if the external ID is unique or not. The default value is `true`. (Boolean, optional) **- supportExternalName** Indicates if external names are supported or not. The default value is `false`. (Boolean, optional) **- baseDirectory** The base directory where the fill will be created in order to store the data. The default is to be dynamically built during run time using user.install.root and cell name. (String, optional) **- fileName** The file name of the repository. The default value is `fileRegistry.xml`. (String, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- saltLength The salt length of the randomly generated salt for password hashing. The default value is 12. (Integer, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createId MgrLDAP Repository	The **create IdMgrLDAP Repository** command creates an LDAP repository configuration.	• Parameters: **- id** The unique identifier for the repository. (String, required) **- ldapServerType** The type of LDAP server that is being used. The default value is `IDS51`. (String, required) **- adapterClassName** The default value is `com.ibm.ws.wim.adapter.db.DBAda pter`. (String, optional) **- supportSorting** Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional) **- supportPaging** Indicates if paging is supported or not. The default value is `false`. (Boolean, optional) **- supportTransaction** Indicates if transactions are supported or not. The default value is `false`. (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createIdMgrLDAP Repository {-id id1 -ldapServerType IDS51}` • Using Jython string: `AdminTask.createIdMgrLDAP Repository ('[-id id1 -ldapServerType IDS51]')` • Using Jython list: `AdminTask.createIdMgrLDAP Repository (['-id', 'id1', '-ldapServerType', 'IDS51'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createIdMgrLDAP Repository {-interactive}` • Using Jython string: `AdminTask.createIdMgrLDAP Repository ('[-interactive]')` • Using Jython list: `AdminTask.createIdMgrLDAP Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- isExtIdUnique Specifies if the external ID is unique. The default value is `true`. (Boolean, optional) **- supportExternalName** Indicates if external names are supported or not. The default value is `false`. (Boolean, optional) **- authentication** Indicates the authentication method to use. The default value is `simple`. Valid values include: `none` or `strong`. (String, optional) **- referal** The LDAP referral. The default value is `ignore`. Valid values include: `follow`, `throw`, or `false`. (String, optional) **- sslEnabled** Indicates to enable SSL or not. The default value is `false`. (Boolean, optional) **- sslConfiguration** The SSL configuration. (String, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- connectionPool The connection pool. The default value is `false`. (Boolean, optional) **- translateRDN** Indicates to translate RDN or not. The default value is `false`. (Boolean, optional) **- searchTimeLimit** The value of search time limit. (Integer, optional) **- searchCountLimit** The value of search count limit. (Integer, optional) **- searchPageSize** The value of search page size. (Integer, optional) **- returnToPrimaryServer** (Integer, optional) **- primaryServerQueryTimeInterval** (Integer, optional) **- default** If you set this parameter to `true`, the default values will be set for the remaining configuration properties of the LDAP repository. (Boolean, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteId MgrLDAP EntityType	The **deleteId MgrLDAP EntityType** command deletes the LDAP entity type configuration data for a specified entity type for a specific LDAP repository.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the entity type. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP EntityType {-id id1 -name name1}` • Using Jython string: `AdminTask.deleteIdMgrLDAP EntityType ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP EntityType (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP EntityType {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrLDAP EntityType ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP EntityType (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteId MgrLDAP EntityType RDNAttr	The **deleteId MgrLDAP EntityType RDNAttr** command deletes the relative distinguished name (RDN) attribute configuration from an LDAP entity type configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- entityTypeName** The name of the entity type. (String, required) **- name** The attribute name that is used to build the relative distinguished name (RDN) for the entity type. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAPEntity TypeRDNAttr {-id id1 -name name1 -entityTypeName entityType}` • Using Jython string: `AdminTask.deleteIdMgrLDAP EntityTypeRDNAttr ('[-id id1 -name name1 -entityType Name entityType]')` • Using Jython list: `AdminTask.deleteIdMgrLDAPEntity TypeRDNAttr (['-id', 'id1', '-name', 'name1', '-entity TypeName', 'entityType'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAPEntity TypeRDNAttr {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrLDAPEntity TypeRDNAttr ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrLDAPEntity TypeRDNAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteId MgrLDAP GroupConfig	The **deleteId MgrLDAP GroupConfig** command deletes the LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP GroupConfig {-id id1}` • Using Jython string: `AdminTask.deleteIdMgrLDAP GroupConfig ('[-id id1]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP GroupConfig (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP GroupConfig {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrLDAP GroupConfig ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP GroupConfig (['-interactive'])`
deleteIdMgr LDAPGroup MemberAttr	The **deleteIdMgr LDAPGroup MemberAttr** command deletes a member attribute configuration from an LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP GroupMemberAttr {-id id1}` • Using Jython string: `AdminTask.deleteIdMgrLDAP GroupMemberAttr ('[-id id1]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP GroupMemberAttr (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP GroupMemberAttr {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrLDAP GroupMemberAttr ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP GroupMemberAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteIdMgr LDAPGroup Dynamic MemberAttr	The **deleteIdMgr LDAPGroup Dynamic MemberAttr** command deletes a dynamic member attribute configuration from an LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the LDAP attribute that is used as the group member attribute. For example, `memberURL`. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP GroupDynamicMemberAttr {-id id1 -name name1}` • Using Jython string: `AdminTask.deleteIdMgrLDAP GroupDynamicMemberAttr ('[ -id id1 -name name1]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP GroupDynamicMemberAttr (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAPGroup DynamicMemberAttr {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrLDAPGroup DynamicMemberAttr ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrLDAPGroup DynamicMemberAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteId MgrLDAP Server	The **deleteId MgrLDAP Server** command deletes the configuration for the LDAP server that you specify from the LDAP repository ID that you specify.	• Parameters: **- id** The ID of the repository. (String, required) **- host** The host name for the primary LDAP server. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP Server {-id id1 -host myhost.ibm.com}` • Using Jython string: `AdminTask.deleteIdMgrLDAP Server ('[-id id1 -host myhost.ibm.com]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP Server (['-id', 'id1', '-host', 'myhost.ibm.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrLDAP Server {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrLDAP Server ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrLDAP Server (['-interactive'])`
deleteIdMgr Repository	The **deleteIdMgr Repository** command deletes a repository that you specify.	• Parameters: **- id** The ID of the repository. Valid values include existing repository IDs. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgr Repository {-id id1}` • Using Jython string: `AdminTask.deleteIdMgr Repository ('[-id id1]')` • Using Jython list: `AdminTask.deleteIdMgr Repository (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRepos itory {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrReposi tory ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrReposi tory (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteIdMgr Repository BaseEntry	The **deleteIdMgr Repository BaseEntry** command deletes a base entry from the specified repository.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The distinguished name of a base entry. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRepository BaseEntry {-id id1 -name name1}` • Using Jython string: `AdminTask.deleteIdMgrRepository BaseEntry ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.deleteIdMgrRepository BaseEntry (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRepository BaseEntry {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrRepository BaseEntry ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrRepository BaseEntry (['-interactive'])`
getIdMgr LDAPAttr Cache	The **getIdMgr LDAPAttr Cache** command returns the LDAP attribute cache configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map with the parameters of the **setIdMgr LDAPAttr Cache** command as keys.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPAttr Cache {-id id1}` • Using Jython string: `AdminTask.getIdMgrLDAPAttr Cache ('[-id id1]')` • Using Jython list: `AdminTask.getIdMgrLDAPAttr Cache (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAP AttrCache {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPAttr Cache ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPAttr Cache (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgr LDAP ContextPool	The **getIdMgr LDAP Context Pool** command returns the LDAP context pool configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map that includes the parameters of the **setIdMgrLDAPContextPool** command as the keys.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPContext Pool {-id id1}` • Using Jython string: `AdminTask.getIdMgrLDAPContext Pool ('[-id id1]')` • Using Jython list: `AdminTask.getIdMgrLDAPContext Pool (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPCon textPool {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPCon textPool ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPCon textPool (['-interactive'])`
getIdMgr LDAP EntityType	The **getIdMgr LDAP EntityType** command returns the LDAP entity type configuration data.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the entity type. (String, required) • Returns: A hash map with the keys the same as the property name of the **addIdMgr LDAP EntityType** command. Multi-valued parameters are returned as list.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPEntity Type {-id id1 -name name1}` • Using Jython string: `AdminTask.getIdMgrLDAPEntity Type ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.getIdMgrLDAPEntity Type (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPEn tityType {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPEn tityType ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPEn tityType (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgr LDAPEntity TypeRDNAttr	The **getIdMgr LDAPEntity TypeRDNAttr** command returns the relative distinguished name (RDN) attribute configuration for an LDAP entity type definition.	• Parameters: **- id** The ID of the repository. (String, required) **- entityTypeName** The name of the entity name. (String, required) • Returns: A hash map with the RDN attribute names as the key. If the object class is set, the value of the key will be set to the value of the object class. Otherwise, the value will be null.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPEntity TypeRDNAttr {-id id1 -entityTypeName name1}` • Using Jython string: `AdminTask.getIdMgrLDAPEntity TypeRDNAttr ('[-id id1 -entityTypeName name1]')` • Using Jython list: `AdminTask.getIdMgrLDAPEntity TypeRDNAttr (['-id', 'id1', '-entityTypeName', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPEntity TypeRDNAttr {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPEntity TypeRDNAttr ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPEntity TypeRDNAttr (['-interactive'])`
getIdMgr LDAPGroupConfig	The **getIdMgr LDAPGroupConfig** command returns the LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map that contains the parameters of the **setIdMgr LDAPGroup Config** command as the keys.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPGroup Config {-id id1}` • Using Jython string: `AdminTask.getIdMgrLDAPGroup Config ('[-id id1]')` • Using Jython list: `AdminTask.getIdMgrLDAPGroup Config (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPGroup Config {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPGroup Config ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPGroup Config (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgr LDAPGroup Dynamic Member Attrs	The **getIdMgr LDAPGroup Dynamic Member Attrs** command returns the dynamic member attribute configuration from the LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map list that contains the parameters of the **addIdMgr LDAPGroup Dynamic MemberAttr** command as the keys.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPGroupDynamic MemberAttrs {-id id1}` • Using Jython string: `AdminTask.getIdMgrLDAPGroupDynamic MemberAttrs ('[-id id1]')` • Using Jython list: `AdminTask.getIdMgrLDAPGroupDynamic MemberAttrs (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPGroup DynamicMemberAttrs {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPGroup DynamicMemberAttrs ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPGroup DynamicMemberAttrs (['-interactive'])`
getIdMgr LDAPGroup MemberAttrs	The **getIdMgr LDAPGroup MemberAttrs** command returns the member attribute configuration for the LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map list that contains the parameters of the **addIdMgr LDAPGroup MemberAttr** command as the keys.	**Batch mode example usage:** **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPGroup MemberAttrs {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPGroup MemberAttrs ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPGroup MemberAttrs (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgr LDAPSearch ResultCache	The **getIdMgr LDAPSearch ResultCache** command returns the LDAP search result cache configuration.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map with the parameters of the **setIdMgr LDAPSearch ResultCache** command as the keys.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAP SearchResultCache {-id id1}` • Using Jython string: `AdminTask.getIdMgrLDAP SearchResultCache ('[-id id1]')` • Using Jython list: `AdminTask.getIdMgrLDAPSear chResultCache (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPSea rchResultCache {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPSear chResultCache ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPSe archResultCache (['-interactive'])`
getIdMgr LDAPServer	The **getIdMgr LDAPServer** command returns the configuration for the LDAP server that you specify for the LDAP repository ID that you specify.	• Parameters: **- id** The ID of the repository. (String, required) **- host** The host name for the primary LDAP server. (String, required) • Returns: A hash map with the keys the same as the parameter names for the **addIdMgr LDAPServer** command.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPServer {-id id1 -host myhost.ibm.com}` • Using Jython string: `AdminTask.getIdMgrLDAPServer ('[-id id1 -host myhost.ibm.com]')` • Using Jython list: `AdminTask.getIdMgrLDAPServer (['-id', 'id1', '-host', 'myhost.ibm.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrLDAPServer {-interactive}` • Using Jython string: `AdminTask.getIdMgrLDAPServer ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrLDAPServer (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgr Repository	The **getIdMgr Repository** command returns the configuration of the specified repository.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map. The keys will vary depending on the type of repository.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrRepos itory {-id id1}` • Using Jython string: `AdminTask.getIdMgrReposi tory ('[-id id1]')` • Using Jython list: `AdminTask.getIdMgrReposi tory (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrRepos itory {-interactive}` • Using Jython string: `AdminTask.getIdMgrReposi tory ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrReposi tory (['-interactive'])`
listIdMgr Custom Properties	The **listIdMgr Custom Properties**command returns a list of custom properties for the repository that you specify.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map that contains keys as the custom property names and values as custom property values.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrCustomPro perties {-id id1}` • Using Jython string: `AdminTask.listIdMgrCustom Properties ('[-id id1]')` • Using Jython list: `AdminTask.listIdMgrCustom Properties (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrCustom Properties {-interactive}` • Using Jython string: `AdminTask.listIdMgrCustom Properties ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrCustom Properties (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMgr LDAPBackupServers	The **listIdMgr LDAPBackupServers**command returns a list of the backup LDAP server or servers.	• Parameters: **- id** The ID of the repository. (String, required) **- primary_host** The host name for the primary LDAP server. (String, required) • Returns: A list of hash maps. The hash maps contains the names of the backup servers as the key and the port numbers as the value. Returning all the data in a hash map does not maintain the order of backup servers.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrLDAPBackupServer {-id id1 -primary_host hostname}` • Using Jython string: `AdminTask.listIdMgrLDAPBackupServer ('[-id id1 -primary_host hostname]')` • Using Jython list: `AdminTask.listIdMgrLDAPBackupServer (['-id', 'id1', '-primary_host', 'hostname'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrLDAPBackupServer {-interactive}` • Using Jython string: `AdminTask.listIdMgrLDAPBackupServer ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrLDAPBackupServer (['-interactive'])`
listIdMgr LDAPEntityTypes	The **listIdMgr LDAPEntityTypes** command lists the name of all of the configured LDAP entity type definitions.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A list that contains the names of the configured LDAP entity types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrLDAPEntityType {-id id1}` • Using Jython string: `AdminTask.listIdMgrLDAPEntityType ('[-id id1]')` • Using Jython list: `AdminTask.listIdMgrLDAPEntityType (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrLDAPEntityType {-interactive}` • Using Jython string: `AdminTask.listIdMgrLDAPEntityType ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrLDAPEntityType (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMgr LDAP Servers	The **listIdMgr LDAP Servers** command lists all of the configured primary LDAP servers.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A list that contains the primary LDAP server names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrLDAP Servers {-id id1}` • Using Jython string: `AdminTask.listIdMgrLDAP Servers ('[-id id1]')` • Using Jython list: `AdminTask.listIdMgrLDAP Servers (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrLDAP Servers {-interactive}` • Using Jython string: `AdminTask.listIdMgrLDAP Servers ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrLDAP Servers (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMgr Repositories	The **listIdMgr Repositories** command lists names and types of all configured repositories.	• Parameters: None • Returns: A hash map with key as the name of the repository and value as another hash map that includes the following keys: – repositoryType - The type of repository. For example, File, LDAP, DB, and so on. – specificRepositoryType - The specific type of repository. For example, LDAP, IDS51, NDS, and so on. – host - The host name where the repository resides. For File, it is LocalHost and for DB it is dataSourceName. This command will not return the Property Extension and Entry Mapping repository data.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrRepositories` • Using Jython string: `AdminTask.listIdMgrRepositories()` • Using Jython list: `AdminTask.listIdMgrRepositories()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrReposi tories {-interactive}` • Using Jython string: `AdminTask.listIdMgrReposit ories ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrReposi tories (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMgr Repository BaseEntries	The **listIdMgr Repository BaseEntries** command lists the base entries for a specified repository.	• Parameters: **- id** The ID of the repository. (String, required) • Returns: A hash map that contains base entry name as key and `nameInRepository` as value.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrRepository BaseEntries {-id id1}` • Using Jython string: `AdminTask.listIdMgrRepository BaseEntries ('[-id id1]')` • Using Jython list: `AdminTask.listIdMgrRepository BaseEntries (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrRepository BaseEntries {-interactive}` • Using Jython string: `AdminTask.listIdMgrRepository BaseEntries ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrRepository BaseEntries (['-interactive'])`
listIdMgr Supported DBTypes	The **listIdMgr Supported DBTypes** command returns a list of supported database types.	• Parameters: None • Returns: A list of supported database types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupportedDBTypes` • Using Jython string: `AdminTask.listIdMgrSupportedDBTypes()` • Using Jython list: `AdminTask.listIdMgrSupportedDBTypes()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupported DBTypes {-interactive}` • Using Jython string: `AdminTask.listIdMgrSupported DBTypes ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrSupported DBTypes (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMg rSupported Message Digest Algorithms	The **listIdMgr Supported Message Digest Algorithms** command returns a list of supported message digest algorithms.	• Parameters: None • Returns: A list of supported message digest algorithms.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupported MessageDigestAlgorithms` • Using Jython string: `AdminTask.listIdMgrSupported MessageDigestAlgorithms()` • Using Jython list: `AdminTask.listIdMgrSupported MessageDigestAlgorithms()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupportedMes sageDigestAlgorithms {-interactive}` • Using Jython string: `AdminTask.listIdMgrSupportedMessage DigestAlgorithms ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrSupportedMessage DigestAlgorithms (['-interactive'])`
listIdMgr Supported LDAPSe rverTypes	The **listIdMgr Supported LDAP ServerTypes** command returns a list of supported LDAP server types.	• Parameters: None • Returns: A list of supported LDAP server types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupported LDAPServerTypes` • Using Jython string: `AdminTask.listIdMgrSupported LDAPServerTypes()` • Using Jython list: `AdminTask.listIdMgrSupported LDAPServerTypes()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrSupported LDAPServerTypes {-interactive}` • Using Jython string: `AdminTask.listIdMgrSupported LDAPServerTypes ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrSupported LDAPServerTypes (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
removeIdMgr LDAPBackupServer	The **removeIdMgr LDAPBackupServer**command removes the backup LDAP server or servers.	• Parameters: **- id** The ID of the repository. (String, required) **- primary_host** The host name for the primary LDAP server. (String, required) **- host** The name of the backup host name. Use a asterisk (*) if you want to remove all backup servers. (String, required) **- port** The port number of the LDAP server. (Integer, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeIdMgrLDAPBackupServer {-id id1 -primary_host myprimaryhost.ibm.com -host myhost.ibm.com}` • Using Jython string: `AdminTask.removeIdMgrLDAPBackupServer ('[-id id1 -primary_host myprimaryhost.ibm.com -host myhost.ibm.com]')` • Using Jython list: `AdminTask.removeIdMgrLDAPBackupServer (['-id', 'id1', '-primary_host', 'myprimaryhost.ibm.com', '-host', 'myhost.ibm.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeIdMgrLDAPBackupServer {-interactive}` • Using Jython string: `AdminTask.removeIdMgrLDAPBackupServer ('[-interactive]')` • Using Jython list: `AdminTask.removeIdMgrLDAPBackupServer (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgr Custom Property	The **setIdMgr Custom Property** command adds the custom properties to a repository configuration.	• Parameters: **- id** The ID of the repository. Valid values include the existing repository IDs. (String, required) **- name** The name of the additional property for the repository that are not defined OOTB.(String, required) **- value** If this parameter is an empty string, the property will be deleted from the repository configuration. If this parameter is not an empty string and name does not exist, it will be added. If name is an empty string, all the custom properties will be deleted. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrCustomProperty {-id id1 -name name1 -value value}` • Using Jython string: `AdminTask.setIdMgrCustomProperty ('[-id id1 -name name1 -value value]')` • Using Jython list: `AdminTask.setIdMgrCustomProperty (['-id', 'id1', '-name', 'name1', '-value', 'value'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrCustom Property {-interactive}` • Using Jython string: `AdminTask.setIdMgrCustom Property ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrCustom Property (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgr LDAPA ttrCache	The **setIdMgr LDAPA ttrCache** command configures the LDAP attribute cache configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- cachesDiskOffLoad** (String, optional) **- enabled** Indicates if you want to enable attribute caching. The default value is `true`. (Boolean, optional) **- cacheSize** The maximum size of the attribute cache defined by the number of attribute objects that are permitted in the attribute cache. The minimum value of this parameter is `100`. The default value is `4000`. (Integer, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPAttrCache {-id id1}` • Using Jython string: `AdminTask.setIdMgrLDAPAttrCache ('[-id id1]')` • Using Jython list: `AdminTask.setIdMgrLDAPAttrCache (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPAttrCache {-interactive}` • Using Jython string: `AdminTask.setIdMgrLDAPAttrCache ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrLDAPAttrCache (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- cacheTimeOut The amount of time in seconds before the cached entries that are located in the attributes cache can be not valid. The minimum value of this parameter is `0`. The attribute objects that are cached will remain in the attributes cache until the virtual member manager changes the attribute objects. The default value is `1200`. (Integer, optional) **- attributeSizeLimit** An integer that represents the maximum number of attribute object values that can cache in the attributes cache. Some attributes, for example, the member attribute, contain many values. The attributeSizeLimit parameter prevents the attributes cache to cache large attributes. The default value is `2000`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- serverTTLAttribute The name of the ttl attribute that is supported by the LDAP server. The attributes cache uses the value of this attribute to determine when the cached entries in the attributes cache will time out. The ttl attribute contains the time, in seconds, that any information from the entry should be kept by a client before it is considered stale and a new copy is fetched. A value of `0` implies that the object will not be cached. For more information about this attribute, go to: http://www.ietf.org/proceedings/98aug/I-D/draft-ietf-asid-ldap-cache-01.txt. The ttl attribute is not supported by all LDAP servers. If this attribute is supported by an LDAP server, you can set the value of the serverTTLAttribute parameter to the name of the ttl attribute in order to allow the value of the ttyl attribute to determine when cached entries will time out. The time out value for different entries in attributes cache can be different.	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		For example, if the value of the serverTTLAttribute parameter is ttl and the attributes cache retrieves attributes of a user from an LDAP server, it will also retrieve the value of the ttl attribute of this user. If the value is 200, the WMM uses this value to set the time out for the attributes of the user in the attributes cache instead of using the value of cacheTimeout. You can set different ttl attribute values for different users. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgrLDAP ContextPool	The **setIdMgr LDAPContextPool**command sets up the LDAP context pool configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- enabled** By default, the context pool is enabled. If you set this parameter to `false`, the context pool is disabled. When the context pool is disabled, new context instances will be created for each request. The default value is `true`. (Boolean, optional) **- initPoolSize** The number of context instances that the the virtual member manager LDAP adapter creates when it creates the pool. The valid range for this parameter is `1` to `50`. The default value is `1`. (Integer, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPContextPool {-id id1}` • Using Jython string: `AdminTask.setIdMgrLDAPContextPool ('[-id id1]')` • Using Jython list: `AdminTask.setIdMgrLDAPContextPool (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPContextPool {-interactive}` • Using Jython string: `AdminTask.setIdMgrLDAPContextPool ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrLDAPContextPool (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- maxPoolSize The maximum number of context instances that the context pool will maintain. Context instances that are in use and those that are idle contribute to this number. When the pool size reaches this number, new context instances cannot be created for new requests. The new request is blocked until a context instance is released by another request or is removed. The request checks periodically if there are context instances available in the pool according to the amount of time that you specify using the poolWaitTime parameter. The minimum value for this parameter is `0`. There is no maximum value. Setting the value of this parameter to `0` means that there is no maximum size and a request for a pooled context instance will use an existing pooled idle context instance or a newly created pooled context instance. The default value is `20`.(Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- prefPoolSize The preferred number of context instances that the context pool will maintain. Context instances that are in use and those that are idle contribute to this number. When there is a request for the use of a pooled context instance and the pool size is less than the preferred size, the context pool creates and uses a new pooled context instance regardless of whether an idle connection is available. When a request finishes with a pooled context instance and the pool size is greater than the preferred size, the context pool closes and removes the pooled context instance from the pool. The valid range for this parameter is from `0` to `100`. Setting the value of this parameter to `0` means that there is no preferred size and a request for a pooled context instance results in a newly created context instance only if no idle ones are available. The default value is `3`.(Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- poolTimeOut An integer that represents the number of milliseconds that an idle context instance may remain in the pool without being closed and removed from the pool. When a context instance is requested from the pool, if this context already exists in the pool for more than the time defined by poolTimeout, this connection will be closed no matter this context instance is stale or active. A new context instance will be created and put back to the pool after it has been released from the request. The minimum value for this parameter is `0`. There is no maximum value. Setting the value of this parameter to `0` means that the context instances in the pool will remain in the pool until they are staled. The context pool catches the communication exception and recreates a new context instance. The default value is `0`.(Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- poolWaitTime The time interval in milliseconds that the request waits until the context pool rechecks if there are idle context instances available in the pool when the number of context instances reaches the maximum pool size. If no idle context instance, the request will continue waiting for the same period of time until next checking. The minimum value for the poolWaitout parameter is `0`. There is no maximum value. A value of `0` for this parameter means that the context pool will not check if idle context exists. The request will be notified when a context instance releases from other requests. The default value is `3000`.(Integer, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgr LDAPG roupConfig	The **setIdMgr LDAPGr oupConfig** command sets up the LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- updateGroup Membership** Updates the group membership if the member is deleted or renamed. Some LDAP servers, for example, Domino server, do not clean up the membership of the user when a user is deleted or renamed. If you choose these LDAP server types in the ldapServerType property, the value of this parameter is set to `true`. Use this parameter to change the value. The default value is `false`. (Boolean, optional) **- name** The name of the membership attribute. For example, `memberOf` in an active directory server and `ibm-allGroups` in IDS. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAP GroupConfig {-id id1}` • Using Jython string: `AdminTask.setIdMgrLDAP GroupConfig ('[-id id1]')` • Using Jython list: `AdminTask.setIdMgrLDAP GroupConfig (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPGr oupConfig {-interactive}` • Using Jython string: `AdminTask.setIdMgrLDAPGr oupConfig ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrLDAPGroup Config (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- scope The scope of the membership attribute. The following are the possible values for this parameter: • direct - The membership attribute only contains direct groups. Direct groups contain the member and are not contained through a nested group. For example, if group1 contains group2, group2 contains user1, then group2 is a direct group of user1, but group1 is not a direct group of user1. • nested - The membership attribute contains both direct groups and nested groups. • all - The membership attribute contains direct groups, nested groups, and dynamic members. The default value is `direct`. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgr LDAPSearch ResultCache	The **setIdMgr LDAPSearch ResultCache** command sets up the LDAP search result cache configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- cachesDiskOffLoad** Loads the attributes caches and the search results onto hard disk. By default, when the number of cache entries reaches the maximum size of the cache, cache entries are evicted to allow new entries to enter the caches. If you enable this parameter, the evicted cache entries will be copied to disk for future access. The default value is `false`. (Boolean, optional) **- enabled** Enables the search results cache. The default value is `true`. (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPSearch ResultCache {-id id1}` • Using Jython string: `AdminTask.setIdMgrLDAPSearch ResultCache ('[-id id1]')` • Using Jython list: `AdminTask.setIdMgrLDAPSearch ResultCache (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrLDAPSearch ResultCache {-interactive}` • Using Jython string: `AdminTask.setIdMgrLDAPSearch ResultCache ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrLDAPSearch ResultCache (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- cacheSize The maximum size of the search results cache. The number of naming enumeration objects that can be put into the search results cache. The minimum value of this parameter is `100`. The default value is `2000`. (Integer, optional) **- cacheTimeOut** The amount of time in seconds before the cached entries in the search results cache can be not valid. The minimum value for this parameter is `0`. A value of `0` means that the cached naming enumeration objects will stay in the search results cache until there are configuration changes. The default value is `600`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		-searchResultSizeLimit The maximum number of entries contained in the naming enumeration object that can be cached in the search results cache.For example, if the results from a search contains 2000 users, the search results will not cache in the search results cache if the value of the of this property is set to `1000`. The default value is `1000`. (Integer, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgr Entry Mapping Repository	The **setIdMgr Entry Mapping Repository** command sets or updates an entry mapping repository configuration.	• Parameters: **- dataSourceName** The name of the data source. The default value is `jdbc/wimDS`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String) **- databaseType** The type of the database. The default value is `DB2`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String) **- dbURL** The URL of the database. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrEntry MappingRepository {-dbAdminId dbid1 -dbAdminPassword pw1}` • Using Jython string: `AdminTask.setIdMgrEntry MappingRepository ('[-dbAdminId dbid1 -dbAdminPassword pw1]')` • Using Jython list: `AdminTask.setIdMgrEntry MappingRepository (['-dbAdminId', 'dbid1', '-dbAdmin Password', 'pw1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrEntryMapping Repository {-interactive}` • Using Jython string: `AdminTask.setIdMgrEntryMapping Repository ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrEntryMapping Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- dbAdminId The database administrator ID. (String, required if database type is not Cloudscape.) **- dbAdminPassword** The database administrator password. (String, required if database type is not Cloudscape.) **- JDBCDriverClass** The JDBC driver class name. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgr Property Extension Repository	The **setIdMgr Property Extension Repository** command sets or updates the property extension repository configuration.	• Parameters: **- dataSourceName** The name of the data source. The default value is `jdbc/wimDS`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String) **- databaseType** The type of the database. The default value is `DB2`. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String) **- dbURL** The URL of the database. The parameter is required if the property extension is not set. The parameter is not required if the command is used to update the existing configuration. (String)	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrProperty ExtensionRepository {-entity RetrievalLimit 10 -JDBC DriverClass classname}` • Using Jython string: `AdminTask.setIdMgrProperty ExtensionRepository ('[-entity RetrievalLimit 10 -JDBC DriverClass classname]')` • Using Jython list: `AdminTask.setIdMgrProperty ExtensionRepository (['-entity RetrievalLimit', '10', '-JDBCDriverClass', 'classname'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrProperty ExtensionRepository {-interactive}` • Using Jython string: `AdminTask.setIdMgrProperty ExtensionRepository ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrPropertyExtensionRepository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- dbAdminId The database administrator ID. (String, required if database type is not Cloudscape.) **- dbAdminPassword** The database administrator password. (String, required if database type is not Cloudscape.) **- entityRetrievalLimit** The limit for the retrieval of entities. (Integer, required) **- JDBCDriverClass** The JDBC driver class name. (String, required) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateId MgrDB Repository	The **updateId MgrDB Repository** command updates the configuration for the database repository that you specify.	• Parameters: **- id** The ID of the repository. (String, required) **- dataSourceName** The name of the data source. The default value is `jdbc/wimDS`. (String, optional) **- databaseType** The type of the database. The default value is `DB2`. (String, optional) **- dbURL** The URL of the database. (String, optional) **- dbAdminId** The database administrator ID. (String, optional) **- dbAdminPassword** The database administrator password. (String, optional) **- entityRetrievalLimit** Indicates the value of the retrieval limit on database entries. The default value is `200`. (Integer, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrDB Repository {-id id1}` • Using Jython string: `AdminTask.updateIdMgrDB Repository ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrDB Repository (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrDB Repository {-interactive}` • Using Jython string: `AdminTask.updateIdMgrDB Repository ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrDB Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- JDBCDriverClass The JDBC driver class name. (String, optional) **- saltLength** The salt length in bits. The default value is `12`. (Integer, optional) **- encryptionKey** The default value is `rZ15ws0ely9yHk3zCs3sTMv/ho8fY17s`. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateId MgrFile Repository	The **updateId MgrFile Repository** command updates the configuration for the file repository that you specify. To update other properties of the file repository use the **update IdMgr Repository**command.	• Parameters: **- id** The ID of the repository. (String, required) **- messageDigest Algorithm** The message digest algorithm that will be used for hashing the password. The default value is `SHA-1`. Valid values include the following: `SHA-245`, `SHA-384`, or `SHA-512`.(String, optional) **- baseDirectory** The base directory where the fill will be created in order to store the data. The default is to be dynamically built during run time using user.install.root and cell name. (String, optional) **- fileName** The file name of the repository. The default value is `fileRegistry.xml`. (String, optional) **- saltLength** The salt length of the randomly generated salt for password hashing. The default value is `12`. (Integer, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrFile Repository {-id id1}` • Using Jython string: `AdminTask.updateIdMgrFile Repository ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrFile Repository (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrFile Repository {-interactive}` • Using Jython string: `AdminTask.updateIdMgrFile Repository ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrFile Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateId MgrLDAP AttrCache	The **updateId MgrLDAP AttrCache** command updates the LDAP attribute cache configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- cachesDiskOffLoad** (String, optional) **- enabled** Indicates if you want to enable attribute caching. The default value is `true`. (Boolean, optional) **- cacheSize** The maximum size of the attribute cache defined by the number of attribute objects that are permitted in the attribute cache. The minimum value of this parameter is `100`. The default value is `4000`. (Integer, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP AttrCache {-id id1}` • Using Jython string: `AdminTask.updateIdMgrLDAP AttrCache ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrLDAP AttrCache (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP AttrCache {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAP AttrCache ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAP AttrCache (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- cacheTimeOut The amount of time in seconds before the cached entries that are located in the attributes cache can be not valid. The minimum value of this parameter is `0`. The attribute objects that are cached will remain in the attributes cache until the virtual member manager changes the attribute objects. The default value is `1200`. (Integer, optional) **- attributeSizeLimit** An integer that represents the maximum number of attribute object values that can cache in the attributes cache. Some attributes, for example, the member attribute, contain many values. The attributeSizeLimit parameter prevents the attributes cache to cache large attributes. The default value is `2000`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- serverTTLAttribute The name of the ttl attribute that is supported by the LDAP server. The attributes cache uses the value of this attribute to determine when the cached entries in the attributes cache will time out. The ttl attribute contains the time, in seconds, that any information from the entry should be kept by a client before it is considered stale and a new copy is fetched. A value of `0` implies that the object will not be cached. For more information about this attribute, go to: http://www.ietf.org/proceedings/98aug/I-D/draft-ietf-asid-ldap-cache-01.txt. The ttl attribute is not supported by all LDAP servers. If this attribute is supported by an LDAP server, you can set the value of the serverTTLAttribute parameter to the name of the ttl attribute in order to allow the value of the ttyl attribute to determine when cached entries will time out. The time out value for different entries in attributes cache can be different.	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		For example, if the value of the serverTTLAttribute parameter is ttl and the attributes cache retrieves attributes of a user from an LDAP server, it will also retrieve the value of the ttl attribute of this user. If the value is 200, the WMM uses this value to set the time out for the attributes of the user in the attributes cache instead of using the value of cacheTimeout. You can set different ttl attribute values for different users. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateId MgrLDAP ContextPool	The **updateId MgrLDAP ContextPool** command updates the LDAP context pool configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- enabled** By default, the context pool is enabled. If you set the value of this parameter to `false`, the context pool is disabled which means that a new context instance will be created for each request. The default value is `true`. (Boolean, optional) **- initPoolSize** The number of context instances that the virtual member manager LDAP adapter creates when it creates the pool. The valid range for this parameter is `1` to `50`. The default value is `1`. (Integer, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP ContextPool {-id id1}` • Using Jython string: `AdminTask.updateIdMgrLDAP ContextPool ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrLDAP ContextPool (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP ContextPool {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAP ContextPool ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAP ContextPool (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- maxPoolSize The maximum number of context instances that can be maintained concurrently by the context pool. Both in-use and idle context instances contribute to this number. When the pool size reaches this number, new context instances cannot created for new request. The new request is blocked until a context instance is released by another request or is removed. The request checks periodically if there are context instances available in the pool according to the value defined for the poolWaitTime parameter. The minimum value of the maxPoolSize parameter is `0`. There is no maximum value. A maximum pool size of 0 means that there is no maximum size and that a request for a pooled context instance will use an existing pooled idle context instance or a newly created pooled context instance. The default value is `20`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- prefPoolSize The preferred number of context instances that the Context Pool should maintain. Both in-use and idle context instances contribute to this number. When there is a request for the use of a pooled context instance and the pool size is less than the preferred size, Context Pool will create and use a new pooled context instance regardless of whether an idle connection is available. When a request is finished with a pooled context instance and the pool size is greater than the preferred size, the Context Pool will close and remove the pooled context instance from the pool. The valid range of the prefPoolSize parameter is `0` to `100`. A preferred pool size of 0 means that there is no preferred size: A request for a pooled context instance will result in a newly created context instance only if no idle ones are available. The default value is `3`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- poolTimeOut An integer that represents the number of milliseconds that an idle context instance may remain in the pool without being closed and removed from the pool. When a context instance is requested from the pool, if this context already exists in the pool for more than the time defined by poolTimeout, this connection will be closed no matter this context instance is stale or active. A new context instance will be created and put back to the pool after it has been released from the request.The minimum value of poolTimeout is 0. There is no maximum value.A poolTimeout of 0 means that the context instances in the pool will remain in the pool until they are staled. In this case, Context Pool will catch the communication exception and recreate a new context instance. The default value is `0`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- poolWaitTime The time interval (in milliseconds) that the request will wait until the Context Pool checks again if there are idle context instance available in the pool when the number of context instances reaches the maximum pool size. If there is still no idle context instance, the request will continue waiting for the same period of time until next checking. The minimum value of poolWaitout is 0. There is no maximum value. A poolWaitTime of 0 means the Context Pool will not check if there are idle context. Instead, the request will be notified when there is a context instance is released from other requests. The default value is `3000`. (Integer, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateId MgrLDAP EntityType	The **updateId MgrLDAP EntityType** command updates an existing LDAP entity type definition to LDAP repository configuration. You can use this command to add more values to multi-valued parameters. If the property already exists, the value of the property will be replaced. If the property does not exist, it will be added.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the entity type. (String, required) **- searchFilter** The search filter that you want to use to search the entity type. (String, optional) **- objectClasses** One or more object classes for the entity type. (String, optional) **- objectClassesForCreate** The object class that will be when you create an entity type object. You do not have to specify the value of this parameter if it is the same as the value of the objectClasses parameter. (String, optional) **- searchBases** The search base or bases to use while searching the entity type. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAPEntity Type {-id id1 -name name1}` • Using Jython string: `AdminTask.updateIdMgrLDAPEntity Type ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.updateIdMgrLDAPEntity Type (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP EntityType {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAP EntityType ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAP EntityType (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgr LDAPGroup Dynamic MemberAttr	The **updateIdMgr LDAPGroup Dynamic MemberAttr** command updates a dynamic member attribute configuration to an LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the LDAP attribute that is used as the group member attribute. For example, `memberURL`. (String, required) **- objectClass** The group object class that contains the dynamic member attribute. For example `groupOfURLs`. If you do not define this parameter, the dynamic member attribute will apply to all group object classes. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAPGroup DynamicMemberAttr {-id id1 -name name1 -objectClass groupOfURLs}` • Using Jython string: `AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr ('[-id id1 -name name1 -objectClass groupOfURLs]')` • Using Jython list: `AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr (['-id', 'id1', '-name', 'name1', '-objectClass', 'groupOfURLs'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAPGroup DynamicMemberAttr {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAPGroup DynamicMemberAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgr LDAPGroup MemberAttr	The **updateIdMgr LDAPGroup MemberAttr** command updates a member attribute configuration of an LDAP group configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The name of the LDAP attribute that is used as the group member attribute. For example, `member` or `uniqueMember`. (String, required) **- objectClass** The group object class that contains the member attribute. For example, `groupOfNames` or `groupOfUnqiueNames`. If you do not define this parameter, the member attribute applies to all group object classes. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP GroupMemberAttr {-id id1 -name name1}` • Using Jython string: `AdminTask.updateIdMgrLDAP GroupMemberAttr ('[-id id1 -name name1]')` • Using Jython list: `AdminTask.updateIdMgrLDAP GroupMemberAttr (['-id', 'id1', '-name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAPGroup MemberAttr {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAPGroup MemberAttr ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAPGroup MemberAttr (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- scope The scope of the member attribute. The following are the valid values: • direct - The member attribute only contains direct members whereby the member is directly contained by the group and not contained in a nested group. For example, if group1 contains group2, group2 contains user1, then group2 is a direct member of group1 but user1 is not a direct member of group1. Both member and uniqueMember are direct member attributes. • nested - The member attribute contains both direct members and nested members.	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		• all - The member attribute contains direct members, nested members and dynamic members. One example is the ibm-allMembers attribute that IBM Tivoli Directory Server supports. The default value is `direct`. (String, optional) **- dummyMember** When you create a group without specifying a member, a dummy member will be filled in automatically to avoid receiving an exception that indicates that there is a mandatory attribute missing. (String, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateId MgrLDAP Repository	The **updateId MgrLDAP Repository** command updates an LDAP repository configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- ldapServerType** The type of LDAP server that is being used. The default value is `IDS51`. (String, optional) **- adapterClassName** The default value is `com.ibm.ws.wim.adapter.ldap.LdapAdapter`. (String, optional) **- certificateMapMode** Specifies whether to map X.509 certificates into a LDAP directory by exact distinguished name or by certificate filter. The default value is `exactdn`. To use the certificate filter for the mapping, specify `certificatefilter`. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP Repository {-id id1}` • Using Jython string: `AdminTask.updateIdMgrLDAP Repository ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrLDAP Repository (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP Repository {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAP Repository ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAP Repository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- certificateFilter If `certificateMapMode` has the value `certificatefilter`, then this property specifies the LDAP filter which maps attributes in the client certificate to entries in LDAP. (String, optional) **- isExtIdUnique** Specifies if the external ID is unique. The default value is `true`. (Boolean, optional) **- loginProperties** Indicates the property name used for login. (String , optional) **- primaryServerQueryTimeInterval** Indicates the polling interval for testing the primary server availability. The value of this parameter is specified in minutes. The default value is `15`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- returnToPrimaryServer Indicates to return to the primary LDAP server when it is available. The default value is `true`. (Boolean, optional) **- supportAsyncMode** Indicates if the async mode is supported or not. The default value is `false`. (Boolean, optional) **- supportSorting** Indicates if sorting is supported or not. The default value is `false`. (Boolean, optional) **- supportPaging** Indicates if paging is supported or not. The default value is `false`. (Boolean, optional) **- supportTransaction** Indicates if transactions are supported or not. The default value is `false`. (Boolean, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- supportExternalName Indicates if external names are supported or not. The default value is `false`. (Boolean, optional) **- sslConfiguration** The SSL configuration. (String, optional) **- translateRDN** Indicates to translate RDN or not. The default value is `false`. (Boolean, optional) **- searchTimeLimit** The value of search time limit. (Integer, optional) **- searchCountLimit** The value of search count limit. (Integer, optional) **- searchPageSize** The value of search page size. (Integer, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgr LDAPSearch ResultCache	The **updateIdMgr LDAPSearch ResultCache** command updates the LDAP search result cache configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- cachesDiskOffLoad** Loads the attributes caches and the search results onto hard disk. By default, when the number of cache entries reaches the maximum size of the cache, cache entries are evicted to allow new entries to enter the caches. If you enable this parameter, the evicted cache entries will be copied to disk for future access. The default value is `false`. (Boolean, optional) **- enabled** Enables the search results cache. The default value is `true`. (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP SearchResultCache {-id id1}` • Using Jython string: `AdminTask.updateIdMgrLDAPSearch ResultCache ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrLDAPSearch ResultCache (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP SearchResultCache {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAPSearch ResultCache ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAPSearch ResultCache (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- cacheSize The maximum size of the search results cache. The number of naming enumeration objects that can be put into the search results cache. The minimum value of this parameter is `100`. The default value is `2000`. (Integer, optional) **- cacheTimeOut** The amount of time in seconds before the cached entries in the search results cache can be not valid. The minimum value for this parameter is `0`. A value of `0` means that the cached naming enumeration objects will stay in the search results cache until there are configuration changes. The default value is `600`. (Integer, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		-searchResultSizeLimit The maximum number of entries contained in the naming enumeration object that can be cached in the search results cache.For example, if the results from a search contains 2000 users, the search results will not cache in the search results cache if the value of the of this property is set to `1000`. The default value is `1000`. (Integer, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgr LDAPServer	The **updateIdMgr LDAPServer**command updates an LDAP server configuration for the LDAP repository ID that you specify.	• Parameters: **- id** The ID of the repository. (String, required) **- host** The host name for the LDAP server that contains the properties that you want to modify. (String, required) **- port** The port number for the LDAP server. (Integer, optional) **- authentication** Indicates the authentication method to use. The default value is `simple`. Valid values include: `none` or `strong`. (String, optional) **- bindDN** The binding domain name for the LDAP server. (String, optional) **- bindPassword** The binding password. The password is encrypted before it is stored.(String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAPServer {-id id1 -host myhost.ibm.com}` • Using Jython string: `AdminTask.updateIdMgrLDAPServer ('[-id id1 -host myhost.ibm.com]')` • Using Jython list: `AdminTask.updateIdMgrLDAPServer (['-id', 'id1', '-host', 'myhost.ibm.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrLDAP Server {-interactive}` • Using Jython string: `AdminTask.updateIdMgrLDAP Server ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrLDAP Server (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- certificateMapMode Specifies whether to map X.509 certificates into a LDAP directory by exact distinguished name or by certificate filter. The default value is `exactdn`. To use the certificate filter for the mapping, specify `certificatefilter`. (String, optional) **- certificateFilter** If `certificateMapMode` has the value `certificatefilter`, then this property specifies the LDAP filter which maps attributes in the client certificate to entries in LDAP. (String, optional) **- connectTimeout** The connection timeout measured in seconds. The default value is `0`. (Integer, optional) **- connectionPool** The connection pool. The default value is `false`. (Boolean, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- derefAliases Controls how aliases are dereferenced. The default value is `always`. Valid values include: • `never` - never deference aliases • `finding` - deferences aliases only during name resolution • `searching` - deferences aliases only after name resolution (String, optional) **- ldapServerType** The type of LDAP server being used. The default value is `IDS51`. (String, optional) **- primary_host** The host name for the primary LDAP server. (String, optional) **- referal** The LDAP referral. The default value is `ignore`. Valid values include: `follow`, `throw`, or `false`. (String, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- sslConfiguration The SSL configuration. (String, optional) **- sslEnabled** Indicates to enable SSL or not. The default value is `false`. (Boolean, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgr Repository	The **updateIdMgr Repository**command updates the common repository configuration.	• Parameters: **- id** The ID of the repository. (String, required) **- adapterClassName** The implementation class name for the repository adapter. (String, optional) **- EntityTypesNotAllowCreate** The name of the entity type that should not be created in this repository. (String, optional) **- EntityTypesNotAllowUpdate** The name of the entity type that should not be updated in this repository. (String, optional) **- EntityTypesNotAllowRead** The name of the entity type that should not be read from this repository. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrRepository {-id id1}` • Using Jython string: `AdminTask.updateIdMgrRepository ('[-id id1]')` • Using Jython list: `AdminTask.updateIdMgrRepository (['-id', 'id1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrRepository {-interactive}` • Using Jython string: `AdminTask.updateIdMgrRepository ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrRepository (['-interactive'])`

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- EntityTypesNotAllowDelete The name of the entity type that should not be deleted from this repository. (String, optional) **- loginProperties** (String, optional) **- readOnly** Indicates if this is a read only repository. The default value is `false`. (Boolean, optional) **- repositoriesForGroups** The repository ID where group data is stored. (String, optional) **- supportPaging** Indicates if the repository supports paging or not. (Boolean, optional) **- supportSorting** Indicates if the repository supports sorting or not. (Boolean, optional)	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- supportTransaction Indicates if the repository supports transaction or not. (Boolean, optional) **- isExtIdUnique** Specifies if the external ID is unique or not. (Boolean, optional) **- supportedExternalName** Indicates if the repository supports external names or not. (Boolean, optional) **- supportAsyncMode** Indicates if the adapter supports async mode or not. The default value is `false`. (Boolean, optional) • Returns: None	

Table 4. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgr Repository BaseEntry	The **updateIdMgr Repository BaseEntry** command updates a base entry to the specified repository.	• Parameters: **- id** The ID of the repository. (String, required) **- name** The distinguished name of a base entry. (String, required) **- nameInRepository** The distinguished name in the repository that uniquely identifies the base entry name. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrReposi toryBaseEntry {-id id1 name name1}` • Using Jython string: `AdminTask.updateIdMgrRepository BaseEntry ('[-id id1 name name1]')` • Using Jython list: `AdminTask.updateIdMgrReposi toryBaseEntry (['-id', 'id1', 'name', 'name1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrReposi toryBaseEntry {-interactive}` • Using Jython string: `AdminTask.updateIdMgrReposi toryBaseEntry ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrReposi toryBaseEntry (['-interactive'])`

Commands for the IdMgrRealmConfig group of the AdminTask object

Use the commands in the IdMgrConfig group to configure the member manager realm and realms. The commands for this group do not require a target object. To see the additional commands related to the member manager, see the `Commands for the IdMgrRepositoryConfig group of the AdminTask object and the Commands for the IdMgrConfig group of the AdminTask object articles.

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the IdMgrRealmConfig group of the AdminTask object:

Table 5.

Command name:	Description:	Parameters and return values:	Examples:
addIdMgrRealmBaseEntry	The **addIdMgrRealmBaseEntry** command adds a base entry to a specified realm configuration.	• Parameters: **- name** The realm name. (String, required) **- baseEntry** The name of a base entry. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask addIdMgrRealmBaseEntry {-name realm1 -baseEntry entry1}` • Using Jython string: `AdminTask addIdMgrRealmBaseEntry ('[-name realm1 -baseEntry entry1]')` • Using Jython list: `AdminTask addIdMgrRealmBaseEntry (['-name', 'realm1', '-base Entry', 'entry1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addIdMgrRealm BaseEntry {-interactive}` • Using Jython string: `AdminTask.addIdMgrRealmBase Entry ('[-interactive]')` • Using Jython list: `AdminTask.addIdMgrRealmBase Entry (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createIdMgrRealm	The **createIdMgrRealm** command creates a realm configuration.	• Parameters: **- name** The realm name. (String, required) **- securityUse** A string that indicates if this virtual realm will be used in security now, later, or never. The default value is `active`. Additional values includes: `inactive` and `nonSelectable`. (String, optional) **- delimiter** The delimiter used for this realm. The default value is `@`. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createIdMgrRealm {-name realm1}` • Using Jython string: `AdminTask.createIdMgrRealm ('[-name realm1]')` • Using Jython list: `AdminTask.createIdMgrRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createIdMgrRealm {-interactive}` • Using Jython string: `AdminTask.createIdMgrRealm ('[-interactive]')` • Using Jython list: `AdminTask.createIdMgrRealm (['-interactive'])`
deleteIdMgrRealm	The **deleteIdMgrRealm** command deletes the realm configuration that you specified.	• Parameters: **- name** The realm name. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRealm {-name realm1}` • Using Jython string: `AdminTask.deleteIdMgrRealm ('[-name realm1]')` • Using Jython list: `AdminTask.deleteIdMgrRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRealm {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrRealm ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrRealm (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteIdMgr RealmBaseEntry	The **deleteIdMgr RealmBaseEntry** command deletes a base entry from a realm configuration that you specified. The realm must always contain at least one base entry, thus you cannot remove every entry.	• Parameters: **- name** The realm name. (String, required) **- baseEntry** The name of a base entry. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRealmBaseEntry {-name realm1 -baseEntry entry1}` • Using Jython string: `AdminTask.deleteIdMgrRealmBaseEntry ('[-name realm1 -baseEntry entry1]')` • Using Jython list: `AdminTask.deleteIdMgrRealmBaseEntry (['-name', 'realm1', '-base Entry', 'entry1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteIdMgrRealm BaseEntry {-interactive}` • Using Jython string: `AdminTask.deleteIdMgrRealm BaseEntry ('[-interactive]')` • Using Jython list: `AdminTask.deleteIdMgrRealm BaseEntry (['-interactive'])`
getIdMgrDefaultRealm	The **getIdMgrDefaultRealm** command returns the default realm name.	• Parameters: None • Returns: The name of the default realm.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrDefaultRealm` • Using Jython string: `AdminTask.getIdMgrDefaultRealm()` • Using Jython list: `AdminTask.getIdMgrDefaultRealm()` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrDefault Realm {-interactive}` • Using Jython string: `AdminTask.getIdMgrDefault Realm ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrDefault Realm (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgrRepositories ForRealm	The **getIdMgrRepositories ForRealm** command returns repository specific details for the repositories configured for a specified realm.	• Parameters: **- name** The realm name. (String, required) • Returns: A hash map with the following keys: – id - The repository ID. – repositoryType - The type of repository, for example, File, LDAP, DB, and so on. – specificRepositoryType - The specific type of repository, for example, for LDAP, IDS51, NDS, and so on. – host - The host name where the repository resides. For example, for File, LocalHost or for DB, dataSourceName. – port - The port number. This only applies to LDAP. – name - The name of the base entry. – nameInRepository - The name in repository for the base entry. This command will not return the property extension and entry mapping repository data.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrRepositories ForRealm {-name realm1}` • Using Jython string: `AdminTask.getIdMgrRepositories ForRealm ('[-name realm1]')` • Using Jython list: `AdminTask.getIdMgrRepositories ForRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrRepositories ForRealm {-interactive}` • Using Jython string: `AdminTask.getIdMgrRepositories ForRealm ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrRepositories ForRealm (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getIdMgrRealm	The **getIdMgrRealm** command returns the configuration parameters for the realm that you specified.	• Parameters: **- name** The realm name. (String, required) • Returns: A hash map that contains keys as the parameters of the **createIdMgrRealm** command.	**Batch mode example usage:** • Using Jacl: `$AdminTask getIdMgrRealm {-name realm1}` • Using Jython string: `AdminTask.getIdMgrRealm ('[-name realm1]')` • Using Jython list: `AdminTask.getIdMgrRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getIdMgrRealm {-interactive}` • Using Jython string: `AdminTask.getIdMgrRealm ('[-interactive]')` • Using Jython list: `AdminTask.getIdMgrRealm (['-interactive'])`
listIdMgrRealms	The **listIdMgrRealms** command returns all of the names of the configured realms.	• Parameters: None • Returns: A list that contains the name of the configured realms.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrRealms` • Using Jython string: `AdminTask.listIdMgrRealms()` • Using Jython list: `AdminTask.listIdMgrRealms()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrRealms {-interactive}` • Using Jython string: `AdminTask.listIdMgrRealms ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrRealms (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listIdMgrRealm BaseEntries	The **listIdMgrRealm BaseEntries** command returns all of the names of the configured realms.	• Parameters: **- name** The realm name. (String, required) • Returns: A list of all the base entries of the specified realm.	**Batch mode example usage:** • Using Jacl: `$AdminTask listIdMgrRealmBase Entries {-name realm1}` • Using Jython string: `AdminTask.listIdMgrRealmBase Entries ('[-name realm1]')` • Using Jython list: `AdminTask.listIdMgrRealmBase Entries (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listIdMgrRealmBase Entries {-interactive}` • Using Jython string: `AdminTask.listIdMgrRealmBase Entries ('[-interactive]')` • Using Jython list: `AdminTask.listIdMgrRealmBase Entries (['-interactive'])`
renameIdMgrRealm	The **renameIdMgrRealm** command renames the name of the realm that you specified.	• Parameters: **- name** The realm name. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask renameIdMgrRealm {-name realm1}` • Using Jython string: `AdminTask.renameIdMgrRealm ('[-name realm1]')` • Using Jython list: `AdminTask.renameIdMgrRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask renameIdMgrRealm {-interactive}` • Using Jython string: `AdminTask.renameIdMgrRealm ('[-interactive]')` • Using Jython list: `AdminTask.renameIdMgrRealm (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
setIdMgrDefaultRealm	The **setIdMgrDefaultRealm** command sets up the default realm configuration.	• Parameters: **- name** The name of the realm that is used as a default realm when the caller does not specify any in context. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setIdMgrDefaultRealm {-name realm1}` • Using Jython string: `AdminTask.setIdMgrDefaultRealm ('[-name realm1]')` • Using Jython list: `AdminTask.setIdMgrDefaultRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setIdMgrDefault Realm {-interactive}` • Using Jython string: `AdminTask.setIdMgrDefault Realm ('[-interactive]')` • Using Jython list: `AdminTask.setIdMgrDefault Realm (['-interactive'])`

Table 5. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateIdMgrRealm	The **updateIdMgrRealm** command updates the configuration for a realm that you specify.	• Parameters: **- name** The realm name. (String, required) **- securityUse** A string that indicates if this realm will be used in security now, later, or never. The default value is `active`. Additional values includes: `inactive` and `nonSelectable`. (String, optional) **- delimiter** The delimiter used for this realm. The default value is `@`. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask updateIdMgrRealm {-name realm1}` • Using Jython string: `AdminTask.updateIdMgrRealm ('[-name realm1]')` • Using Jython list: `AdminTask.updateIdMgrRealm (['-name', 'realm1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateIdMgrRealm {-interactive}` • Using Jython string: `AdminTask.updateIdMgrRealm ('[-interactive]')` • Using Jython list: `AdminTask.updateIdMgrRealm (['-interactive'])`

Commands for the WIMManagementCommands group of the AdminTask object

Use the commands in the WIMManagementCommands group to configure the virtual member manager. The commands for this group do not require a target object. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the WIMManagementCommands group of the AdminTask object:

Table 6.

Command name:	Description:	Parameters and return values:	Examples:
addMemberToGroup	The **addMemberToGroup** command adds a user or a group to a group.	• Parameters: **- memberUniqueName** Specifies the unique name value for the user or group that you want to add to the specified group. This parameter maps to the uniqueName property in the virtual member manager. (String, required) **- groupUniqueName** Specifies the unique name value for the group to which you want to add the user or group that you specified in the memberUniqueName parameter. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: Void if the command is successful. Returns an exception if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask addMemberToGroup {-memberUniqueName uid=meyersd,cn=users,dc=yourco,dc=com -groupUniqueName cn=admins,cn=groups,dc=yourco,dc=com}` • Using Jython string: `AdminTask.addMemberToGroup ('[-memberUniqueName uid=meyersd,cn=users,dc=yourco,dc=com -group UniqueName cn=admins,cn=groups,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.addMemberToGroup (['-memberUniqueName', 'uid=meyersd,cn=users,dc=yourco,dc=com', '-group UniqueName', 'cn=admins,cn=groups,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addMemberToGroup {-interactive}` • Using Jython string: `AdminTask.addMemberToGroup ('[-interactive]')` • Using Jython list: `AdminTask.addMemberToGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createGroup	The **createGroup** command creates a new group in the virtual member manager. After the command completes, the new group will appear in the repository. For LDAP, a group must contain a member. The memberUniqueName parameter is optional in this case. If you set the memberUniqueName parameter to the unique name of a group or a user, the group or user will be added as a member of the group.	• Parameters: **- cn** Specifies the common name for the group that you want to create. This parameter maps to the cn property in virtual member manager. (String, required) **- description** Specifies additional information about the group that you want to create. This parameter maps to the description property in a virtual member manager object. (String, optional) **- parent** Specifies the repository in which you want to create the group. This parameter maps to the parent property in the virtual member manager. (String, optional) **- memberUniqueName** Specifies the unique name value for the user or group that you want to add to the new group. This parameter maps to the uniqueName property in the virtual member manager. (String, optional) • Returns: The unique name of the group that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createGroup {-cn groupA -description a group of admins}` • Using Jython string: `AdminTask.createGroup ('[-cn groupA -description a group of admins]')` • Using Jython list: `AdminTask.createGroup (['-cn', 'groupA', '-description', 'a group of admins'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createGroup {-interactive}` • Using Jython string: `AdminTask.createGroup ('[-interactive]')` • Using Jython list: `AdminTask.createGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createUser	The **createUser** command creates a new user in the default repository or a repository that the parent command parameter specifies. This command creates a person entity and a login account entity in the virtual member manager.	• Parameters: **- uid** Specifies the unique ID for the user that you want to create. Virtual member manager then creates a uniqueId value and a uniqueName value for the user. This parameter maps to the uid property in the virutal member manager. (String, required) **- password** Specifies the password for the user. This parameter maps to the password property in the virtual member manager. (String, required) **- confirmPassword** Specifies the password again to validate how it was entered for the password parameter. This parameter maps to the password property in virtual member manager. (String, optional) **- cn** Specifes the first name or given name of the user. This parameter maps to the cn property in virutal member manager. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createUser {-uid 123 -password tempPass -confirmPassword tempPass -cn Jane -surname Doe -ibm-primaryEmail janedoe@acme.com}` • Using Jython string: `AdminTask.createUser ('[-uid 123 -password tempPass -confirmPassword tempPass -cn Jane -surname Doe -ibm-primaryEmail janedoe@acme.com]')` • Using Jython list: `AdminTask.createUser (['-uid', '123', '-password', 'tempPass', '-confirmPassword', 'tempPass', '-cn', 'Jane', '-surname', 'Doe', '-ibm-primaryEmail', 'janedoe@acme.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createUser {-interactive}` • Using Jython string: `AdminTask.createUser ('[-interactive]')` • Using Jython list: `AdminTask.createUser (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- surname Specifies the last name or family name of the user. This parameter maps to the sn property in virtual member manager. (String, optional) **- ibm-primaryEmail** Specifies the e-mail address of the user. This parameter maps to the ibm-PrimaryEmail property in the virtual member manager. (String, optional) **- parent** Specifies the repository in which you want to create the user. This parameter maps to the parent property in the virtual member manager. (String, optional) • Returns: The unique name of the user that you created.	

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteGroup	The **deleteGroup** command deletes a group in the virtual member manager. You cannot use this command to delete descendants. When this command completes, the group will be deleted from the repository.	• Parameters: **- uniqueName** Specifies the unique name value for the group that you want to delete. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: Void if the command is successful. Returns an error if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteGroup {-uniqueName cn=operators,cn=users,dc=yourco,dc=com}` • Using Jython string: `AdminTask.deleteGroup ('[-uniqueName cn=operators,cn=users,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.deleteGroup (['-uniqueName', 'cn=operators,cn=users,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteGroup {-interactive}` • Using Jython string: `AdminTask.deleteGroup ('[-interactive]')` • Using Jython list: `AdminTask.deleteGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
deleteUser	The **deleteUser** command deletes a user from the virtual member manager. This includes a person object and an account object in the non-merged repositories.	• Parameters: **- uniqueName** Specifies the unique name value for the user that you want to delete. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: Void if the command is successful and an exception if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteUser {-uniqueName uid=dmeyers,cn=users,dc=yourco,dc=com}` • Using Jython string: `AdminTask.deleteUser ('[-uniqueName uid=dmeyers,cn=users,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.deleteUser (['-uniqueName', 'uid=dmeyers,cn=users,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteUser {-interactive}` • Using Jython string: `AdminTask.deleteUser ('[-interactive]')` • Using Jython list: `AdminTask.deleteUser (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
duplicate Membership OfGroup	Use the **duplicate Membership OfGroup** command to make a one group a member of all of the same groups as another group. For example, group A is in group B and group C. To add group D to the same groups as group A, use the **duplicate Membership OfGroup** command.	• Parameters: **- copyToName** Specifies the name of the group to which you want to add the memberships of the group specified in the copyFromName parameter. (String, required) **- copyFromName** Specifies the name of the group from which you want to copy the group memberships for another group to use. (String, required) • Returns: Void if the command is successful. Returns an exception if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask duplicateMembership OfGroup {-copyToName cn=operators,cn=groups,dc=yourco,dc=com -copyFromName cn=admins,cn=groups,dc=yourco,dc=com}` • Using Jython string: `AdminTask.duplicateMembershipOfGroup ('[-copyToName cn=operators,cn=groups,dc=yourco,dc=com -copyFromName cn=admins,cn=groups,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.duplicateMembershipOfGroup (['-copyToName', 'cn=operators,cn=groups,dc=yourco,dc=com', '-copyFromName', 'cn=admins,cn=groups,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask duplicateMembershipOfGroup {-interactive}` • Using Jython string: `AdminTask.duplicateMembershipOfGroup ('[-interactive]')` • Using Jython list: `AdminTask.duplicateMembershipOfGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
duplicate Membership OfUser	Use the **duplicate Membership OfUser** command to make a one user a member of all of the same groups as another user. For example, user 1 is in group B and group C. To add user 2 to the same groups as user 1, use the **duplicate Membership OfUser** command.	• Parameters: **- copyToName** Specifies the name of the user to which you want to add the memberships of the user specified in the copyFromName parameter. (String, required) **- copyFromName** Specifies the name of the user from which you want to copy the group memberships for another user to use. (String, required) • Returns: Void if the command is successful. Returns an exception if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask duplicateMembership OfUser {-copyToName uid=meyersd,cn=users,dc=yourco,dc=com -copyFromName uid=jhart,cn=users,dc=yourco,dc=com}` • Using Jython string: `AdminTask.duplicateMembership OfUser ('[-copyToName uid=meyersd,cn=users,dc=yourco,dc=com -copyFromName uid=jhart,cn=users,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.duplicateMembership OfUser (['-copyToName', 'uid=meyersd,cn=users,dc=yourco,dc=com', '-copyFromName', 'uid=jhart,cn=users,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask duplicateMembership OfUser {-interactive}` • Using Jython string: `AdminTask.duplicateMembership OfUser ('[-interactive]')` • Using Jython list: `AdminTask.duplicateMembership OfUser (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getGroup	The **getGroup** command retrieves the common name and description of a group.	• Parameters: **- uniqueName** Specifies the unique name value for the group that you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: A map of common name and description properties.	**Batch mode example usage:** • Using Jacl: `$AdminTask getGroup {-uniqueName cn=operators, cn=groups,dc=yourco,dc=com}` • Using Jython string: `AdminTask.getGroup ('[ -uniqueName cn=operators, cn=groups,dc=yourco, dc=com]')` • Using Jython list: `AdminTask.getGroup (['-uniqueName', 'cn=operators,cn=groups,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getGroup {-interactive}` • Using Jython string: `AdminTask.getGroup ('[-interactive]')` • Using Jython list: `AdminTask.getGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getMembership OfGroup	The **getMembership OfGroup** command retrieves the groups of which a group is a member.	• Parameters: **- uniqueName** Specifies the unique name value for the group whose group memberships you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: A list of unique names of each of the groups of which the group is a member.	**Batch mode example usage:** • Using Jacl: `$AdminTask getMebmership OfGroup {-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com}` • Using Jython string: `AdminTask.getMebmership OfGroup ('[-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com]')` • Using Jython list: `AdminTask.getMebmership OfGroup (['-uniqueName', 'uid=dmeyers,cn=users, dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getMembership OfGroup {-interactive}` • Using Jython string: `AdminTask.getMembership OfGroup ('[-interactive]')` • Using Jython list: `AdminTask.getMembership OfGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getMem bership OfUser	The **getMem bership OfUser** command retrieves the groups of which a user is a member.	• Parameters: **- uniqueName** Specifies the unique name value for the user whose group memberships you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: A list of unique names for each group of which the user is a member.	**Batch mode example usage:** • Using Jacl: `$AdminTask getMebmer shipOfUser {-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com}` • Using Jython string: `AdminTask.getMebmership OfUser ('[-uniqueName uid=dmeyers,cn=users, dc=yourco,dc=com]')` • Using Jython list: `AdminTask.getMebmership OfUser (['-uniqueName', 'uid=dmeyers,cn=users ,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getMembership OfUser {-interactive}` • Using Jython string: `AdminTask.getMembership OfUser ('[-interactive]')` • Using Jython list: `AdminTask.getMembership OfUser (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getMembers OfGroup	The **getMembers OfGroup** command retrieves the members of a group.	• Parameters: **- uniqueName** Specifies the unique name value for the group whose members you want to view. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: The unique name of each of the members of the group and the type of each member.	**Batch mode example usage:** • Using Jacl: `$AdminTask getMembersOf Group {-uniqueName cn=operators,cn=groups ,dc=yourco,dc=com}` • Using Jython string: `AdminTask.getMembersOf Group ['(-uniqueName cn=operators,cn=groups ,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.getMembersOf Group [('-uniqueName', 'cn=operators,cn=groups ,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getMembersOfGroup {-interactive}` • Using Jython string: `AdminTask.getMembersOfGroup ('[-interactive]')` • Using Jython list: `AdminTask.getMembersOfGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getUser	The **getUser** command retrieves information about a user in the virtual member manager.	• Parameters: **- uniqueName** Specifies the unique name value for the user that you want to view. This parameter maps to the uniqueName property in the virtual member manager. (String, required) • Returns: A map that contains the following properties: uniqueName, cn, sn, uid, and ibm-primaryEmail. These attributes are fixed and you cannot change them.	**Batch mode example usage:** • Using Jacl: `$AdminTask getUser {-user Name uid=dmeyers,cn=us ers,dc=yourco,dc=com}` • Using Jython string: `AdminTask.getUser ('[-use rName uid=dmeyers,cn= users,dc=yourco,dc=com]')` • Using Jython list: `AdminTask.getUser (['-use rName', 'uid=dmeyers,c n=users,dc=yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getUser {-interactive}` • Using Jython string: `AdminTask.getUser ('[-interactive]')` • Using Jython list: `AdminTask.getUser (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
removeMember FromGroup	The **removeMember FromGroup** command removes a user or a group from a group.	• Parameters: **- memberUniqueName** Specifies the unique name value for the user or group that you want to remove from the specified group. This parameter maps to the uniqueName property in virtual member manager. (String, required) **- groupUniqueName** Specifies the unique name value for the group from which you want to remove the user or group that you specified with the memberUniqueName paramter. This parameter maps to the uniqueName property in virtual member manager. (String, required) • Returns: Void if the command is successful. Returns an exception if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeMember FromGroup {-memberUnique Name uid=meyersd,cn= users,dc=yourco,dc=com -groupUniqueName cn= admins,cn-groups,dc= yourco,dc=com}` • Using Jython string: `AdminTask.removeMemberF romGroup ('[-memberUnique Name uid=meyersd,cn= users,dc=yourco,dc=com -groupUniqueName cn=a dmins,cn-groups,dc=your co,dc=com]')` • Using Jython list: `AdminTask.removeMemberFrom Group (['-memberUniqueName', 'uid=meyersd,cn=users, dc=yourco,dc=com', '-groupUniqueName', 'cn=admins,cn-groups,dc= yourco,dc=com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeMember FromGroup {-interactive}` • Using Jython string: `AdminTask.removeMemberFr omGroup ('[-interactive]')` • Using Jython list: `AdminTask.removeMemberFr omGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
searchGroups	Use the **searchGroups** command to find groups in the virtual member manager that match criteria that you provide. For example, you can use the **searchGroups** command to find all of the groups with a common name that begins with `IBM`. You can search for any virtual member manager property because the command is generic.	• Parameters: **- cn** The first name or given name of the user. This parameter maps to the cn property in the virtual member manager. You must set this parameter or the description parameter, but not both. (String, optional) **- description** Specifies information about the group. This parameter maps to the description entity in a virtual member manager object. You must set this parameter or the cn parameter, but not both. (String, optional) **- timeLimit** Specifies the maximum amount of time in milliseconds that the search can run. The default value is no time limit. (String, optional) **- countLimit** Specifies the maximum number of results that you want returned from the search. By default, all groups found in the search are returned. (String, optional) • Returns: A list of unique names of all of the groups that match the search criteria that you provided.	**Batch mode example usage:** • Using Jacl: `$AdminTask searchGroups {cn *IBM*}` • Using Jython string: `AdminTask.searchGroups ('[cn *IBM*]')` • Using Jython list: `AdminTask.searchGroups (['cn', '*IBM*'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask searchGroups {-interactive}` • Using Jython string: `AdminTask.searchGroups ('[-interactive]')` • Using Jython list: `AdminTask.searchGroups (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
searchUsers	Use the **searchUsers** command to find users in the virtual member manager that match criteria that you provide. For example, you can use the **searchUsers** command to find all of the telephone numbers that contain 919. You can search for any virtual member manager property because the command is generic.	• Parameters: **- principalName** Specifies the principal name oft he user that is used as the logon ID for the user in the system. This parameter maps to the principalName property in virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional) **- uid** Specifies the unique ID value for the user for whom you want to search. This parameter maps to the uid property in virtual member manage. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional) **- cn** Specifies the first name or given name of the user. This parameter maps to the cn property in virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask searchUsers {-principalName */IBM/US*}` • Using Jython string: `AdminTask.searchUsers ('[-principalName */IBM/US*]')` • Using Jython list: `AdminTask.searchUsers (['-principalName', '*/IBM/US*'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask searchUsers {-interactive}` • Using Jython string: `AdminTask.searchUsers ('[-interactive]')` • Using Jython list: `AdminTask.searchUsers (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- sn Specifies the last name or family name of the user. This parameter maps to the sn property in virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional) **- ibm-primaryEmail** Specifies the email address of the user. This parameter maps to the ibm-PrimaryEmail property in the virtual member manager. You must specify only one of the following parameters: principalName, uid, cn, sn, or ibm-primaryEmail. (String, optional) **- timeLimit** Specifies the maximum amount of time in milliseconds that the search can run. The default is not time limit. (String, optional) **- countLimit** Specifies the maximum number of results that you want returned from the search. By default, all users found int he search are returned. (String, optional) • Returns: A list of unique names of all of the users that match the search criteria that you provided.	

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateGroup	The **updateGroup** command updates the common name or the description of a group.	• Parameters: **- uniqueName** Specifies the unique name value for the group for which you want to modify the properties. This parameter maps to the uniqueName property in virtual member manager. (String, required) **- cn** Specifies the new common name used for the group. This parameter maps to the cn property in virtual member manager. (String, optional) **- description** Specifies the new information about the group. This parameter maps to the description entity in a virtual member manager object. (String, optional) • Returns: Void if the command is successful. Returns an exception if the command fails.	**Batch mode example usage:** • Using Jacl: `$AdminTask updateGroup {-uniqueName cn=operators,cn=groups,dc=yourco,dc=com -cn groupA}` • Using Jython string: `AdminTask.updateGroup ('[-uniqueName cn=operators,cn=groups,dc=yourco,dc=com -cn groupA]')` • Using Jython list: `AdminTask.updateGroup (['-uniqueName', 'cn=operators,cn=groups,dc=yourco,dc=com', '-cn', 'groupA'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateGroup {-interactive}` • Using Jython string: `AdminTask.updateGroup ('[-interactive]')` • Using Jython list: `AdminTask.updateGroup (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
updateUser	The **updateUser** command updates the following properties: uniqueName, uid, password, cn, sn, or ibm-primaryEmail.	• Parameters: **- uniqueName** Specifies the unique name value for the user for which you want to modify the properties. This parameter maps to the uniqueName property in virtual member manager. (String, required) **- uid** Specifies the new unique ID value for the user. This parameter maps to the uid property in virtual member manager. (String, optional) **- password** Specifies the new password for the user. This parameter maps to the password property in virtual member manager. (String, optional) **- confirmPassword** Specifies the password again to validate how it was entered on the password parameter. This parameter maps to the password property in virtual member manager. (String, optional) **- cn** Specifies the new first name or given name of the user. This parameter maps to the cn property in virtual member manager. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask updateUser {-uniqueName uid=dmeyers,cn=users,dc=yourco,dc=com -uid 123}` • Using Jython string: `AdminTask.updateUser ('[-uniqueName uid=dmeyers,cn=users,dc=yourco,dc=com -uid 123]')` • Using Jython list: `AdminTask.updateUser (['-uniqueName', 'uid=dmeyers,cn=users,dc=yourco,dc=com', '-uid', '123'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask updateUser {-interactive}` • Using Jython string: `AdminTask.updateUser ('[-interactive]')` • Using Jython list: `AdminTask.updateUser (['-interactive'])`

Table 6. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- surname Specifies the new last name or family name of the user. This parameter maps to the sn property in virtual member manager. (String, optional) **- ibm-primaryEmail** Specifies the new e-mail address of the user. This parameter maps to the mail property in virtual member manager. (String, optional) • Returns: Void if the command is successful. Returns an exception if the command fails.	

Commands for the KeyStoreCommands group of the AdminTask object

Use the commands in the KeyStoreCommands group to create or delete key stores. The commands for this group do not require a target object. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the KeyStoreCommands group of the AdminTask object:

Table 7.

Command name:	Description:	Parameters and return values:	Examples:
changeKey Store Password	The **changeKey Store Password** command changes the password on the key store.	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- keyStorePassword** The password that protects the key store that you want to change. (String, required) **- newkeyStorePassword** The new password that protects the key store. (String, required) **- newkeyStorePassword Verify** The new password that protects the key store. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask changeKeyStore Password {-keyStoreName testKS –keyStorePassword testpwd –newKeyStorePassword keyPWD –newKeyStorePasswordVerify keyPWD}` • Using Jython string: `AdminTask.changeKeyStore Password ('[-keyStoreName testKS –keyStorePassword testpwd –newKeyStorePassword keyPWD –newKeyStorePassword Verify keyPWD]')` • Using Jython list: `AdminTask.changeKeyStore Password (['-keyStoreName', 'testKS', '–keyStorePassword', 'testpwd', '–newKeyStore Password', 'keyPWD', '–new KeyStorePasswordVerify', 'keyPWD'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask changeKeyStore Password {-interactive}` • Using Jython string: `AdminTask.changeKeyStore Password ('[-interactive]')` • Using Jython list: `AdminTask.changeKeyStore Password (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
change Multiple KeyStore Passwords	The **change Multiple KeyStore Passwords** command updates all of the key stores in the configuration that have a give password and changed them to a new password. This is useful because when you create key store files on the system, they will have `WebAS` as a password by default.	• Parameters: **- keyStorePassword** Specifies the name of the password that you want to change. (String, required) **- newKeyStorePassword** Specifies the new password that you will use to access the key store. (String, required) **- newKeyStorePasswordVerify** Confirms the new key store password. (String, required) • Returns: A list of key store aliases that where changed	**Batch mode example usage:** • Using Jacl: `$AdminTask changeMultiple KeyStorePasswords {-key StorePassword WebAS -new KeyStorePassword newpwd -newKeyStorePassword Verify newpwd}` • Using Jython string: `AdminTask.changeMultiple KeyStorePasswords ('[-key StorePassword WebAS -new KeyStorePassword newpwd -newKeyStorePasswordVerify newpwd]')` • Using Jython list: `AdminTask.changeMultiple KeyStorePasswords (['-key StorePassword', 'WebAS', '-newKeyStorePassword', 'newpwd', '-newKeyStore PasswordVerify', 'newpwd'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask changeMul tipleKeyStorePasswords {-interactive}` • Using Jython string: `AdminTask.changeMulti pleKeyStorePasswords ('[-interactive]')` • Using Jython list: `AdminTask.changeMulti pleKeyStorePasswords (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createKeyStore	The **createKeyStore** command creates the key store settings in the configuration and the key store database.	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreType** The implementation of the key store management. (String, required) **- keyStoreLocation** The location of the key store. For file based, the location is the files system path to the key store database. For hardware key store, the location is the path to the token library. (String, required) **- keyStorePassword** The password that protects the key store. (String, required) **- keyStorePasswordVerify** The password that protects the key store. (String, required) **- keyStoreProvider** The provider used to implement the key store. (String, optional) **- isKeyStoreFileBased** Set the value of this parameter to `true` if the key store is file based. Set the value of this parameter to `false` for hardware crypto key stores. (Boolean, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createKeyStore {-keyStoreName testKS –location c:\temp\test KeyFile.p12 keyStorePass word testpwd –keyStore PasswordVerify testpwd –isKeyStoreFileBased true –keyStoreInitAtStartup true –keyStoreReadOnly false}` • Using Jython string: `AdminTask.createKeyStore ('[-keyStoreName testKS –location c:\temp\testKey File.p12 keyStorePassword testpwd –keyStorePass wordVerify testpwd –isKey StoreFileBased true –key StoreInitAtStartup true –keyStoreReadOnly false]')` • Using Jython list: `AdminTask.createKeyStore (['-keyStoreName', 'testKS', '–location', 'c:\temp\test KeyFile.p12', 'keyStorePass word', 'testpwd', '–keySto rePasswordVerify', 'testpwd', '–isKeyStoreFileBased', 'true', '–keyStoreInitAt Startup', 'true', '–key StoreReadOnly', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createKey Store {-interactive}` • Using Jython string: `AdminTask.createKeySt ore ('[-interactive]')` • Using Jython list: `AdminTask.createKeyS tore (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
		- keyStoreHostList A list of host names that indicate from where the key store is remotely managed, separated by commas. (String, optional) **- keyStoreInitAtStartup** Set the value of this parameter to `true` if the key store is initialized at startup. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- keyStoreReadOnly** Set the value of this parameter to `true` if you cannot write to the key store. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- keyStoreStashFile** Set the value of this parameter to `true` if you want to create stash files for CMS type key store. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- scopeName** The name of the scope. (String, optional) **- enableCryptoOperations** Specifies if the key store object will be used for hardware cryptographic operations or not. The default value is `false`. (Boolean, optional) • Returns: The configuration object name of the key store object that you created.	

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
createCMSKeyStore	The **createCMSKeyStore** command creates a CMS key store database and the key store settings in the configuration.	• Parameters: **- cmsKeyStoreURI** The URI of the CMS key store. (String, required) **- pluginHostName** The host name of the plug-in. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createCMSKeyStore` • Using Jython: `AdminTask.createCMSKeyStore()` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCMSKeyStore {-interactive}` • Using Jython string: `AdminTask.createCMSKeyStore ('[-interactive]')` • Using Jython list: `AdminTask.createCMSKeyStore (['-interactive'])`
deleteKeyStore	The **deleteKeyStore** command deletes the settings of a key store from the configuration and the key store file.	• Parameters: **- name** The name that uniquely identifies the key store that you want to delete. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteKeyStore {-name testKS}` • Using Jython string: `AdminTask.deleteKeyStore ('[-name testKS]')` • Using Jython list: `AdminTask.deleteKeyStore (['-name', 'testKS'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteKeyStore {-interactive}` • Using Jython string: `AdminTask.deleteKeyStore ('[-interactive]')` • Using Jython list: `AdminTask.deleteKeyStore (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
exchangeSigners	The **exchangeSigners** command exchange signer certificate between key stores.	• Parameters: **- keyStoreName1** The name that uniquely identifies a key store. You must specify a second key store name using the keyStoreName2 parameter. (String, required) **- keyStoreScope1** The scope name of the key store that you specified with the keyStoreName1 parameter. (String, required) **- certificateAlaisList1** A list of aliases separated by a comma. (String, optional) **- keyStoreName2** The name that uniquely identifies a key store. You must specify a second key store name using the keyStoreName1 parameter. (String, required) **- keyStoreScope2** The scope name of the key store that you specified with the keyStoreName2 parameter. (String, required) **- certificateAliasList2** A list of aliases separated by a comma. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask exchangeSigners {-keyStoreName1 testKS –certificateAliasList1 testCert1 –keyStoreName2 secondKS –certificate AlaisList2 certAlis}` • Using Jython string: `AdminTask.exchangeSigners ('[-keyStoreName1 testKS –certificateAliasList1 testCert1 –keyStoreName2 secondKS –certificateAlais List2 certAlis]')` • Using Jython list: `AdminTask.exchangeSigners (['-keyStoreName1', 'testKS', '–certificateAliasList1', 'testCert1', '–keyStoreName 2', 'secondKS', '–certifica teAlaisList2', 'certAlis'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask exchangeSigners {-interactive}` • Using Jython string: `AdminTask.exchangeSigners ('[-interactive]')` • Using Jython list: `AdminTask.exchangeSigners (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
getKeyStoreInfo	The **getKeyStoreInfo** command displays the settings of a particular key store.	• Parameters: **- name** The name that uniquely identifies the key store. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: The settings of the key store that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getKeyStore {-name testKS}` • Using Jython string: `AdminTask.getKeyStore ('[-name testKS]')` • Using Jython list: `AdminTask.getKeyStore (['-name', 'testKS'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getKeyStore Info {-interactive}` • Using Jython string: `AdminTask.getKeyStore Info ('[-interactive]')` • Using Jython list: `AdminTask.getKeyStore Info (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listKeyFileAliases	The **listKeyFileAliases** command lists the certificates in a key store file.	• Parameters: **- keyFilePath** The path of the key file. (String, required) **- keyFilePassword** The password for the key file. (String, required) **- keyFileType** The key file type. (String, required) • Returns: A list of certificate aliases.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeyFile Aliases {-keyFilePaht c:\temp\testKeyFile.p12 –keyFilePassword testPwd –keyFileType PKCS12}` • Using Jython string: `AdminTask.listKeyFile Aliases ('[-keyFilePaht c:\temp\testKeyFile.p12 –keyFilePassword testPwd –keyFileType PKCS12]')` • Using Jython list: `AdminTask.listKeyFile Aliases (['-keyFilePaht', 'c:\temp\testKeyFile.p12', '–keyFilePassword', 'test Pwd', '–keyFileType', 'PKCS12'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeyFileA liases {-interactive}` • Using Jython string: `AdminTask.listKeyFileA liases ('[-interactive]')` • Using Jython list: `AdminTask.listKeyFileA liases (['-interactive'])`

Table 7. (continued)

Command name:	Description:	Parameters and return values:	Examples:
listKeyStores	The **listKeyStores** command lists the key store for a particular scope.	• Parameters: **- displayObjectName** Set the value of this parameter to `true` to list the key store configuration objects within a scope. Set the value of this parameter to `false` to list the strings that contain the key store name and management scope. (String, optional) **- scopeName** The name of the scope. (String, optional) • Returns: A list of key stores.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeyStores` • Using Jython: `AdminTask.listKeyStores()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeyStores {-interactive}` • Using Jython string: `AdminTask.listKeyStores ('[-interactive]')` • Using Jython list: `AdminTask.listKeyStores (['-interactive'])`
listKeyStoresTypes	The **listKeyStoresTypes** command lists all valid key store types.	• Parameters: None • Returns: A list of key store types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeyStoreTypes` • Using Jython: `AdminTask.listKeyStoreTypes()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeyStores Types {-interactive}` • Using Jython string: `AdminTask.listKeyStores Types ('[-interactive]')` • Using Jython list: `AdminTask.listKeyStores Types (['-interactive'])`

Commands for the SSLConfigCommands group of the AdminTask object

Use the commands in the SSLConfigCommands group to create and delete SSL configurations. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the SSLConfigCommands group of the AdminTask object:

Table 8.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createSSLConfig	The **createSSLConfig** command creates an SSL configuration that is based on key store and trust store settings. You can use the SSL configuration settings to make the SSL connections.	None	• Parameters: **- alias** The name of the alias. (String, required) **- scopeName** The name of the scope. (String, optional) **- clientKeyAlias** The certificate alias name for the client. (String, optional) **- serverKeyAlias** The certificate alias name for the server. (String, optional) **- type** The type of SSL configuration. (String, optional) **- clientAuthentication** Set the value of this parameter to `true` to request client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- securityLevel** The cipher group that you want to use. Valid values include: `HIGH`, `MEDIUM`, `LOW`, and `CUSTOM`. (String, optional) **- enabledCiphers** A list of ciphers used during SSL handshake. (String, optional) **- jsseProvider** One of the JSSE providers. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSSLConfig {-alias testSSLCfg -clientKeyAlias key1 -serverKeyAlias key2 -trustStoreName trustKS –keyStoreName testKS -keyManagerName testKeyMgr}` • Using Jython string: `AdminTask.createSSLConfig ('[-alias testSSLCfg -clientKeyAlias key1 -serverKeyAlias key2 -trustStoreName trustKS –keyStoreName testKS -keyManagerName testKeyMgr]')` • Using Jython list: `AdminTask.createSSLConfig (['-alias', 'testSSLCfg', '-clientKeyAlias', 'key1', '-serverKeyAlias', 'key2', '-trustStoreName', 'trustKS', '–keyStoreName', 'testKS', '-keyManagerName', 'testKeyMgr'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSSLConfig {-interactive}` • Using Jython string: `AdminTask.createSSLConfig ('[-interactive]')` • Using Jython list: `AdminTask.createSSLConfig (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- clientAuthenticationSupported Set the value of this parameter to `true` to support client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- sslProtocol** The protocol type for the SSL handshake. Valid values include: `SSL_TLS`, `SSL`, `SSLv2`, `SSLv3`, `TLS`, `TLSv1`. (String, optional) **- trustManagerObjectName** A list of trust managers separated by commas. (String, optional) **- trustStoreNames** The key store that holds trust information used to validate the trust from remote connections. (String, required) **- trustStoreScopeName** The management scope name of the trust store. (String, optional)	

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- keyStoreName The key store that holds the personal certificates that provide identity for the connection. (String, required) **- keyStoreScopeName** The management scope name of the key store. (String, optional) **- sslKeyRingName** Specifies a system SSL (SSSL) key ring name. The value for this parameter has no affect unless the SSL configuration type is SSSL. (String, optional) • Returns: The configuration object name of the SSL configuration object that you created.	

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createSSL Config Property	The **createSSL Config Property** command creates a property for an SSL configuration. Use this command to set SSL configuration settings that are different than the settings in the SSL configuration object.	None	• Parameters: **- sslConfigAliasName** The alias name of the SSL configuration. (String, required) **- scopeName** The name of the scope. (String, optional) **- propertyName** The name of the property. (String, required) **- propertyValue** The value of the property. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask createSSLConfig Property {-sslConfigAlias Name NodeDefaultSSLSettings -scopeName (cell):localhost Node01Cell:(node):localhost Node01 -propertyName test. property -propertyValue testValue}` • Using Jython string: `AdminTask.createSSLConfig Property ('[-sslConfig AliasName NodeDefaultSSL Settings -scopeName (cell) :localhostNode01Cell:(node) :localhostNode01 -property Name test.property -property Value testValue]')` • Using Jython list: `AdminTask.createSSLConfig Property (['-sslConfigAlias Name', 'NodeDefaultSSL Settings', '-scopeName', '(cell):localhostNode01Cell: (node):localhostNode01', '-propertyName', 'test. property', '-propertyValue', 'testValue'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSSLConfig Property {-interactive}` • Using Jython string: `AdminTask.createSSLConfig Property ('[-interactive]')` • Using Jython list: `AdminTask.createSSLConfig Property (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteSSLConfig	The **deleteSSLConfig** command deletes the SSL configuration object that you specify from the configuration.	None	• Parameters: **- alias** The name of the alias. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSSLConfig {-alias NodeDefaultSSL Settings -scopeName (cell ):localhostNode01Cell:( node):localhostNode01}` • Using Jython string: `AdminTask.deleteSSLConfig ('[-alias NodeDefaultSSL Settings -scopeName (cell) :localhostNode01Cell:(no de):localhostNode01]')` • Using Jython list: `AdminTask.deleteSSLConfig (['-alias', 'NodeDefault SSLSettings', '-scopeName', '(cell):localhostNode01 Cell:(node):localhost Node01'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSSLConfig {-interactive}` • Using Jython string: `AdminTask.deleteSSLConfig ('[-interactive]')` • Using Jython list: `AdminTask.deleteSSLConfig (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getSSLConfig	The **getSSLConfig** command obtains information about an SSL configuration and displays the settings.	None	• Parameters: **- alias** The name of the alias. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: Information about the SSL configuration that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getSSLConfig {-alias NodeDefaultSSL Settings -scopeName (cell):localhostNode01 Cell:(node):localhost Node01 }` • Using Jython string: `AdminTask.getSSLConfig ('[-alias NodeDefault SSLSettings -scopeName (cell):localhostNode01 Cell:(node):localhost Node01]')` • Using Jython list: `AdminTask.getSSLConfig (['-alias', 'Node DefaultSSLSettings', '-scopeName', '(cell): localhostNode01Cell: (node):localhostNode01'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getSSLConfig {-interactive}` • Using Jython string: `AdminTask.getSSLConfig ('[-interactive]')` • Using Jython list: `AdminTask.getSSLConfig (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getSSLConfig Properties	The **getSSLConfig Properties** command obtains information about SSL configuration properties.	None	• Parameters: **- alias** The name of the alias. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: Information about SSL configuration properties.	**Batch mode example usage:** • Using Jacl: `$AdminTask getSSLConfig Properties {-sslConfig AliasName NodeDefault SSLSettings -scopeName (cell):localhostNode01 Cell:(node):localhost Node01}` • Using Jython string: `AdminTask.getSSLConfig Properties ('[-sslCon figAliasName NodeDefa ultSSLSettings -scope Name (cell):localhost Node01Cell:(node):loc alhostNode01]')` • Using Jython list: `AdminTask.getSSLConfig Properties (['-sslCon figAliasName', 'NodeDe faultSSLSettings', '-scopeName', '(cell): localhostNode01Cell: (node):localhostNode01'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getSSLConfig Properties {-interactive}` • Using Jython string: `AdminTask.getSSLConfig Properties ('[-intera ctive]')` • Using Jython list: `AdminTask.getSSLConfig Properties (['-intera ctive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listSSLCiphers	The **listSSLCiphers** command lists the SSL ciphers.	None	• Parameters: **- sslConfigAliasName** The alias name of the SSL configuration. (String, required) **- scopeName** The name of the scope. (String, optional) **- securityLevel** The cipher group that you want to use. Valid values include: `HIGH`, `MEDIUM`, `LOW`, and `CUSTOM`. (String, required) **- provider** (String, optional) • Returns: A list of SSL ciphers.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSSLCiphers {-sslConfigAliasName test SSLCfg -securityLevel HIGH}` • Using Jython string: `AdminTask.listSSLCiphers ('[-sslConfigAliasName testSSLCfg -security Level HIGH]')` • Using Jython list: `AdminTask.listSSLCiphers (['-sslConfigAliasName', 'testSSLCfg', '-security Level', 'HIGH'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSSLCiphers {-interactive}` • Using Jython string: `AdminTask.listSSLCiphers ('[-interactive]')` • Using Jython list: `AdminTask.listSSLCiphers (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listSSLConfig	The **listSSLConfig** command lists the defined SSL configurations within a management scope.	None	• Parameters: **- scopeName** The name of the scope. (String, optional) **- displayObjectName** Set the value of this parameter to `true` to list the SSL configuration objects within the scope. Set the value of this parameter to `false` to list the strings that contain the SSL configuration alias and management scope. (Boolean, optional) • Returns: A list of the defined SSL configurations.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSSLConfig {-scopeName (cell): localhostNode01Cell:(node): localhostNode01 -displayObjectName true}` • Using Jython string: `AdminTask.listSSLConfig ('[-scopeName (cell): localhostNode01Cell: (node):localhostNode01 -displayObjectName true]')` • Using Jython list: `AdminTask.listSSLConfig (['-scopeName', '(cell) :localhostNode01Cell: (node):localhostNode01', '-displayObjectName', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSSLConfig {-interactive}` • Using Jython string: `AdminTask.listSSLConfig ('[-interactive]')` • Using Jython list: `AdminTask.listSSLConfig (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listSSLConfig Properties	The **listSSLConfig Properties** command lists the properties for a SSL configuration.	None	• Parameters: **- sslConfigAliasName** The alias name of the SSL configuration. (String, required) **- scopeName** The name of the scope. (String, optional) **- displayObjectName** Set the value of this parameter to `true` to list the SSL configuration objects within the scope. Set the value of this parameter to `false` to list the strings that contain the SSL configuration alias and management scope. (Boolean, optional) • Returns: A list of properties.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSSLConfig Property {-alias No deDefaultSSLSettings -scope Name (cell):localhostNode01Cell:(node):localhostNode01 -displayObject Name true}` • Using Jython string: `AdminTask.listSSLConfig Property ('[-alias No deDefaultSSLSettings -scopeName (cell):localhostNode01Cell:(node):localhostNode01 -displayObjectName true]')` • Using Jython list: `AdminTask.listSSLConfigProperty (['-alias', 'No', 'deDefaultSSLSettings', '-scopeName', '(cell):localhostNode01Cell:(node):localhostNode01', '-displayObjectName', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSSLConfig Properties {-interactive}` • Using Jython string: `AdminTask.listSSLConfig Properties ('[-interactive]')` • Using Jython list: `AdminTask.listSSLConfig Properties (['-interactive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifySSLConfig	The **modifySSLConfig** command modifies the settings of an existing SSL configuration.	None	• Parameters: **- alias** The name of the alias. (String, required) **- scopeName** The name of the scope. (String, optional) **- clientKeyAlias** The certificate alias name for the client. (String, optional) **- serverKeyAlias** The certificate alias name for the server. (String, optional) **- type** The type of SSL configuration. (String, optional) **- clientAuthentication** Set the value of this parameter to `true` to request client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- securityLevel** The cipher group that you want to use. Valid values include: `HIGH`, `MEDIUM`, `LOW`, and `CUSTOM`. (String, optional) **- enabledCiphers** A list of ciphers used during SSL handshake. (String, optional) **- jsseProvider** One of the JSSE providers. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySSL Config {-alias test SSLCfg -clientKeyAli as tstKey1 -serverKey Alias tstKey2 -secur ityLevel LOW}` • Using Jython string: `AdminTask.modifySSL Config ('[-alias te stSSLCfg -clientKey Alias tstKey1 -serv erKeyAlias tstKey2 -securityLevel LOW]')` • Using Jython list: `AdminTask.modifySSL Config (['-alias', 'testSSLCfg', '-cli entKeyAlias', 'tst Key1', '-serverKey Alias', 'tstKey2', '-securityLevel', 'LOW'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySSL Config {-interactive}` • Using Jython string: `AdminTask.modifySSL Config ('[-interact ive]')` • Using Jython list: `AdminTask.modifySSL Config (['-interact ive'])`

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- clientAuthenticationSupported Set the value of this parameter to `true` to support client authentication. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- sslProtocol** The protocol type for the SSL handshake. Valid values include: `SSL_TLS`, `SSL`, `SSLv2`, `SSLv3`, `TLS`, `TLSv1`. (String, optional) **- trustManagerObjectNames** A list of trust managers separated by commas. (String, optional) **- trustStoreName** The key store that holds trust information used to validate the trust from remote connections. (String, optional) **- trustStoreScopeName** The management scope name of the trust store. (String, optional)	

Table 8. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- keyStoreName The key store that holds the personal certificates that provide identity for the connection. (String, optional) **- keyStoreScopeName** The management scope name of the key store. (String, optional) **- sslKeyRingName** Specifies a system SSL (SSSL) key ring name. The value for this parameter has no affect unless the SSL configuration type is SSSL. (String, optional) • Returns: None	

Commands for the DescriptivePropCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the DescriptivePropCommands group of the AdminTask object:

Table 9.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createDescriptiveProp	The **createDescriptiveProp** command creates key manager settings in the configuration. Use this command during SSL handshake to determine which certificate to use.	None	• Parameters: **- parentDataType** (String, required) **- parentClassName** (String, required) **- parentScopeName** (String, optional) **- name** (String, required) **- value** (String, required) **- type** (String, required) **- displayNameKey** (String, required) **- nlsRangeKey** (String, optional) **- hoverHelpKey** (String, optional) **- range** (String, optional) **- inclusive** (Boolean, optional) **- firstClass** (Boolean, optional) • Returns: The configuration object name of the key manager object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createKeyManager {-name testKM –keyManager Class com.ibm.ws.security. ltpa.LTPAKeyPairGenerator}` • Using Jython string: `AdminTask.createKeyManager ('[-name testKM –keyManag erClass com.ibm.ws.securi ty.ltpa.LTPAKeyPairGenera tor]')` • Using Jython list: `AdminTask.createKeyManager (['-name', 'testKM', '–key ManagerClass', 'com.ibm.ws .security.ltpa.LTPAKeyPair Generator'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createDescrip tiveProp {-interactive}` • Using Jython string: `AdminTask.createDescripti veProp ('[-interactive]')` • Using Jython list: `AdminTask.createDescripti veProp (['-interactive'])`
deleteDescriptiveProp	The **deleteDescriptiveProp** command deletes key manager settings from the configuration.	None	• Parameters: **- parentDataType** (String, required) **- parentClassName** (String, required) **- parentScopeName** (String, optional) **- name** (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteDescrip tiveProp {-interactive}` • Using Jython string: `AdminTask.deleteDescrip tiveProp ('[-interactive]')` • Using Jython list: `AdminTask.deleteDescrip tiveProp (['-interactive'])`

Table 9. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getDescriptiveProp	The **getDescriptiveProp** command obtains information about key manager settings.	None	• Parameters: **- parentDataType** (String, required) **- parentClassName** (String, required) **- parentScopeName** (String, optional) **- name** (String, required) • Returns: Information about key manager settings.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getDescriptiveProp {-interactive}` • Using Jython string: `AdminTask.getDescriptiveProp ('[-interactive]')` • Using Jython list: `AdminTask.getDescriptiveProp (['-interactive'])`
listDescriptiveProp	The **listDescriptiveProp** command lists the key managers within a particular management scope.	None	• Parameters: **- parentDataType** (String, required) **- parentClassName** (String, required) **- parentScopeName** (String, optional) **- displayObjectName** Set the value of this parameter to `true` to list the key manager objects within the scope. Set the value of this parameter to `false` to list the strings that contain the key manager name and management scope. (Boolean, optional) • Returns: A list of key managers.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listDescriptiveProp {-interactive}` • Using Jython string: `AdminTask.listDescriptiveProp ('[-interactive]')` • Using Jython list: `AdminTask.listDescriptiveProp (['-interactive'])`

Table 9. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifyDescriptiveProp	The **modifyDescriptiveProp** command modifies the settings of an existing key manager.	None	• Parameters: **- parentDataType** (String, required) **- parentClassName** (String, required) **- parentScopeName** (String, optional) **- name** (String, required) **- value** (String, optional) **- type** (String, optional) **- displayNameKey** (String, optional) **- nlsRangeKey** (String, optional) **- hoverHelpKey** (String, optional) **- range** (String, optional) **- inclusive** (Boolean, optional) **- firstClass** (Boolean, optional) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask modifyDescriptiveProp {-interactive}` • Using Jython string: `AdminTask.modifyDescriptiveProp ('[-interactive]')` • Using Jython list: `AdminTask.modifyDescriptiveProp (['-interactive'])`

Commands for the TrustManagerCommands group of the AdminTask object

Use the commands in the TrustManagerCommands group to create and delete a trust manager. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the TrustManagerCommands group of the AdminTask object:

Table 10.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createTrustManager	The **createTrust ManagerInfo** command creates trust manager settings in the configuration. Use this command during SSL handshake to make trust decisions about remote endpoints.	None	• Parameters: **- name** The name that uniquely identifies the trust manager. (String, required) **- scopeName** The name of the scope. (String, optional) **- provider** The provider name of the trust manager. (String, optional) **- algorithm** The algorithm name of the trust manager. (String, optional) **- trustManagerClass** Specifies a class that implements the `javax.net.sslX509TrustManager` interface. You cannot use this parameter with the provider or algorithm parameters. (String, optional) • Returns: The configuration object name of the trust manager object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createTrust Manager {-name testTM –provider IBMJSSE2 –algorithm IbmX509}` • Using Jython string: `AdminTask.createTrust Manager ('[-name testTM –provider IBMJSSE2 –algorithm IbmX509]')` • Using Jython list: `AdminTask.createTrust Manager (['-name', 'testTM', '–provider', 'IBMJSSE2', '–algorithm', 'IbmX509'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createTrust Manager {-interactive}` • Using Jython string: `AdminTask.createTrust Manager ('[-interactive]')` • Using Jython list: `AdminTask.createTrust Manager (['-interactive'])`

Table 10. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteTrustManager	The **deleteTrustManager** command deletes the trust manager settings from the configuration.	None	• Parameters: **- name** The name that uniquely identifies the trust manager. (String, optional) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteTrust Manager {-name testTM}` • Using Jython string: `AdminTask.deleteTrust Manager ('[-name testTM]')` • Using Jython list: `AdminTask.deleteTrust Manager (['-name', 'testTM'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteTrust Manager {-interactive}` • Using Jython string: `AdminTask.deleteTrust Manager ('[-interactive]')` • Using Jython list: `AdminTask.deleteTrust Manager (['-interactive'])`
getTrustManager	The **getTrustManager** command obtains the setting of a trust manager.	None	• Parameters: **- name** The name that uniquely identifies the trust manager. (String, optional) **- scopeName** The name of the scope. (String, optional) • Returns: The settings of the trust manager group that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getTrustManager {-name testTM}` • Using Jython string: `AdminTask.getTrustManager ('[-name testTM]')` • Using Jython list: `AdminTask.getTrustManager (['-name', 'testTM'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getTrustMana ger {-interactive}` • Using Jython string: `AdminTask.getTrustMana ger ('[-interactive]')` • Using Jython list: `AdminTask.getTrustMana ger (['-interactive'])`

Table 10. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listTrustManagers	The **listTrustManagers** command lists the trust managers within a particular management scope.	None	• Parameters: **- scopeName** The name of the scope. (String, optional) **- displayObjectName** Set the value of this parameter to true to list the trust manager objects within a scope. Set the value of this parameter to false to list the strings that contain the trust manager name and management scope. (Boolean, optional) • Returns: A list of trust managers that are found within the management scope that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask listTrustManagers {-displayObjectName true}` • Using Jython string: `AdminTask.listTrustManagers ('[-displayObjectName true]')` • Using Jython list: `AdminTask.listTrustManagers (['-displayObjectName', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listTrustManagers {-interactive}` • Using Jython string: `AdminTask.listTrustManagers ('[-interactive]')` • Using Jython list: `AdminTask.listTrustManagers (['-interactive'])`

Table 10. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifyTrustManager	The **modifyTrustManager** command changes existing trust manager settings.	None	• Parameters: **- name** The name that uniquely identifies the trust manager. (String, required) **- scopeName** The name of the scope. (String, optional) **- provider** The provider name of the trust manager. (String, optional) **- algorithm** The algorithm name of the trust manager. (String, optional) **- trustManagerClass** Specifies a class that implements the `javax.net.sslX509TrustManager` interface. You cannot use this parameter with the provider or algorithm parameters. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyTrustManager {-name testTM –trustManagerClass test.trust.manager}` • Using Jython string: `AdminTask.modifyTrustManager ('[-name testTM –trustManagerClass test.trust.manager]')` • Using Jython list: `AdminTask.modifyTrustManager (['-name', 'testTM', '–trustManagerClass', 'test.trust.manager'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyTrustManager {-interactive}` • Using Jython string: `AdminTask.modifyTrustManager ('[-interactive]')` • Using Jython list: `AdminTask.modifyTrustManager (['-interactive'])`

Commands for the keyManagerCommands group of the AdminTask object

Use the commands in the keyManagerCommands group to manage key managers. You can use these commands to create, modify, list, or obtain information about key managers. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the keyManagerCommands group of the AdminTask object:

Table 11.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createKeyManager	The **createKeyManager** command creates the key manager settings in the configuration. Use this command during SSL handshake to determine which certificate alias to use.	None	• Parameters: **- name** The name that uniquely identifies the key manager. (String, required) **- scopeName** The name of the scope. (String, optional) **- provider** The provider name of the key manager. (String, optional) **- algorithm** The algorithm name of the key manager. (String, optional) **- keyManagerClass** The name of the key manager implementation class. You can not use this parameter with the provider or the algorithm parameter. (String, optional) • Returns: The configuration object name of the key manager object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createKeyManager {-name testKM –keyManagerClass com.ibm.ws.security.ltpa.LTPAKeyPairGenerator}` • Using Jython string: `AdminTask.createKeyManager ('[-name testKM –keyManagerClass com.ibm.ws.security.ltpa.LTPAKeyPairGenerator]')` • Using Jython list: `AdminTask.createKeyManager (['-name', 'testKM', '–keyManagerClass', 'com.ibm.ws.security.ltpa.LTPAKeyPairGenerator'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createKeyManager {-interactive}` • Using Jython string: `AdminTask.createKeyManager ('[-interactive]')` • Using Jython list: `AdminTask.createKeyManager (['-interactive'])`

Table 11. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteKeyManager	The **deleteKeyManager** command deletes the key manager settings from the configuration.	None	• Parameters: **- name** The name that uniquely identifies the key manager. (String, optional) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteKeyMa nager {-name testKM}` • Using Jython string: `AdminTask.deleteKeyMa nager ('[-name testKM]')` • Using Jython list: `AdminTask.deleteKeyMa nager (['-name', 'testKM'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteKeyM anager {-interactive}` • Using Jython string: `AdminTask.deleteKeyMa nager ('[-interactive]')` • Using Jython list: `AdminTask.deleteKeyM anager (['-interactive'])`
getKeyManager	The **getKeyManager** command obtains the settings of a key manager.	None	• Parameters: **- name** The name that uniquely identifies the key manager. (String, optional) **- scopeName** The name of the scope. (String, optional) • Returns: The settings of the key manager that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getKeyMana ger {-name testKM}` • Using Jython string: `AdminTask.getKeyMana ger ('[-name testKM]')` • Using Jython list: `AdminTask.getKeyMana ger (['-name', 'testKM'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getKeyManag er {-interactive}` • Using Jython string: `AdminTask.getKeyManag er ('[-interactive]')` • Using Jython list: `AdminTask.getKeyManag er (['-interactive'])`

Table 11. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listKeyManagers	The **listKeyManagers** command lists the key managers within a particular management scope.	None	• Parameters: **- scopeName** The name of the scope. (String, optional) **- displayObjectName** Set the value of this parameter to `true` to list the key manager objects within the scope. Set the value of this parameter to `false` to list the strings that contain the key manager name and the management scope. (Boolean, optional) • Returns: A list of key managers.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeyManagers` • Using Jython: `AdminTask.listKeyManagers()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeyManagers {-interactive}` • Using Jython string: `AdminTask.listKeyManagers ('[-interactive]')` • Using Jython list: `AdminTask.listKeyManagers (['-interactive'])`

Table 11. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifyKeyManagers	The **modifyKeyManagers** command changes existing key manager settings.	None	• Parameters: **- name** The name that uniquely identifies the key manager. (String, required) **- scopeName** The name of the scope. (String, optional) **- provider** The provider name of the key manager. (String, optional) **- algorithm** The algorithm name of the key manager. (String, optional) **- keyManagerClass** The name of the key manager implementation class. You can not use this parameter with the provider or the algorithm parameter. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyKeyManager {-name testKM –provider IBMJSSE2 –algorithm IbmX509}` • Using Jython string: `AdminTask.modifyKeyManager ('[-name testKM –provider IBMJSSE2 –algorithm IbmX509]')` • Using Jython list: `AdminTask.modifyKeyManager (['-name', 'testKM', '–provider', 'IBMJSSE2', '–algorithm', 'IbmX509'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyKeyManagers {-interactive}` • Using Jython string: `AdminTask.modifyKeyManagers ('[-interactive]')` • Using Jython list: `AdminTask.modifyKeyManagers (['-interactive'])`

Commands for the SSLConfigGroupCommands group of the AdminTask object

Use the commands in the SSLConfigGroupCommands group to create or delete a SSL configuration group. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the SSLConfigGroupCommands group of the AdminTask object:

Table 12.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createSSLConfigGroup	The **createSSLConfigGroup** command creates a SSL configuration group.	None	• Parameters: **- name** The name that uniquely identifies the SSL configuration group. (String, required) **- direction** The direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, required) **- certificateAlias** (String, required) **- scopeName** The name of the scope. (String, optional) **- sslConfigAliasName** The alias that uniquely identifies the SSL configurations in the group. (String, required) **- sslConfigScopeName** The scope that uniquely identifies the SSL configurations in the group. (String, optional) • Returns: The configuration object name of the SSL configuration group object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createSSLConfigGroup { -name testSSLCfgGrp –direction inbound –certificateAlias alias1 –sslConfigAliasName testSSLCfg}` • Using Jython string: `AdminTask.createSSLConfigGroup ['(-name testSSLCfgGrp –direction inbound –certificateAlias alias1 –sslConfigAliasName testSSLCfg)']` • Using Jython list: `AdminTask.createSSLConfigGroup [('-name', 'testSSLCfgGrp', '–direction', 'inbound', '–certificateAlias', 'alias1', '–sslConfigAliasName', 'testSSLCfg')]` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSSLConfigGroup {-interactive}` • Using Jython string: `AdminTask.createSSLConfigGroup ('[-interactive]')` • Using Jython list: `AdminTask.createSSLConfigGroup (['-interactive'])`

Table 12. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteSSLConfigGroup	The **deleteSSLConfigGroup** command deletes a SSL configuration group from the configuration.	None	• Parameters: **- name** The name that uniquely identifies the SSL configuration group. (String, required) **- direction** The direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSSLConfigGroup {-name createSSLCfgGrp –direction inbound}` • Using Jython string: `AdminTask.deleteSSLConfigGroup ['(-name createSSLCfgGrp –direction inbound)']` • Using Jython list: `AdminTask.deleteSSLConfigGroup [('-name', 'createSSLCfgGrp', '–direction', 'inbound')]` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSSLConfigGroup {-interactive}` • Using Jython string: `AdminTask.deleteSSLConfigGroup ('[-interactive]')` • Using Jython list: `AdminTask.deleteSSLConfigGroup (['-interactive'])`

Table 12. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getSSLConfigGroup	The **getSSLConfigGroup** command returns information about a SSL configuration setting.	None	• Parameters: **- name** The name that uniquely identifies the SSL configuration group. (String, required) **- direction** The direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: The settings of the SSL configuration group that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getSSLConfig Group {-name createSSL CfgGrp –direction inbound}` • Using Jython string: `AdminTask.getSSLConfig Group ['(-name create SSLCfgGrp –direction inbound)']` • Using Jython list: `AdminTask.getSSLConfig Group [('-name', 'create SSLCfgGrp', '–direction', 'inbound')]` **Interactive mode example usage:** • Using Jacl: `$AdminTask getSSLConfig Group {-interactive}` • Using Jython string: `AdminTask.getSSLConfig Group ('[-interactive]')` • Using Jython list: `AdminTask.getSSLConfig Group (['-interactive'])`

Table 12. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listSSLConfigGroups	The **listSSLConfigGroups** command lists the SSL configuration groups within a scope and a direction.	None	• Parameters: **- direction** The direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, optional) **- scopeName** The name of the scope. (String, optional) **- displayObjectName** If you set this parameter to true, the command returns a list of all of the SSL configuration group objects within the scope. If you set this parameter to false, the command returns a list of strings that contain the SSL configuration name and management scope. (Boolean, optional) • Returns: A list of SSL configuration groups.	**Batch mode example usage:** • Using Jacl: `$AdminTask –listSSLConfigGroups {-displayObjectName true}` • Using Jython string: `AdminTask.listSSLConfigGroups ['(-displayObjectName true)']` • Using Jython list: `AdminTask.listSSLConfigGroups [('-displayObjectName' 'true')]` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSSLConfigGroups {-interactive}` • Using Jython string: `AdminTask.listSSLConfigGroups ('[-interactive]')` • Using Jython list: `AdminTask.listSSLConfigGroups (['-interactive'])`

Table 12. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifySSLConfigGroup	The **modifySSLConfigGroup** command modifies the setting of an existing SSL configuration group.	None	• Parameters: **- name** The name that uniquely identifies the SSL configuration group. (String, required) **- direction** The direction to which the SSL configuration applies. Valid values include inbound or outbound. (String, required) **- certificateAlias** (String, optional) **- scopeName** The name of the scope. (String, optional) **- sslConfigAliasName** The alias that uniquely identifies the SSL configurations in the group. (String, optional) **- sslConfigScopeName** The scope that uniquely identifies the SSL configurations in the group. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySSLConfig Group {-name createSSLCfg Grp –direction inbound –certificateAlias alias2}` • Using Jython string: `AdminTask.modifySSLConfig Group ['(-name createSSL CfgGrp –direction inbound –certificateAlias alias2)']` • Using Jython list: `AdminTask.modifySSLConfig Group [('-name', 'create SSLCfgGrp', '–direction', 'inbound', '–certificate Alias', 'alias2')]` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySSLConf igGroup {-interactive}` • Using Jython string: `AdminTask.modifySSLConf igGroup ('[-interactive]')` • Using Jython list: `AdminTask.modifySSLConf igGroup (['-interactive'])`

Commands for the DynamicSSLConfigSelections group of the AdminTask object

Use the commands in the DynamicSSLConfigSelections group to create or delete a dynamic SSL configuration selection. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the DynamicSSLConfigSelections group of the AdminTask object:

Table 13.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createDynamic SSLConfig Selection	The **createDynamic SSLConfig Selection** command creates the configuration settings for the dynamic SSL configuration selection.	None	• Parameters: **- dynSSLConfigSelectionName** The name that uniquely identifies the dynamic SSL configuration selection. (String, required) **- scopeName** The name of the scope. (String, optional) **- dynSSLConfigSelectionDescription** The description of the dynamic SSL configuration selection. (String, optional) **- dynSSLConfigSelectionInfo** The information for the dynamic SSL configuration selection. (String, required) **- sslConfigName** The name of the SSL configuration. (String, required) **- sslConfigScope** The scope of the SSL configuration. (String, optional) **- certificateAlias** The alias name to identify the certificate. (String, required) • Returns: The configuration object name of the dynamic SSL configuration selection object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createDynamicSSLConfigSelection` • Using Jython: `AdminTask.createDynamicSSLConfigSelection()` **Interactive mode example usage:** • Using Jacl: `$AdminTask createDynamicSSLConfigSelection {-interactive}` • Using Jython string: `AdminTask.createDynamicSSLConfigSelection ('[-interactive]')` • Using Jython list: `AdminTask.createDynamicSSLConfigSelection (['-interactive'])`

Table 13. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteDynamic SSLConfig Selection	The **deleteDynamic SSLConfig Selection** command deletes the dynamic SSL configuration selection from the configuration.	None	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteDynamic SSLConfigSelection` • Using Jython: `AdminTask.deleteDynamic SSLConfigSelection()` • **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteDynamic SSLConfigSelection {-interactive}` • Using Jython string: `AdminTask.deleteDynamicSSLConfigSelection ('[-interactive]')` • Using Jython list: `AdminTask.deleteDynamic SSLConfigSelection (['-interactive'])`
getDynamic SSLConfig Selection	The **getDynamic SSLConfig Selection**command obtains information about a particular dynamic SSL configuration selection.	None	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask getDynamicSSL ConfigSelection` • Using Jython: `AdminTask.getDynamicSSL ConfigSelection()` **Interactive mode example usage:** • Using Jacl: `$AdminTask getDynamicSSL ConfigSelection {-interactive}` • Using Jython string: `AdminTask.getDynamicSSL ConfigSelection ('[-interactive]')` • Using Jython list: `AdminTask.getDynamicSSL ConfigSelection (['-interactive'])`

Table 13. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listDynamic SSLConfig Selections	The **listDynamic SSLConfig Selections**command lists the configuration objects name for a dynamic SSL configuration selection.	None	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask listDynamic SSLConfigSelection` • Using Jython: `AdminTask.listDynamic SSLConfigSelection()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listDynamicSSL ConfigSelections {-intera ctive}` • Using Jython string: `AdminTask.listDynamicSSL ConfigSelections ('[-inte ractive]')` • Using Jython list: `AdminTask.listDynamicSSL ConfigSelections (['-int eractive'])`

Commands for the ManagementScopeCommands group of the AdminTask object

Use the commands in the ManagementScopeCommands group to create or delete a management scope. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the ManagementScopeCommands group of the AdminTask object:

Table 14.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createManagementScope	The **createManagementScope** command creates a management scope setting in the configuration.	None	• Parameters: **- scopeName** The name that uniquely identifies the management scope. (String, required) **- scopeType** The type of the management scope. Valid types include cell, node, nodegroup, cluster, server, and endpoint. (String, optional) • Returns: The configuration object name of the management scope object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createManagementScope {-name (cell):localhostNode01Cell –scopeType cell}` • Using Jython string: `AdminTask.createManagementScope ['(-name (cell):localhostNode01Cell –scopeType cell)']` • Using Jython list: `AdminTask.createManagementScope [('-name', '(cell):localhostNode01Cell', '–scopeType', 'cell')]` **Interactive mode example usage:** • Using Jacl: `$AdminTask createManagementScope {-interactive}` • Using Jython string: `AdminTask.createManagementScope ('[-interactive]')` • Using Jython list: `AdminTask.createManagementScope (['-interactive'])`

Table 14. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteManage mentScope	The **deleteManage mentScope** command deletes a management object from the configuration.	None	• Parameters: **- scopeName** The name that uniquely identifies the management scope. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteManage mentScope {-scopeName (cell):localhostNode01Cell}` • Using Jython string: `AdminTask.deleteManagem entScope ('[-scopeName (cell):localhostNode01 Cell]')` • Using Jython list: `AdminTask.deleteManagem entScope (['-scopeName', '(cell):localhostNode01Cell'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteManage mentScope {-interactive}` • Using Jython string: `AdminTask.deleteManage mentScope ('[-interactive]')` • Using Jython list: `AdminTask.deleteManage mentScope (['-interactive'])`

Table 14. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getManagementScope	The **getManagementScope** command displays the setting of a management scope object.	None	• Parameters: **- scopeName** The name that uniquely identifies the management scope. (String, required) • Returns: The settings of the management scope object.	**Batch mode example usage:** • Using Jacl: `$AdminTask getManageme ntScope {-scopeName (ce ll):localhostNode01Cell}` • Using Jython string: `AdminTask.getManagement Scope ('[-scopeName (ce ll):localhostNode01Cell]')` • Using Jython list: `AdminTask.getManagement Scope (['-scopeName', '(cell):localhostNode01 Cell'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getManagement Scope {-interactive}` • Using Jython string: `AdminTask.getManagement Scope ('[-interactive]')` • Using Jython list: `AdminTask.getManagement Scope (['-interactive'])`

Table 14. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listManagementScopes	The **listManagementScopes** command lists the management scopes in the configuration.	None	• Parameters: **-displayObjectName** Set the value to true to display the object names of the management scope. (Boolean, optional) • Returns: A list that contains all of the management scope names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listManagem entScopes { -name testKM –provider IBMJSSE2 –alg orithm IbmX509}` • Using Jython string: `AdminTask.listManageme ntScopes ('[-name test KM –provider IBMJSSE2 –algorithm IbmX509]')` • Using Jython list: `AdminTask.listManageme ntScopes (['-name', 'testKM', '–provider', 'IBMJSSE2', '–algorithm', 'IbmX509'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listManagem entScopes {-interactive}` • Using Jython string: `AdminTask.listManagem entScopes ('[-interac tive]')` • Using Jython list: `AdminTask.listManagem entScopes (['-interac tive'])`

Commands for the WSCertExpMonitorCommands group of the AdminTask object

Use the commands in the WSCertExpMonitorCommands group to start or update the certificate expiration monitor. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the WSCertExpMonitorCommands group of the AdminTask object:

Table 15.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createWSCert ExpMonitor	The **createWSCert ExpMonitor** command creates the certificate expiration monitor settings in the configuration.	None	• Parameters: **- name** The name that uniquely identifies the certificate expiration monitor. (String, required) **- autoReplace** Set the value of this parameter to true if you want to replace a certificate within a certificate expiration date. If not, set the value of this parameter to false. (Boolean, required) **- deleteOld** Set the value of this parameter to true if you want to delete an old certificate during certificate expiration monitoring. If not, set the value of this parameter to false. (Boolean, required) **- daysBeforeNotification** The number of days before a certificate expires that you want to be notified of the expiration. (Integer, required) **- wsScheduleName** The name of the scheduler to use for certificate expiration. (String, required)	**Batch mode example usage:** • Using Jacl: `$AdminTask createWSCert ExpMonitor {-name test CertMon –autoReplace true –deleteOld true –days BeforeNotification 30 –wsScheduleName testSch edule –wsNotificationName testNotifier –isEnabled false}` • Using Jython string: `AdminTask.createWSCert ExpMonitor ('[-name tes tCertMon –autoReplace true –deleteOld true –daysBeforeNotification 30 –wsScheduleName test Schedule –wsNotificati onName testNotifier –isEnabled false]')` • Using Jython list: `AdminTask.createWSCert ExpMonitor (['-name', 'testCertMon', '–autoRe place', 'true', '–delete Old', 'true', '–daysBe foreNotification', '30', '–wsScheduleName', 'te stSchedule', '–wsNotifi cationName', 'testNotif ier', '–isEnabled', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createWSCert ExpMonitor {-interactive}` • Using Jython string: `AdminTask.createWSCert ExpMonitor ('[-interac tive]')` • Using Jython list: `AdminTask.createWSCert ExpMonitor (['-interac tive'])`

Table 15. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- wsNotificationName The name of the notifier to use for certificate expiration. (String, required) **- isEnabled** Set the value of this parameter to true if the certificate expiration monitor is enabled. If not, set the value of this parameter to false. (Boolean, optional) • Returns: The configuration object name of the certificate expiration monitor object that you created.	
deleteWSCert ExpMonitor	The **deleteWSCert ExpMonitor**command deletes the settings of a scheduler from the configuration.	None	• Parameters: **- name** The name that uniquely identifies the certificate expiration monitor. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteWSCert ExpMonitor {-name test CertMon}` • Using Jython string: `AdminTask.deleteWSCertE xpMonitor ('[-name test CertMon]')` • Using Jython list: `AdminTask.deleteWSCert ExpMonitor (['-name', 'testCertMon'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteWSCert ExpMonitor {-interactive}` • Using Jython string: `AdminTask.deleteWSCert ExpMonitor ('[-interac tive]')` • Using Jython list: `AdminTask.deleteWSCert ExpMonitor (['-interac tive'])`

Table 15. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getWSCertExpMonitor	The **getWSCertExpMonitor** command displays the settings of a particular scheduler.	None	• Parameters: **- name** The name that uniquely identifies the certificate expiration monitor. (String, required) • Returns: The settings of the scheduler that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask getWSCertExp Monitor {-name testCertMon}` • Using Jython string: `AdminTask getWSCertExp Monitor ('[-name testCe rtMon]')` • Using Jython list: `AdminTask getWSCertExpM onitor (['-name', 'test CertMon'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getWSCertExp Monitor {-interactive}` • Using Jython string: `AdminTask.getWSCertExp Monitor ('[-interactive]')` • Using Jython list: `AdminTask.getWSCertExp Monitor (['-interactive'])`

Table 15. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listWSCertExpMonitor	The **listWSCertExpMonitor** command lists the scheduler in the configuration.	None	• Parameters: **-displayObjectNames** If you set the value of this parameter to true, the command returns the certificate expiration monitor configuration object. If you set the value of this parameter to false, the command returns the name of the certificate expiration monitor. (Boolean, optional) • Returns: The scheduler in the configuration.	**Batch mode example usage:** • Using Jacl: `$AdminTask listWSCertExp Monitor {-displayObject Name false}` • Using Jython string: `AdminTask.listWSCertExp Monitor ('[-displayObject Name false]')` • Using Jython list: `AdminTask.listWSCertExp Monitor (['-displayObject Name', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listWSCert ExpMonitor {-interactive}` • Using Jython string: `AdminTask.listWSCertExp Monitor ('[-interactive]')` • Using Jython list: `AdminTask.listWSCertExp Monitor (['-interactive'])`

Table 15. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifyWSCert ExpMonitor	The **modifyWSCert ExpMonitor**command changes the setting of an existing scheduler.	None	• Parameters: **- name** The name that uniquely identifies the certificate expiration monitor. (String, required) **- autoReplace** Set the value of this parameter to true if you want to replace a certificate within a certificate expiration date. If not, set the value of this parameter to false. (Boolean, required) **- deleteOld** Set the value of this parameter to true if you want to delete an old certificate during certificate expiration monitoring. If not, set the value of this parameter to false. (Boolean, required) **- daysBeforeNotification** The number of days before a certificate expires that you want to be notified of the expiration. (Integer, required) **- wsScheduleName** The name of the scheduler to use for certificate expiration. (String, required)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyWSCert ExpMonitor {-name testC ertMon –autoReplace fal se –deleteOld false –d aysBeforeNotification 20 –isEnabled true}` • Using Jython string: `AdminTask.modifyWSCert ExpMonitor ('[-name t estCertMon –autoReplac e false –deleteOld fal se –daysBeforeNotifica tion 20 –isEnabled true]')` • Using Jython list: `AdminTask.modifyWSCert ExpMonitor (['-name', 'testCertMon', '–autoR eplace', 'false', '–de leteOld', 'false', '–d aysBeforeNotification', '20', '–isEnabled', 'tr ue'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyWSCert ExpMonitor {-interactive}` • Using Jython string: `AdminTask.modifyWSCert ExpMonitor ('[-interac tive]')` • Using Jython list: `AdminTask.modifyWSCert ExpMonitor (['-intera ctive'])`

Table 15. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- wsNotificationName The name of the notifier to use for certificate expiration. (String, required) **- isEnabled** Set the value of this parameter to true if the certificate expiration monitor is enabled. If not, set the value of this parameter to false. (Boolean, optional) • Returns: None	
startCertificate ExpMonitor	The **startCertificate ExpMonitor**command performs certificate monitoring. This command visits all key stores and checks to see if they are within certificate expiration range.	None	• Parameters: None • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask startCertificateExpMonitor` • Using Jython: `AdminTask.startCertificateExpMonitor()` **Interactive mode example usage:** • Using Jacl: `$AdminTask startCertificateExpMonitor {-interactive}` • Using Jython string: `AdminTask.startCertificateExpMonitor ('[-interactive]')` • Using Jython list: `AdminTask.startCertificateExpMonitor (['-interactive'])`

Commands for the KeySetGroupCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the KeySetGroupCommands group of the AdminTask object:

Table 16.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createKeySetGroup	The **createKeySetGroup** command creates the key set group settings in the configuration. Use this command to manage groups of public, private, and shared keys.	None	• Parameters: **- name** The name that uniquely identifies the key set group. (String, required) **- scopeName** The name of the scope. (String, optional) **- autoGenerate** Set the value of this parameter to true if you want to automatically generate keys. If not, set the value to false. (Boolean, optional) **- wsScheduleName** The name of the scheduler to use to perform key generation. (String, required) **- keySetObjectNames** A list of key set configuration names separated by colons (:). (String, required) • Returns: The configuration object name of the key set group object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createKeySet Group {-name keySetGrp –autoGenerate true –wsS cheduleName testSchedule –keySetObjectNames test KeySet(cells/localhost Node01Cell\|security.xml# KeySet_1130354347825)}` • Using Jython string: `AdminTask.createKeySetGr oup('[-name keySetGrp – autoGenerate true –wsSch eduleName testSchedule –keySetObjectNames test KeySet(cells/localhostN ode01Cell\|security.xml# KeySet_1130354347825)]')` • Using Jython list: `AdminTask.createKeySetGr oup(['-name', 'keySetGrp', '–autoGenerate', 'true', '–wsScheduleName', 'test Schedule', '–keySetObject Names', 'testKeySet(cells /localhostNode01Cell\|sec urity.xml#KeySet_1130354 347825)'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createKey SetGroup {-interactive}` • Using Jython string: `AdminTask.createKeySet Group ('[-interactive]')` • Using Jython list: `AdminTask.createKeySet Group (['-interactive'])`

Table 16. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteKeySetGroup	The **deleteKeySetGroup** command deletes the settings of a key set group from the configuration.	None	• Parameters: **- name** The name that uniquely identifies the key set group. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteKeySet Group {-name keySetGrp }` • Using Jython string: `AdminTask.deleteKeySet Group ('[-name keySetGrp]')` • Using Jython list: `AdminTask.deleteKeySet Group (['-name', 'key SetGrp'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteKey SetGroup {-interactive}` • Using Jython string: `AdminTask.deleteKeySet Group ('[-interactive]')` • Using Jython list: `AdminTask.deleteKeySet Group (['-interactive'])`

Table 16. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
generateKeys ForKey SetGroup	The **generateKeys ForKey SetGroup**command generates keys for all of the keys in the key sets that make up the key set group.	None	• Parameters: **-keySetGroupName** The name of the key set group. (String, required) **-keySetGroupScope** The scope of the key set group. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask generateKey ForKeySetGroup {-keySet GroupName keySetGrp}` • Using Jython string: `AdminTask.generateKey ForKeySetGroup ('[-key SetGroupName keySetGrp]')` • Using Jython list: `AdminTask.generateKey ForKeySetGroup (['-key SetGroupName', 'keySetGrp'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask generateKeys ForKeySetGroup {-intera ctive}` • Using Jython string: `AdminTask.generateKeys ForKeySetGroup ('[-int eractive]')` • Using Jython list: `AdminTask.generateKeys ForKeySetGroup (['-inte ractive'])`

Table 16. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getKeySetGroups	The **getKeySetGroups** command displays the settings of a particular key set group.	None	• Parameters: **- name** The name that uniquely identifies the key set group. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: The settings of the specified key set group.	**Batch mode example usage:** • Using Jacl: `$AdminTask getKeySetGroup { -name keySetGrp }` • Using Jython string: `AdminTask.getKeySetGroup ('[-name keySetGrp]')` • Using Jython list: `AdminTask.getKeySetGroup (['-name', 'keySetGrp'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getKeySetGroups {-interactive}` • Using Jython string: `AdminTask.getKeySetGroups ('[-interactive]')` • Using Jython list: `AdminTask.getKeySetGroups (['-interactive'])`

Table 16. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listKeySetGroups	The **listKeySetGroups** command lists the key set groups for a particular scope.	None	• Parameters: **- scopeName** The name of the scope. (String, optional) **- displayObjectNames** If you set the value of this parameter to true, the command returns a list of all of the key set group objects within a scope. If you set the value of this parameter to false, the command returns a list of strings that contain the key set group name and management scope. (Boolean, optional) • Returns: A list of key set groups.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeySetGroup {-displayObjectName true}` • Using Jython string: `AdminTask.listKeySetGroup ('[-displayObjectName true]')` • Using Jython list: `AdminTask.listKeySetGroup (['-displayObjectName', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeySetGroups {-interactive}` • Using Jython string: `AdminTask.listKeySetGroups ('[-interactive]')` • Using Jython list: `AdminTask.listKeySetGroups (['-interactive'])`

Table 16. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifyKeySetGroup	The **modifyKeySetGroup** command changes the settings of an existing key set group.	None	• Parameters: **- name** The name that uniquely identifies the key set group. (String, required) **- scopeName** The name of the scope. (String, optional) **- autoGenerate** Set the value of this parameter to true if you want to automatically generate keys. If not, set the value to false. (Boolean, optional) **- wsScheduleName** The name of the scheduler to use to perform key generation. (String, optional) **- keySetObjectNames** A list of key set configuration names separated by colons (:). (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyKeySet Group {-name keySetGrp –autoGenetate false}` • Using Jython string: `AdminTask.modifyKeySet Group ('[-name keySetGrp –autoGenetate false]')` • Using Jython list: `AdminTask.modifyKeySet Group (['-name', 'key SetGrp', '–autoGenetate', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyKeySet Group {-interactive}` • Using Jython string: `AdminTask.modifyKeySet Group ('[-interactive]')` • Using Jython list: `AdminTask.modifyKeySet Group (['-interactive'])`

Commands for the KeySetCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the KeySetCommands group of the AdminTask object:

Table 17.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createKeySet	The **createKeySet** command creates the key set settings in the configuration. Use this command to control key instances that have the same type.	None	• Parameters: **- name** The name that uniquely identifies the key set. (String, required) **- scopeName** The name of the scope. (String, optional) **- aliasPrefix** The prefix for the key alias when a new key generates. (String, required) **- password** The password that protects the key in the key store. (String, required) **- maxKeyReferences** The maximum number of key references returned keys from this key set. (Integer, required) **- deleteOldKeys** Set the value of this parameter to `true` to delete old keys when new keys are generated. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- keyGenerationClass** The class that is used to generate new keys in the key set. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createKeySet {-name testKeySet –alias Prefix test –password pwd –maxKeyReferences 2 –deleteOldKeys true –keyStoreName testKeyStore –isKeyPair false}` • Using Jython string: `AdminTask.createKeySet('[-name testKeySet –alias Prefix test –password pwd –maxKeyReferences 2 –deleteOldKeys true –keyStoreName testKeyStore –isKeyPair false]')` • Using Jython list: `AdminTask.createKeySet(['-name', 'testKeySet', '–aliasPrefix', 'test', '–password', 'pwd', '–maxKeyReferences', '2', '–deleteOldKeys', 'true', '–keyStoreName', 'testKeyStore', '–isKeyPair', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createKeySet {-interactive}` • Using Jython string: `AdminTask.createKeySet('[-interactive]')` • Using Jython list: `AdminTask.createKeySet(['-interactive'])`

Table 17. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- keyStoreName The key store that contains the keys. (String, required) **- keyStoreScopeName** The management scope where the key store is located. (String, optional) **- isKeyPair** Set the value of this parameter to `true` if the keys in the key set are key pairs. Otherwise, set the value of this parameter to `false`. (Boolean, optional) • Returns: The configuration object name of the key set object that you created.	
deleteKeySet	The **deleteKeySet** command deletes the settings of a key set from the configuration.	None	• Parameters: **- name** The name that uniquely identifies the key set. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteKeySet {-name testKeySet}` • Using Jython string: `AdminTask.deleteKeySet ('[-name testKeySet]')` • Using Jython list: `AdminTask.deleteKeySet (['-name', 'testKeySet'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteKeySet {-interactive}` • Using Jython string: `AdminTask.deleteKeySet ('[-interactive]')` • Using Jython list: `AdminTask.deleteKeySet (['-interactive'])`

Table 17. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
generateKey ForKeySet	The **generateKey ForKeySet** command generates keys for the keys in the key set.	None	• Parameters: **- keySetName** The name of the key set. (String, required) **- keySetScope** The scope of the key set. (String, optional) **- keySetSaveConfig** Set the value of this parameter to `true` to save the configuration of the key set. Otherwise, set the value of this parameter to `false`. (Boolean, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask generateKey ForKeySet{ -keySetName testKeySet }` • Using Jython string: `AdminTask.generateKeyFor KeySet('[-keySetName testKeySet]')` • Using Jython list: `AdminTask.generateKeyFor KeySet(['-keySetName', 'testKeySet'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask generateKeyFor KeySet {-interactive}` • Using Jython string: `AdminTask.generateKeyFo rKeySet ('[-interactive]')` • Using Jython list: `AdminTask.generateKeyFo rKeySet (['-interactive'])`
getKeySet	The **getKeySet** command displays the settings of a particular key set.	None	• Parameters: **- name** The name that uniquely identifies the key set. (String, required) **- scopeName** The name of the scope. (String, optional) • Returns: The settings of the specified key set group.	**Batch mode example usage:** • Using Jacl: `$AdminTask getKeySet {-name testKeySet}` • Using Jython string: `AdminTask.getKeySet ('[-name testKeySet]')` • Using Jython list: `AdminTask.getKeySet (['-name', 'testKeySet'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getKeySet {-interactive}` • Using Jython string: `AdminTask.getKeySet ('[-interactive]')` • Using Jython list: `AdminTask.getKeySet (['-interactive'])`

Table 17. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listKeySets	The **listKeySets** command lists the key sets in a particular scope.	None	• Parameters: **- scopeName** The name of the scope. (String, optional) **- 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` if you want to list the strings that contain the key set group name and management scope. (Boolean, optional) • Returns: The key sets for the scope that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeySets {-displayObjectName true}` • Using Jython string: `AdminTask.listKeySets ('[-displayObjectName true]')` • Using Jython list: `AdminTask.listKeySets (['-displayObjectName', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeySets {-interactive}` • Using Jython string: `AdminTask.listKeySets ('[-interactive]')` • Using Jython list: `AdminTask.listKeySets (['-interactive'])`

Table 17. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
modifyKeySet	The **modifyKeySet** command changes the settings of an existing key set.	None	• Parameters: **- name** The name that uniquely identifies the key set. (String, required) **- scopeName** The name of the scope. (String, optional) **- aliasPrefix** The prefix for the key alias when a new key generates. (String, optional) **- password** The password that protects the key in the key store. (String, optional) **- maxKeyReferences** The maximum number of key references returned keys from this key set. (Integer, optional) **- deleteOldKeys** Set the value of this parameter to `true` to delete old keys when new keys are generated. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- keyGenerationClass** The class that is used to generate new keys in the key set. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifyKeySet {-name testKeySet -maxKeyReferences 3 –deleteOldKeys false}` • Using Jython string: `AdminTask.modifyKeySet ('[-name testKeySet -maxKeyReferences 3 –deleteOldKeys false]')` • Using Jython list: `AdminTask.modifyKeySet (['-name', 'testKeySet', '-maxKeyReferences', '3', '–deleteOldKeys', 'false'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifyKeySet {-interactive}` • Using Jython string: `AdminTask.modifyKeySet ('[-interactive]')` • Using Jython list: `AdminTask.modifyKeySet (['-interactive'])`

Table 17. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- keyStoreName The key store that contains the keys. (String, optional) **- keyStoreScopeName** The management scope where the key store is located. (String, optional) **- isKeyPair** Set the value of this parameter to `true` if the keys in the key set are key pairs. Otherwise, set the value of this parameter to `false`. (Boolean, optional) • Returns: None	

Commands for the KeyReferenceCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the KeyReferenceCommands group of the AdminTask object:

Table 18.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createKeyReference	The **createKeyReference** command creates the key reference setting in the configuration for key set objects.	None	• Parameters: **- keySetName** The name that uniquely identifies the key set to which the key reference belongs. (String, required) **- keySetScope** The management scope of the key set. (String, optional) **- keyAlias** The alias name that identifies the key for the key set that you specify. (String, required) **- keyPassword** The password used for encrypting the key. (String, optional) **- keyPasswordVerify** The password used for encrypting the key. (String, optional) **- version** The version of the key reference. (String, optional) **- keyReferenceSaveConfig** Set the value of this parameter to `true` to save the key reference to the configuration. Otherwise, set the value to `false`. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createKeyReference {-keySetName testKeySet -keyAlias testKey –password testPWD –passwordVerify testPWD –keyReferenceSaveConfig true}` • Using Jython string: `AdminTask.createKeyReference ('[-keySetName testKeySet -keyAlias testKey –password testPWD –passwordVerify testPWD –keyReferenceSaveConfig true]')` • Using Jython list: `AdminTask.createKeyReference (['-keySetName', 'testKeySet', '-keyAlias', 'testKey', '–password', 'testPWD', '–passwordVerify', 'testPWD', '–keyReferenceSaveConfig', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createKeyReference {-interactive}` • Using Jython string: `AdminTask.createKeyReference ('[-interactive]')` • Using Jython list: `AdminTask.createKeyReference (['-interactive'])`

Table 18. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			• Returns: The configuration object name of the key reference scope object that you created.	
deleteKeyReference	The **deleteKeyReference** command deletes a key reference object from the key set object in the configuration.	None	• Parameters: **- keySetName** The name that uniquely identifies the key set to which the key reference belongs. (String, required) **- keySetScope** The management scope of the key set. (String, optional) **- keyAlias** The alias name that identifies the key for the key set that you specify. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteKeyReference { -keySetName tesKeySet –keyAlias testKey }` • Using Jython string: `AdminTask.deleteKeyReference ('[-keySetName tesKeySet –keyAlias testKey]')` • Using Jython list: `AdminTask.deleteKeyReference (['-keySetName', 'testKeySet', '–keyAlias', 'testKey'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteKeyReference {-interactive}` • Using Jython string: `AdminTask.deleteKeyReference ('[-interactive]')` • Using Jython list: `AdminTask.deleteKeyReference (['-interactive'])`

Table 18. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getKeyReference	The **getKeyReference** command displays the setting of a key reference object.	None	• Parameters: **- keySetName** The name that uniquely identifies the key set to which the key reference belongs. (String, required) **- keySetScope** The management scope of the key set. (String, optional) **- keyAlias** The alias name that identifies the key for the key set that you specify. (String, required) • Returns: The settings of the key reference object.	**Batch mode example usage:** • Using Jacl: `$AdminTask getKeyReference { -keySetName testKeySet –keyAlias testKey }` • Using Jython string: `AdminTask.getKeyReference ('[-keySetName testKeySet –keyAlias testKey]')` • Using Jython list: `AdminTask.getKeyReference (['-keySetName', 'testKeySet', '–keyAlias', 'testKey'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask getKeyReference {-interactive}` • Using Jython string: `AdminTask.getKeyReference ('[-interactive]')` • Using Jython list: `AdminTask.getKeyReference (['-interactive'])`

Table 18. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listKeyReferences	The **listKeyReferences** command lists the key references for a particular key set in the configuration.	None	• Parameters: **- keySetName** The name that uniquely identifies the key set to which the key reference belongs. (String, required) **- keySetScope** The management scope of the key set. (String, optional) • Returns: The configuration object name of the key reference scope object that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask listKeyRef ereces { -keySetName testKeySet}` • Using Jython string: `AdminTask.listKeyRefere ces ('[-keySetName test KeySet]')` • Using Jython list: `AdminTask.listKeyRefere ces (['-keySetName', 'testKeySet'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listKeyRef erences {-interactive}` • Using Jython string: `AdminTask.listKeyRefer ences ('[-interactive]')` • Using Jython list: `AdminTask.listKeyRefer ences (['-interactive'])`

Commands for the securityEnablement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the securityEnablement group of the AdminTask object:

Table 19.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
GlobalSettings		None	• Parameters: **- secureApps** Enables application security. The default value is `true`. (Boolean, required) **- secureLocalResources** Enables Java 2 Security. The default value is `true`. (Boolean, required) **- ltpaPassword** Defines the LTPA password. Valid values include the password used by LTPA. (String, required) **- userRegistryType** Defines the type of user registry to use. Valid values include: `Embedded`, `LDAP`, `LocalOS`, or `Custom`. (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask GlobalSettings {-interactive}` • Using Jython string: `AdminTask.GlobalSettings ('[-interactive]')` • Using Jython list: `AdminTask.GlobalSettings (['-interactive'])`

Table 19. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
CustomRegistry	The **CustomRegistry** command checks that the specified name for the primary administrator does not already exist in the custom registry.	None	• Parameters: **- primaryAdminName** Represents a name of the primary administrator. The value must be a valid administrator name. (String, required) **- serverUserName** Represents the user name used to connect to internal security routines. The value must be a valid user name. (String, required) **- customRegistryClass** Represents the class name of the custom user registry implementation of the UserRegistry. The value must be a valid class name for a custom registry. (String, required) **- customProperties** Custom user registry properties. An array or name-value properties. (String, required) **- customName** The custom user registry property name. (String, required) **- customValue** The custom user registry property value that is associated with the property name. (String, required) • Returns: A value of `true` if the primary administrator does not already exist in the custom registry. Otherwise, returns a value of `false`.	**Interactive mode example usage:** • Using Jacl: `$AdminTask CustomRegistry {-interactive}` • Using Jython string: `AdminTask.CustomRegistry ('[-interactive]')` • Using Jython list: `AdminTask.CustomRegistry (['-interactive'])`

Table 19. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
EmbeddedRegistry	The **EmbeddedRegistry** command checks that the specified name for the primary administrator does not already exist in the WIM file-based registry. Returns true if it does not; false, otherwise.	None	• Parameters: **- primaryAdminName** This string represents a name of the primary administrator. The value must be a valid administrator name. (String, required) **- adminPassword** The password associated with the primary administrator. The value must be a valid administrator password. (String, required) **- serverUserName** This string represents the user name used to connect to internal security routines. The value must be a valid user name. (String, required) • Returns: A value of `true` if the specified name for the primary administrator does not already exist in the WIM file-based registry. Otherwise, returns a value of `false`.	**Interactive mode example usage:** • Using Jacl: `$AdminTask EmbeddedRegistry {-interactive}` • Using Jython string: `AdminTask.EmbeddedRegistry ('[-interactive]')` • Using Jython list: `AdminTask.EmbeddedRegistry (['-interactive'])`

Table 19. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
LDAPRegistry	The **LDAPRegistry** command checks, after all the inputs for the LDAP registry have been defined, that a connection can be made successfully to the LDAP server. An SSL connection test to LDAP is also supported using this command.	None	• Parameters: **- ldapServerType** The type of LDAP user registry to be connected to WebSphere Application Server. Valid values include: `IBM Tivoli Directory Server`, `SecureWay`, `Sun ONE`, `Domino`, `Active Directory`, `eDirectory`, `Custom`. (String, required) **- ldapHostname** The host name for the LDAP server. (String, required) **- ldapPort** The port to connect to the LDAP server. (Integer, required) **- ldapBaseDN** The base distinguished name of the directory service, the starting point for searches. (String, required) **- ldapBindDN** The bind distinguished name, used to bind to the directory server. (String, required)	**Interactive mode example usage:** • Using Jacl: `$AdminTask LDAPRegistry {-interactive}` • Using Jython string: `AdminTask.LDAPRegistry ('[-interactive]')` • Using Jython list: `AdminTask.LDAPRegistry (['-interactive'])`

Table 19. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- ldapBindPassword The password for the application server, used to bind to the directory service (String, required) **- ldapServerUserName** The user ID that is used to run WebSphere Application Server for security purposes. (String, required) • Returns: A value of `true` if the connection was successful. Otherwise, returns a value of `false`.	
LocalOSRegistry	The **LocalOSRegistry** command checks that the specified name for the primary administrator does not already exist in the LocalOS registry.	None	• Parameters: **- primaryAdminName** This string represents a name of the primary administrator. The value must be a valid administrator name. (String, required) **- serverUserName** This string represents the user name used to connect to internal security routines. The value must be a valid user name. (String, required) • Returns: A value of `true` if the specified name for the primary administrator does not already exist in the LocalOS registry. Otherwise, returns a value of `false`.	**Interactive mode example usage:** • Using Jacl: `$AdminTask LocalOSRegistry {-interactive}` • Using Jython string: `AdminTask.LocalOSRegistry ('[-interactive]')` • Using Jython list: `AdminTask.LocalOSRegistry (['-interactive'])`

Table 19. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
CurrentSettings	The **CurrentSettings** command returns a string that represents all of the existing settings set by the wizard that will be read from the workspace instead of from the configuration repository of the `security.xml` file.	None	• Parameters: **-currentWizardSettings** A properties object that contains all the current settings selected through the wizard. (Properties, required) • Returns: A string of security settings as name-value pairs.	**Interactive mode example usage:** • Using Jacl: `$AdminTask CurrentSettings {-interactive}` • Using Jython string: `AdminTask.CurrentSettings ('[-interactive]')` • Using Jython list: `AdminTask.CurrentSettings (['-interactive'])`

Commands for the CertificateRequestCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the CertificateRequestCommands group of the AdminTask object:

Table 20.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createCertificate Request	The **createCertificate Request** command creates a certificate request that is associated with a particular key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- certificateVersion** The certificate version. (String, required) **- certificateSize** (Integer, required) **- certificateCommonName** (String, required) **- certificateOrganization** (String, optional) **- certificateOrganizationUnit** (String, optional) **- certificateLocality** (String, optional) **- certificateState** The state code for the certificate. (String, optional) **- certificateZip** The zip code for the certificate. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createCertificateRequest {-keyStoreName testKeyStore -certificateAlias certReq -certificateSize 1024 -certificate CommonName localhost -certificate Organization testing -certificate RequestFilePath c:\temp\testCertReq.arm}` • Using Jython string: `AdminTask.createCertificateRequest ('[-keyStoreName testKeyStore -certificateAlias certReq -certificateSize 1024 -certificate CommonName localhost -certificate Organization testing -certificate RequestFilePath c:\temp\testCertReq.arm]` • Using Jython list: `AdminTask.createCertificateRequest (['-keyStoreName', 'testKeyStore', '-certificateAlias', 'certReq', '-certificateSize', '1024', '-certificateCommonName', 'localhost', '-certificateOrganization', 'testing', '-certificateRequestFilePath', 'c:\temp\testCert Req.arm'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createCertific ateRequest {-interactive}` • Using Jython string: `AdminTask.createCertific ateRequest ('[-interactive]')` • Using Jython list: `AdminTask.createCertifica teRequest (['-interactive'])`

Table 20. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- certificateCountry The country for the certificate. (String, optional) **- certificateValidDays** The amount of time in days for which the certificate is valid. (Integer, optional) **- certificateRequestFilePath** The file location of the certificate request that can be sent to a certificate authority. (String, required) • Returns: The configuration object name of the key store object that you created.	
deleteCertificate Request	The **deleteCertificate Request** command deletes a certificate request from a key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCertificateRequest {-interactive}` • Using Jython string: `AdminTask.deleteCertificateRequest ('[-interactive]')` • Using Jython list: `AdminTask.deleteCertificateRequest (['-interactive'])`

Table 20. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
extractCertificate Request	The **extractCertificate Request**command extracts a certificate request to a file.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- certificateRequestFilePath** The file location of the certificate request that can be sent to a certificate authority. (String, required) • Returns: A certificate request file is created that contains the extracted certificate.	**Interactive mode example usage:** • Using Jacl: `$AdminTask extractCertificateRequest {-interactive}` • Using Jython string: `AdminTask.extractCertificateRequest ('[-interactive]')` • Using Jython list: `AdminTask.extractCertificateRequest (['-interactive'])`

Table 20. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getCertificateRequest	The **getCertificateRequest** command obtains information about a particular certificate request in a key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) • Returns: Information about the certificate request.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getCertificate Request {-interactive}` • Using Jython string: `AdminTask.getCertificate Request ('[-interactive]')` • Using Jython list: `AdminTask.getCertificate Request (['-interactive'])`
listCertificateRequest	The **listCertificateRequest** command lists all the certificate requests associated with a particular key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) • Returns: An attribute list for each certificate request in a key store.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listCertifica teRequest {-interactive}` • Using Jython string: `AdminTask.listCertifica teRequest ('[-interactive]')` • Using Jython list: `AdminTask.listCertificat eRequest (['-interactive'])`

Commands for the SignerCertificateCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the SignerCertificateCommands group of the AdminTask object:

Table 21.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
addSignerCertificate	The **addSignerCertificate** command adds a signer certificate from a certificate file to a key store.	None	• Parameters: **- keyStoreName** The name of the key store where the signer certificate is located. (String, required) **- keyStoreScope** The management scope of the key store. (String, optional) **- certificateFilePath** The full path name of the file that contains the signer certificate. (String, required) **- certificateAlias** The alias name of the signer certificate in the key store. (String, required) **- base64Encoded** Set the value of this parameter to `true` if the certificate is ascii base 64 encoded. Set the value of this parameter to `false` if the certificate is binary. (String, required) • Returns: The configuration object name of the key store object that you created.	**Interactive mode example usage:** • Using Jacl: `$AdminTask addSignerCert ificate {-interactive}` • Using Jython string: `AdminTask.addSignerCert ificate ('[-interactive]')` • Using Jython list: `AdminTask.addSignerCert ificate (['-interactive'])`

Table 21. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteSignerCertificate	The **deleteSignerCertificate** command deletes a signer certificate from a key store.	None	• Parameters: **- keyStoreName** The name of the key store where the signer certificate is located. (String, required) **- keyStoreScope** The management scope of the key store. (String, optional) **- certificateAlias** The alias name of the signer certificate in the key store. (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSignerCertificate {-interactive}` • Using Jython string: `AdminTask.deleteSignerCertificate ('[-interactive]')` • Using Jython list: `AdminTask.deleteSignerCertificate (['-interactive'])`

Table 21. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
extractSignerCertificate	The **extractSignerCertificate** command extracts a signer certificate from a key store to a file.	None	• Parameters: **- keyStoreName** The name of the key store where the signer certificate is located. (String, required) **- keyStoreScope** The management scope of the key store. (String, optional) **- certificateAlias** The alias name of the signer certificate in the key store. (String, required) **- certificateFilePath** The full path name of the file that contains the signer certificate. (String, required) **- base64Encoded** Set the value of this parameter to `true` if the certificate is ascii base 64 encoded. Set the value of this parameter to `false` if the certificate is binary. (String, required) • Returns: The certificate file is created and contains the signer certificate.	**Interactive mode example usage:** • Using Jacl: `$AdminTask extractSigner Certificate {-interactive}` • Using Jython string: `AdminTask.extractSignerC ertificate ('[-interactive]')` • Using Jython list: `AdminTask.extractSignerCe rtificate (['-interactive'])`

Table 21. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getSignerCertificate	The **getSignerCertificate** command obtains information about a signer certificate from a key store.	None	• Parameters: **- keyStoreName** The name of the key store where the signer certificate is located. (String, required) **- keyStoreScope** The management scope of the key store. (String, optional) **- certificateAlias** The alias name of the signer certificate in the key store. (String, required) • Returns: Information about a signer certificate.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getSignerCert ificate {-interactive}` • Using Jython string: `AdminTask.getSignerCert ificate ('[-interactive]')` • Using Jython list: `AdminTask.getSignerCert ificate (['-interactive'])`
listSignerCertificates	The **listSignerCertificates** command lists all signer certificates in a particular key store.	None	• Parameters: **- keyStoreName** The name of the key store where the signer certificate is located. (String, required) **- keyStoreScope** The management scope of the key store. (String, optional) • Returns: A list of signer certificate aliases.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listSignerCer tificates {-interactive}` • Using Jython string: `AdminTask.listSignerCer tificates ('[-interactive]')` • Using Jython list: `AdminTask.listSignerCer tificates (['-interactive'])`

Table 21. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
retrieveSignerFromPort	The **retrieveSignerFromPort** command retrieves a signer from a remote host and stores the signer in a key store.	None	• Parameters: **- host** The host name of the system from where the signer certificate will be retrieved. (String, required) **- port** The port of the remote system from where the signer certificate will be retrieved. (Integer, required) **- keyStoreName** The name of the key store where the signer certificate is located. (String, required) **- keyStoreScope** The management scope of the key store. (String, required) **- sslConfigName** The name of the SSL configuration object. (String, optional) **- sslConfigScopeName** The management scope where the SSL configuration object is located. (String, optional) • Returns: The signer certificate is created in the key store file.	**Batch mode example usage:** • Using Jacl: `$AdminTask retrieveSigner FromPort {-host serverHost -port 443 -keyStoreName testKeyStore -certificate Alias serverHostSigner}` • Using Jython string: `AdminTask.retrieveSigner FromPort ('[-host server Host -port 443 -keyStore Name testKeyStore -certi ficateAlias serverHost Signer]')` • Using Jython list: `AdminTask.retrieveSigner FromPort (['-host', 'serverHost', '-port', '443', '-keyStoreName', 'testKeyStore', '-certi ficateAlias', 'serverHost Signer'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask retrieveSigner FromPort {-interactive}` • Using Jython string: `AdminTask.retrieveSigner FromPort ('[-interactive]')` • Using Jython list: `AdminTask.retrieveSigner FromPort (['-interactive'])`

Table 21. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
retrieveSigner InfoFromPort	The **retrieveSigner InfoFromPort** command retrieves signer information from a port on a remote host.	None	• Parameters: **- host** The host name of the system from where the signer certificate will be retrieved. (String, required) **- port** The port of the remote system from where the signer certificate will be retrieved. (Integer, required) **- sslConfigName** The name of the SSL configuration object. (String, optional) **- sslConfigScopeName** The management scope where the SSL configuration object is located. (String, optional) • Returns: Information about the signer certificate from the remote host port.	**Interactive mode example usage:** • Using Jacl: `$AdminTask retrieveSigner InfoFromPort {-interactive}` • Using Jython string: `AdminTask.retrieveSigner InfoFromPort ('[-interactive]')` • Using Jython list: `AdminTask.retrieveSigner InfoFromPort (['-interactive'])`

Commands for the PersonalCertificateCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the personalCertificateCommands group of the AdminTask object:

Table 22.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createSelfSigned Certificate	The **createSelfSigned Certificate** command creates a personal certificate in a key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- certificateVersion** The version of the certificate. (String, required) **- certificateSize** The size of the certificate. (Integer, required) **- certificateCommonName** The common name of the certificate. (String, required) **- certificateOrganization** The organization of the certificate. (String, optional) **- certificateOrganizationUnit** The organizational unit of the certificate. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask createSelfSignedCertificate {-keyStoreName testKeyStore -certificateAlias default -certificateCommonName localhost -certificateOrganization ibm}` • Using Jython string: `AdminTask.createSelfSignedCertificate ('[-keyStoreName testKeyStore -certificateAlias default -certificateCommonName localhost -certificateOrganization ibm]')` • Using Jython list: `AdminTask.createSelfSignedCertificate (['-keyStoreName', 'testKeyStore', '-certificateAlias', 'default', '-certificateCommonName', 'localhost', '-certificateOrganization', 'ibm'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createSelfSigned Certificate {-interactive}` • Using Jython string: `AdminTask.createSelfSigned Certificate ('[-interactive]')` • Using Jython list: `AdminTask.createSelfSigned Certificate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
			- certificateLocality The locality of the certificate. (String, optional) **- certificateState** The state of the certificate. (String, optional) **- certificateZip** The zip code of the certificate. (String, optional) **- certificateCountry** The country of the certificate. (String, optional) **- certificateValidDays** The amount of time in days for which the certificate is valid. (Integer, optional) • Returns: None	
deleteCertificate	The **deleteCertificate** command deletes a personal certificate from a key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteCertificate {-interactive}` • Using Jython string: `AdminTask.deleteCertificate ('[-interactive]')` • Using Jython list: `AdminTask.deleteCertificate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
exportCertificate	The **exportCertificate** command exports a personal certificate from one key store to another.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- keyStorePassword** The password to the key store. (String, required) **- keyFilePath** The full path to a key store file that is located in a file system. The store from where a certificate will be imported or exported. (String, required) **- keyFilePassword** The password to the key store file. (String, required) **- keyFileType** The type of the key file. (String, required) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- aliasInKeyStore** (String, optional) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask exportCertificate {-interactive}` • Using Jython string: `AdminTask.exportCertificate ('[-interactive]')` • Using Jython list: `AdminTask.exportCertificate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
extractCertificate	The **extractCertificate** command extracts the signer part of a personal certificate to a file.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- certificateRequestFilePath** The full path of the request file that contains the certificate. (String, required) **- base64Encoded** Set the value of this parameter to `true` if the certificate is ascii base 64 encoded. Set the value of this parameter to `false` if the certificate is binary. (Boolean, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask extractCertificate {-keyStoreName testKeyStore -certificateFilePath c:\temp\CertFile.arm -certificateAlias testCertificate}` • Using Jython string: `AdminTask.extractCertificate ('[-keyStoreName testKeyStore -certificateFilePath c:\temp\CertFile.arm -certificateAlias testCertificate]')` • Using Jython list: `AdminTask.extractCertificate (['-keyStoreName', 'testKeyStore', '-certificateFilePath', 'c:\temp\CertFile.arm', '-certificateAlias', 'testCertificate'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask extractCertificate {-interactive}` • Using Jython string: `AdminTask.extractCertificate ('[-interactive]')` • Using Jython list: `AdminTask.extractCertificate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getCertificate	The **getCertificate** command obtains information about a particular personal certificate in a key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) • Returns: Information about the certificate request.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getCertificate {-interactive}` • Using Jython string: `AdminTask.getCertificate ('[-interactive]')` • Using Jython list: `AdminTask.getCertificate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
importCertificate	The **importCertificate** command imports a personal certificate from a key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- keyFilePath** The full path to a key store file that is located in a file system. The store from where a certificate will be imported or exported. (String, required) **- keyFilePassword** The password to the key store file. (String, required) **- keyFileType** The type of the key file. (String, required) **- certificateAliasFromKeyFile** The certificate alias in the key file from which the certificate is being imported. (String, required) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask importCertificate {-interactive}` • Using Jython string: `AdminTask.importCertificate ('[-interactive]')` • Using Jython list: `AdminTask.importCertificate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
listPersonalCertificates	The **listPersonalCertificates** command lists the personal certificates in a particular key store.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) • Returns: A list of attributes for each personal certificate in a key store.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listPersonalCertificates {-interactive}` • Using Jython string: `AdminTask.listPersonalCertificates ('[-interactive]')` • Using Jython list: `AdminTask.listPersonalCertificates (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
receiveCertificate	The **receiveCertificate** command receives a signer certificate from a file to a personal certificate.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- certificateFilePath** The full path of the file that contains the certificate. (String, required) **- base64Encoded** Set the value of this parameter to `true` if the certificate is ascii base 64 encoded. Set the value of this parameter to `false` if the certificate is binary. (Boolean, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask receiveCertificate {-keyStoreName testKeyStore -certificateFilePath c:\temp\ CertFile.arm}` • Using Jython string: `AdminTask.receiveCertificate ('[-keyStoreName testKeyStore -certificateFilePath c:\temp\ CertFile.arm]')` • Using Jython list: `AdminTask.receiveCertificate (['-keyStoreName', 'testKeyStore', '-certificateFilePath', 'c:\ temp\CertFile.arm'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask receiveCertif icate {-interactive}` • Using Jython string: `AdminTask.receiveCerti ficate ('[-interactive]')` • Using Jython list: `AdminTask.receiveCertif icate (['-interactive'])`

Table 22. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
replaceCertificate	The **replaceCertificate** command replaces a personal certificate with a new one. Replaces all signer certificates from the personal certificate.	None	• Parameters: **- keyStoreName** The name that uniquely identifies the key store configuration object. (String, required) **- keyStoreScope** The scope name of the key store. (String, optional) **- certificateAlias** The name that uniquely identifies the certificate request in a key store. (String, required) **- replacementCertificateAlias** The alias of the certificate that is used to replace a different certificate. (String, required) **- deleteOldCert** Set the value of this parameter to `true` if you want to delete the old signer certificates during certificate replacement. Otherwise, set the value of this parameter to `false`. (Boolean, optional) **- deleteOldSigners** Set the value of this parameter to `true` if you want to delete the old certificates during certificate replacement. Otherwise, set the value of this parameter to `false`. (Boolean, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask replaceCertificate {-keyStoreName testKeyStore -certificateAlias default -replacementCertificateAlias replaceCert -deleteOldCert true -deleteOldSigners true}` • Using Jython string: `AdminTask.replaceCertificate ('[-keyStoreName testKeyStore -certificateAlias default -replacementCertificateAlias replaceCert -deleteOldCert true -deleteOldSigners true]')` • Using Jython list: `AdminTask.replaceCertificate (['-keyStoreName', 'testKeyStore', '-certificateAlias', 'default', '-replacementCertificateAlias', 'replaceCert', '-deleteOldCert', 'true', '-deleteOldSigners', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask replaceCertificate {-interactive}` • Using Jython string: `AdminTask.replaceCertificate ('[-interactive]')` • Using Jython list: `AdminTask.replaceCertificate (['-interactive'])`

Commands for the SPNEGO TAI group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the SPNEGO TAI group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
addSpnegoTAIProperties	The **addSpnegoTAIProperties** command adds properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.	None	• Parameters: **- spnId** This is the SPN identifier for the group of custom properties that are to be defined with this command. If you do not specify this parameter, an unused SPN identifier is assigned. (String, optional) **- host** Specifies the host name portion in the SPN used by the SPNEGO TAI to establish a Kerberos secure context. (String, required) **- filter** Defines the filtering criteria used by the class specified with the above attribute. If no filter is specified, all HTTP requests are subject to SPNEGO authentication. (String, optional) **- filterClass** Specifies the name of the Java class used by the SPNEGO TAI to select which HTTP requests will be subject to SPNEGO authentication. If no filter class is specified, the default filter class, com.ibm.ws.security.spnego.HTTPHeaderFilter, is used. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask addSpnegoTAIProperties -host myhost.ibm.com -filter user-agent%=IE 6` • Using Jython string: `AdminTask.addSpnegoTAIProperties ('[-host myhost.ibm.com -filter user-agent%=IE 6]')` • Using Jython list: `AdminTask.addSpnegoTAIProperties (['-host', 'myhost.ibm.com', '-filter', 'user-agent%=IE', '6'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addSpnegoTAIProperties -interactive` • Using Jython string: `AdminTask.addSpnegoTAIProperties ('[-interactive]')` • Using Jython list: `AdminTask.addSpnegoTAIProperties ['-interactive'])`

			- noSpnegoPage Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application if it does not support SPNEGO authentication. (String, optional). If you do not specify the noSpnegoPage attribute then the default is used: `"<html><head><title> SPNEGO authentication is not supported. </title></head>" + "<body>SPNEGO authe ntication is not supported on this client.</body> </html>";` **- ntlmTokenPage** Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application when the SPNEGO token received by the interceptor after the challenge-response handshake contains a NT LAN manager (NTLM) token instead of the expected SPNEGO token. (String, optional). If you do not specify the ntlmTokenPage attribute then the default is used: `"<html><head><title> An NTLM Token was received.</title> </head>" + "<body>Your bro wser configuration is correct, but you have not logged into a supported Windows Domain." + "<p>Please login to the application using the normal login page.</html>";`	

			- trimUserName Specifies whether (`true`) or not (`false`) the SPNEGO TAI is to remove the suffix of the principal user name, starting from the `@` that precedes the Kerberos realm name. If this attribute is set to true, the suffix of the principal user name is removed. If this attribute is set to `false`, the suffix of the principal name is retained. The default value used is `true`. (String, optional) • Returns: None	

createKrb ConfigFile	The **createKrb ConfigFile** command creates the Kerberos configuration file for use with the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.	None	• Parameters: **- krbPath** Provides the fully qualified file system location of the Kerberos configuration (krb5.ini or krb5.conf) file. (String, required) **- realm** Provides the Kerberos realm name. The value of this attribute is used by the SPNEGO TAI to form the Kerberos service principal name for each of the hosts specified with the property com.ibm.ws.security.spnego.SPN<id>.hostname (String, required) **- kdcHost** Provides the host name of the Kerberos Key Distribution Center (KDC). (String, required) **- kdcPort** Provides the port number of the KDC. The default value, if not specified, is 88. (String, optional) **- dns** Provides the default domain name service (DNS) that is used to produce a fully qualified host name. (String, required) **- keytabPath** Provides the file system location of the Kerberos keytab file. (String, required) **- encryption** Identifies the list of supported encryption types, separated by a space. The specified value is used for the default_tkt_enctypes and default_tgs_enctypes. The default encryption types, if not specified, are des-cbc-md5 and rc4-hmac. (String, optional) • Returns: None	**Interactive mode example usage:** • Using Jacl: `$AdminTask createKrbCo nfigFile -interactive` • Using Jython string: `AdminTask.createKrbCon figFile ('[-interactive]')` • Using Jython list: `AdminTask.createKrbCon figFile ['-interactive'])`

deleteSpnegoTAIProperties	The **deleteSpnegoTAIProperties** command deletes properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.	None	• Parameters: **- spnId** The SPN identifier for the group of custom properties that are to be deleted with this command. If you do not specify this parameter, all SPNEGO TAI custom properties are deleted. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteSpnegoTAIProperties {-spnId 2}` • Using Jython string: `AdminTask.deleteSpnegoTAIProperties ('[-spnId 2]')` • Using Jython list: `AdminTask.deleteSpnegoTAIProperties (['-spnId', '2'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteSpnegoTAIProperties -interactive` • Using Jython string: `AdminTask.deleteSpnegoTAIProperties ('[-interactive]')` • Using Jython list: `AdminTask.deleteSpnegoTAIProperties ['-interactive'])`

modifySpnegoTAIProperties	The **modifySpnegoTAIProperties** command modifies the properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.	None	• Parameters: **- spnId** The SPN identifier for the group of custom properties that are to be defined with this command. (String, required) **- host** Specifies the host name portion in the SPN used by the SPNEGO TAI to establish a Kerberos secure context. (String, optional) **- filter** Defines the filtering criteria used by the class specified with the above attribute. (String, optional) **- filterClass** Specifies the name of the Java class used by the SPNEGO TAI to select which HTTP requests will be subject to SPNEGO authentication. If no class is specified, all HTTP requests will be subject to SPNEGO authentication. (String, optional) **- noSpnegoPage** Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application if it does not support SPNEGO authentication. (String, optional)	**Batch mode example usage:** • Using Jacl: `$AdminTask modifySpnegoTAI PROPERTIES -spnId 1 -filter host==myhost.company.com` • Using Jython string: `AdminTask.modifySpnegoTAI PROPERTIES ('[-spnId 1 -filter host==myhost.company.com]')` • Using Jython list: `AdminTask.modifySpnegoTAI PROPERTIES (['-spnId', '1', '-filter', 'host==myhost.company.com'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask modifySpnegoTAI Properties -interactive` • Using Jython string: `AdminTask.modifySpnegoTAI Properties ('[-interactive]')` • Using Jython list: `AdminTask.modifySpnegoTAI Properties ['-interactive'])`

			- ntlmTokenPage Specifies the URL of a resource that contains the content the SPNEGO TAI will include in the HTTP response to be displayed by the (browser) client application when the SPNEGO token received by the interceptor after the challenge-response handshake contains a NT LAN manager (NTLM) token instead of the expected SPNEGO token. (String, optional) **- trimUserName** Specifies whether (`true`) or not (`false`) the SPNEGO TAI is to remove the suffix of the principal user name, starting from the ″@″ that precedes the Kerberos realm name. If this attribute is set to true, the suffix of the principal user name is removed. If this attribute is set to `false`, the suffix of the principal name is retained. The default value used is `true`. (String, optional) • Returns: None	

showSpnegoTAIProperties	The **showSpnegoTAIProperties** command displays the properties in the configuration of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) trust association interceptor (TAI) for WebSphere Application Server.	None	• Parameters: **- spnId** The service principal name (SPN) identifier for the group of custom properties that are to be displayed with this command. If you do not specify this parameter, all SPNEGO TAI custom properties are displayed. (String, optional) • Returns: A list of properties.	**Batch mode example usage:** • Using Jacl: `$AdminTask showSpnegoTAI Properties -spnId 1` • Using Jython string: `AdminTask.showSpnegoTAI Properties ('[-spnId 1]')` • Using Jython list: `AdminTask.showSpnegoTAI Properties (['-spnId', '1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask showSpnegoTAI Properties -interactive` • Using Jython string: `AdminTask.showSpnegoTAI Properties ('[-interact ive]')` • Using Jython list: `AdminTask.showSpnegoTAI Properties ['-interact ive'])`

Commands for the AuthorizationGroupCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the AuthorizationGroupCommands group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

addResourceToAuthorizationGroup	The **addResourceToAuthorizationGroup** command adds a resource instance to an existing authorization group. A resource instance cannot belong to more than one authorization group.	None	• Parameters: **- authorizationGroupName** The name of the authorization group. (String, required) **- resourceName** The name of the resource instance that you want to add to an authorization group. (String, required) The resourceName parameter should be in the following format: `ResourceType= ResourceName` where `ResourceType` is one of the following values: Application, Server, ServerCluster, Node, NodeGroup `ResourceName` is the name of the resource instance, for example, `server1`. The following are example uses of the resourceName parameter: – `Node=node1:Se rver=server1` This example uniquely identifies `server1`. `node1` is required if another `server1` exists on a different node. – `Application=app1` • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask addResourceToAuthorizationGroup {-authorizationGroupName groupName -resourceName Application=app1}` • Using Jython string: `AdminTask.addResourceToAuthorizationGroup('[-authorizationGroupName groupName -resourceName Application=app1]')` • Using Jython list: `AdminTask.addResourceToAuthorizationGroup(['-authorizationGroupName', 'groupName', '-resourceName', 'Application=app1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask addResourceToAuthorizationGroup {-interactive}` • Using Jython string: `AdminTask.addResourceToAuthorizationGroup ('[-interactive]')` • Using Jython list: `AdminTask.addResourceToAuthorizationGroup (['-interactive'])`

createAuthorizationGroup	The **createAuthorizationGroup** command creates a new authoirzation group. When you create a new authorization group, no members are associated with it. Also, no user to administrative role mapping for the authorization table is associated with the authorization group.	None	• Parameters: **- authorizationGroupName** The name of the authorization group that you want to create. (String, required) • Returns: The configuration ID of the authorization group that you created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createAuthorizationGroup {-authorizationGroupName groupName}` • Using Jython string: `AdminTask.createAuthorizationGroup('[-authorizationGroupName groupName]')` • Using Jython list: `AdminTask.createAuthorizationGroup(['-authorizationGroupName', 'groupName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createAuthorizationGroup -interactive` • Using Jython string: `AdminTask.createAuthorizationGroup ('[-interactive]')` • Using Jython list: `AdminTask.createAuthorizationGroup (['-interactive'])`

deleteAuthorizationGroup	The deletes an existing authorization group. When you delete an authorization group, the authorization table that corresponds is also deleted.	None	• Parameters: **- authorizationGroup Name** The name of the authorization group that you want to delete. (String, required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteAuthorizationGroup {-authorizationGroupName groupName}` • Using Jython string: `AdminTask.deleteAuthorizationGroup('[-authorizationGroupName groupName]')` • Using Jython list: `AdminTask.deleteAuthorizationGroup(['-authorizationGroupName', 'groupName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteAuthorizationGroup {-interactive}` • Using Jython string: `AdminTask.deleteAuthorizationGroup ('[-interactive]')` • Using Jython list: `AdminTask.deleteAuthorizationGroup (['-interactive'])`
	The command lists the existing authorization groups.	None	• Parameters: None • Returns: A list of short names of all existing authorization groups. (String [])	**Batch mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroups` • Using Jython: `AdminTask.listAuthorizationGroups()` **Interactive mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroups {-interactive}` • Using Jython string: `AdminTask.listAuthorizationGroups ('[-interactive]')` • Using Jython list: `AdminTask.listAuthorizationGroups (['-interactive'])`

listAuthorizationGroupsForGroupID	The **listAuthorizationGroupsForGroupID** command lists all of the authorization groups to which a given user group has access. This command lists the authorization groups and the granted roles for each authorization group. The group ID can be a short name or a fully qualified domain name if the LDAP user registry is being used. This command will list `cell` as a group if the user has cell level access.	None	• Parameters: **- groupid** The ID of the user group. (String, required) • Returns: The map of the authorization group and granted roles. (Map[String=String[]])	**Batch mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroupsForGroupID {-groupid userGroupName}` • Using Jython string: `AdminTask.listAuthorizationGroupsForGroupID('[-groupid userGroupName]')` • Using Jython list: `AdminTask.listAuthorizationGroupsForGroupID(['-groupid', 'userGroupName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroupsForGroupID {-interactive}` • Using Jython string: `AdminTask.listAuthorizationGroupsForGroupID ('[-interactive]')` • Using Jython list: `AdminTask.listAuthorizationGroupsForGroupID (['-interactive'])`

listAuthorizationGroupsForUser ID	The **listAuthorizationGroupsForUser ID** command lists all of the authorization groups to which a given user has access. This command lists the authorization groups and the granted roles for each authorization group. The user ID and the group ID can be a short name or a fully qualified domain name if the LDAP user registry is being used. This command will list `cell` as a group if the user has cell level access.	None	• Parameters: **- userid** The ID of the user. (String, required) • Returns: The map of the authorization group and granted roles. (Map[String=String[]])	**Batch mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroupsForUserID{-userid userName}` • Using Jython string: `AdminTask.listAuthorizationGroupsForUserID('[-userid userName]')` • Using Jython list: `AdminTask.listAuthorizationGroupsForUserID(['-userid', 'userName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroupsForUserID {-interactive}` • Using Jython string: `AdminTask.listAuthorizationGroupsForUserID ('[-interactive]')` • Using Jython list: `AdminTask.listAuthorizationGroupsForUserID (['-interactive'])`

listAuthorizationGroupsOfResource	The **listAuthorizationGroupsOfResource** command lists authorization groups for a given resource. If the value of the traverseContainedObjects parameter is false, only the authorization group of the resource is returned. If the value of the traverseContainedObjects parameter is true, it returns the authorization group of the resource and the authorization groups of all the parent resources in the containment tree.	None	• Parameters: **- resourceName** The name of the resource. (String, required) The resourceName parameter must be in the following format: `ResourceType=ResourceName` where `ResourceType` can be any one of the following values: Application, Server, ServerCluster, Node, or NodeGroup. `ResourceName` is the name of the resource instance, for example, server1. The following are examples of the resourceName parameter: `Node=node1: Server=server` This example uniquely identifies *server1*. The name of the node is required if a server on a different node uses the same server name. `Application=app1`	**Batch mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroupsOfResource {-resourceName Application=app1}` • Using Jython string: `AdminTask.listAuthorizationGroupsOfResource('[-resourceName Application=app1]')` • Using Jython list: `AdminTask.listAuthorizationGroupsOfResource(['-resourceName', 'Application=app1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listAuthorizationGroupsOfResource {-interactive}` • Using Jython string: `AdminTask.listAuthorizationGroupsOfResource ('[-interactive]')` • Using Jython list: `AdminTask.listAuthorizationGroupsOfResource (['-interactive'])`

			- traverseContained Resources Finds the authorization groups of all the parent resources by traversing the resource containment tree upwards. The default value is false. (Boolean, optional) • Returns: The short names of all of the authorization groups for which the resource belongs. If the you do not specify the traverseContainedResources parameter, the result of this command will only contain one value because a resource instance can only belong to one authorization group. (String[])	
listResources OfAuthoriz ationGroup	The **listResources OfAuthorizat ionGroup** command lists all of the resources within the given authorization group.	None	• Parameters: **- authorizationGroupName** The name of the authorization group. (String, required) • Returns: The configuration IDs of all of the resource instances within the authorization group. (String[])	**Batch mode example usage:** • Using Jacl: `$AdminTask listResourcesOf AuthorizationGroup {-autho rizationGroupName groupName}` • Using Jython string: `AdminTask.listResourcesOf AuthorizationGroup('[-auth orizationGroupName groupName]')` • Using Jython list: `AdminTask.listResourcesOfAu thorizationGroup(['-authori zationGroupName', 'groupName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listResourcesOfAu thorizationGroup {-interactive}` • Using Jython string: `AdminTask.listResourcesOfAu thorizationGroup ('[-intera ctive]')` • Using Jython list: `AdminTask.listResourcesOfA uthorizationGroup (['-inte ractive'])`

listResources ForGroupID	The **listResources ForGroupID** command lists all the objects that a given group has access to. This command lists the resources and the granted roles for each resource. The resources that this command returns include the resources from the authorization groups to which the user group is granted roles and the resources that are descendants of the resources with in authorization groups to which the user group is granted access to any role. The group ID can be a short name or fully qualified domain name if a LDAP user registry is used.	None	• Parameters: **- groupid** The ID of the user group. (String, required) • Returns: The map of the granted role and resources. (Map[String=String[]])	**Batch mode example usage:** • Using Jacl: `$AdminTask listResourcesFor GroupID {-groupid userGr oupName}` • Using Jython string: `AdminTask.listResourcesFor GroupID('[-groupid userGro upName]')` • Using Jython list: `AdminTask.listResourcesFor GroupID(['-groupid', 'user GroupName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listResourcesFor GroupID {-interactive}` • Using Jython string: `AdminTask.listResourcesFor GroupID ('[-interactive]')` • Using Jython list: `AdminTask.listResourcesFor GroupID (['-interactive'])`

listResources ForUserID	The **listResources ForUserID** command lists all the objects that a given user has access to. This command lists the resources and the granted roles for each resource. The resources that this command returns include the resources from the authorization groups to which the user is granted roles and the resources that are descendants of the resources with in authorization groups to which the user is granted access to any role. The user ID can be a short name or fully qualified domain name if a LDAP user registry is used.	None	• Parameters: **- userid** The ID of the user. (String, required). • Returns: The map of granted role and resources. (Map[String=String[]])	**Batch mode example usage:** • Using Jacl: `$AdminTask listResourcesFor UserID {-userid userName }` • Using Jython string: `AdminTask.listResourcesFor UserID('[-userid userName]')` • Using Jython list: `AdminTask.listResourcesFor UserID(['-userid', 'userName'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listResources ForUserID {-interactive}` • Using Jython string: `AdminTask.listResources ForUserID ('[-interactive]')` • Using Jython list: `AdminTask.listResources ForUserID (['-interactive'])` **Example output:** `{deployer=[], operator=[], administrator=[cells/IBM-LP1 6L31HVE8Cell07/clusters/C1|cluster.xml, cells/IBM-LP16L 31HVE8Cell07/nodes/IBM-LP16L 31HVE8Node05/servers/cm1|server.xml], monitor=[], configurator=[]}`

mapGroupsToAdminRole	The **mapGroupsToAdminRole** command maps group IDs to one or more administrative roles in an authorization group. The name of the authorization group that you provide determines which authorization table will be used. If you do not specify an authorization group name, the mapping is done to the cell level authorization table. The group ID can be a short name or a fully qualified domain name if the LDAP user registry is used.	None	• Parameters: **- authorizationGroupName** The name of the authorization group. If you do not specify this parameters, the cell level authorization group is assumed. (String, optional) **- roleName** The name of the administrative role. (String, required) **- groupids** The list of group IDs that will mapped to the administrative role. (String[], required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask mapGroupsToAdminRole {-authorizationGroupName group Name - roleName administrator -groupids group1}` • Using Jython string: `AdminTask.mapGroupsToAdminRole ('[-authorizationGroupName group Name -roleName administrator -groupids group1]')` • Using Jython list: `AdminTask.mapGroupsToAdminRole (['-authorizationGroupName', 'groupName', '-roleName', 'administrator', '-groupids', 'group1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask mapGroupsToAdm inRole {-interactive}` • Using Jython string: `AdminTask.mapGroupsToAdmin Role ('[-interactive]')` • Using Jython list: `AdminTask.mapGroupsToAdmin Role (['-interactive'])`

mapUsersTo AdminRole	The **mapUsersTo AdminRole** command 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. If you do not specify the name of the authorization group, the mapping is done to the cell level authorization table. The user ID can be a short name or fully qualified domain name in case LDAP user registry is used.	None	• Parameters: **- authorizationGroup Name** The name of the authorization group. If you do not specify this parameter, the cell level authorization group is assumed. (String, optional) **- roleName** The name of the administrative role. (String, required) **- userids** The list of user IDs that will be mapped to the administrative role (String[], required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask mapUsersToAdminRole {-authorizationGroupName group Name - roleName administrator -userids user1}` • Using Jython string: `AdminTask.mapUsersToAdminRole ('[-authorizationGroupName groupName -roleName administ rator -userids user1]')` • Using Jython list: `AdminTask.mapUsersToAdminRole (['-authorizationGroupName', 'groupName', '-roleName', 'administrator', '-userids', 'user1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask mapUsersToAdmin Role {-interactive}` • Using Jython string: `AdminTask.mapUsersToAdmin Role ('[-interactive]')` • Using Jython list: `AdminTask.mapUsersToAdmin Role (['-interactive'])`

removeGroupsFromAdminRole	The **removeGroupsFromAdminRole** command removes previously mapped group IDs from administrative roles in the authorization group. The name of the authorization group that you provide determines which authorization table is involved. If you do not specify an authorization group name, the group IDs are removed from the cell level authorization table. The group ID can be a short name or fully qualified domain name if a LDAP user registry is used.	None	• Parameters: **- authorizationGroupName** The name of the authorization group. If you do not specify this parameter, the cell level authorization group is assumed. (String, optional) **- roleName** The name of the administrative role. (String, required) **- userids** A list of group IDs that you want to remove from the administrative role. (String[], required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeGroupsFromAdminRole {-authorizationGroupName groupName - roleName administrator -groupids group1}` • Using Jython string: `AdminTask.removeGroupsFromAdminRole('[-authorizationGroupName groupName -roleName administrator -groupids group1]')` • Using Jython list: `AdminTask.removeGroupsFromAdminRole(['-authorizationGroupName', 'groupName', '-roleName', 'administrator', '-groupids', 'group1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeGroupsFromAdminRole {-interactive}` • Using Jython string: `AdminTask.removeGroupsFromAdminRole ('[-interactive]')` • Using Jython list: `AdminTask.removeGroupsFromAdminRole (['-interactive'])`

removeResourceFromAuthorizationGroup	The **removeResourceFromAuthorizationGroup** command removes resources from an existing authorization group. If you do not specify the authorization group, it will be determined and the resource will be removed from that authorization group.	None	• Parameters: **- authorizationGroupName** The name of the authorization group. (String, optional) **- resourceName** The name of the resource instance that you want to remove from the authorization group. (String, required) The resourceName parameter must be in the following format: `ResourceType=ResourceName` where the `ResourceType` can be any of the following: Application, Server, ServerCluster, Node, or NodeGroup. The `ResourceName` is the name of the resource instance, for example, server1. The following are examples of the resourceName parameter: `Node=node1:Server=server1` This example uniquely identifies server1. node1 is required if the name of the server exists on multiple nodes. `Application=app1` • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeResourceFromAuthorizationGroup {-authorizationGroupName groupName -resourceName Application=app1}` • Using Jython string: `AdminTask.removeResourceFromAuthorizationGroup('[-authorizationGroupName groupName -resourceName Application=app1]')` • Using Jython list: `AdminTask.removeResourceFromAuthorizationGroup(['-authorizationGroupName', 'groupName', '-resourceName', 'Application=app1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeResourceFromAuthorizationGroup {-interactive}` • Using Jython string: `AdminTask.removeResourceFromAuthorizationGroup ('[-interactive]')` • Using Jython list: `AdminTask.removeResourceFromAuthorizationGroup (['-interactive'])`

removeUsers FromAdmin Role	The **removeUsers FromAdmin Role** command removes previously mapped user IDs from administrative roles in the authorization group. The name of the authorization group that you provide determines which authorization table is involved. If you do not specify an authorization group name, the user ID from the cell level authorization table will be used. The user ID can be a short name or a fully qualified domain name if a LDAP user registry is used.	None	• Parameters: **- authorizationGroup Name** The name of the authorization group. If you do not specify this parameter, the cell level authorization group is assumed. (String, optional) **- roleName** The name of the administrative role. (String, required) **- userids** A list of user IDs that you want to remove from the administrative role. (String[], required) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeUsersFromAdmin Role {-authorizationGroupName groupName - roleName administra tor -userids user1}` • Using Jython string: `AdminTask.removeUsersFromAdmin Role('[-authorizationGroupName groupName -roleName administra tor -userids user1]')` • Using Jython list: `AdminTask.removeUsersFromAdmin Role(['-authorizationGroupName', 'groupName', '-roleName', 'adm inistrator', '-userids', 'user1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeUsersFrom AdminRole {-interactive}` • Using Jython string: `AdminTask.removeUsersFrom AdminRole ('[-interactive]')` • Using Jython list: `AdminTask.removeUsersFrom AdminRole (['-interactive'])`

Commands for the ChannelFrameworkManagement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the ChannelFrameworkManagement group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

createChain	The **createChain** command creates a new chain of transport channels that are based on a chain template.	The instance of the transport channel service under which the new chain is created. (ObjectName, required)	• Parameters: **- template** The chain template on which to base the new chain. (ObjectName, required) **- name** The name of the new chain. (String, required) **- endPoint** The name of the end point to be used by the instance of the TCP inbound channel in the new chain if the chain is an inbound chain. (ObjectName, optional) • Returns: The object name of the channel chain that was created.	**Batch mode example usage:** • Using Jacl: `$AdminTask createChain (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr\|server.xml#TransportChannelService_1) {-template WebContainer(templates/chains\|webcontainer-chains.xml#Chain_1) -name trialChain1}` `$AdminTask createChain (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1) {-template WebContainer(templates/chains|webcontainer-chains.xml#Chain_1) -name trialChain1 -endPoint (cells/rohitbuildCell01/nodes/rohitbuildCellManager01|serverindex.xml#EndPoint_3) }` • Using Jython string: `AdminTask.createChain('cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1', '[-template "WebContainer(templates/chains|webcontainer-chains.xml#Chain_1)" -name trialChain]')` `AdminTask.createChain('cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1', '[-template "WebContainer(templates/chains|webcontainer-chains.xml#Chain_1)" -name trialChain -endPoint "(cells/rohitbuildCell01/nodes/rohitbuildCellManager01|serverindex.xml#EndPoint_3)"]')` • Using Jython list: `AdminTask.createChain('cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1', ['-template', "WebContainer(templates/chains|webcontainer-chains.xml#Chain_1)", '-name', 'trialChain'])` `AdminTask.createChain('cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1', ['-template', "WebContainer(templates/chains|webcontainer-chains.xml#Chain_1)", '-name', 'trialChain', '-endPoint', "(cells/rohitbuildCell01/nodes/rohitbuildCellManager01|serverindex.xml#EndPoint_3)"])`

				Interactive mode example usage: • Using Jacl: `$AdminTask createChain {-interactive}` • Using Jython string: `AdminTask.createChain ('[-interactive]')` • Using Jython list: `AdminTask.createChain (['-interactive'])`
deleteChain	The **deleteChain** command deletes an existing chain and, optionally, the transport channels in the chain.	The chain to be deleted. (Object name, required)	• Parameters: **- deleteChannels** If the value of this attribute is true, non-shared transport channels used by the specified chain will be deleted. (Boolean, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deleteChain trialChain1 (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr\|server.xml#Chain_1093554462922)` `$AdminTask deleteChain trialChain (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#Chain_1093554378078) {-deleteChannels true}` • Using Jython string: `AdminTask.deleteChain('trialChain1(cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)')` `AdminTask.deleteChain('trialChain1 (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)', '[-deleteChannels true]')` • Using Jython list: `AdminTask.deleteChain('trialChain1 (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)')` `AdminTask.deleteChain('trialChain1 (cells/rohitbuildCell01/nodes/rohitbuildCellManager01/servers/dmgr|server.xml#TransportChannelService_1)', ['-deleteChannels', 'true'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deleteChain {-interactive}` • Using Jython string: `AdminTask.deleteChain ('[-interactive]')` • Using Jython list: `AdminTask.deleteChain (['-interactive'])`

listChain Templates	The **listChain Templates** command displays a list of templates that you can use to create chains in this configuration. All templates have a certain type of transport channel as the last transport channel in the chain.	None	• Parameters: **- acceptorFilter** The templates returned by this method all have a transport channel instance of the specified type as the last transport channel in the chain. (String, optional) • Returns: A list of all the chain template object names. If you specify the acceptorFilter parameter, the list that returns is filtered to match the filter that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask listChainTemplates {}` `$AdminTask listChainTemplates "-acceptorFilter WebContainer InboundChannel"` • Using Jython string: `AdminTask.listChainTemplates()` `AdminTask.listChainTemplates ('[-acceptorFilter WebCont ainerInboundChannel]')` • Using Jython list: `AdminTask.listChainTemplates()` `AdminTask.listChainTemplates (['-acceptorFilter', 'WebC ontainerInboundChannel'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listChainTemplates {-interactive}` • Using Jython string: `AdminTask.listChainTemplates ('[-interactive]')` • Using Jython list: `AdminTask.listChainTemplates (['-interactive'])`

listChains	The **listChains** command lists all the chains that are configured under a particular instance of the transport channel service.	The instance of the transport channel service under which the chains are configured. (ObjectName, required)	• Parameters: **- acceptorFilter** The chains that are returned by this parameter will have a transport channel instance of the type that you specify as the last transport channel in the chain. (String, optional) **- endPointFilter:** The chains returned by this parameter will have a TCP inbound channel using an end point with the name that you specify.(String, optional) • Returns: A list of all the channel chain object names that match the specified filters. If no you do not specify any parameters, all of the channel chains that are configured under the particular instance of transport channel service are returned.	**Batch mode example usage:** • Using Jacl: `$AdminTask listChains (cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2\|server.xml#TransportChannelService_1093445762328)` `$AdminTask listChains (cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328) {-acceptorFilter *WebContainerInboundChannel*}` `$AdminTask listChains (cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328) {-endPointFilter *WC_adminhost*}` • Using Jython string: `AdminTask.listChains('(cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328)')` `AdminTask.listChains('(cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328)', '[-acceptorFilter WebContainerInboundChannel]')` `AdminTask.listChains('(cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328)', '[-endPointFilter WC_adminhost]')` • Using Jython list: `AdminTask.listChains('(cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328)')` `AdminTask.listChains('(cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328)', ['-acceptorFilter', 'WebContainerInboundChannel'])` `AdminTask.listChains('(cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2|server.xml#TransportChannelService_1093445762328)', ['-endPointFilter', 'WC_adminhost'])`

				Interactive mode example usage: • Using Jacl: `$AdminTask listChains {-interactive}` • Using Jython string: `AdminTask.listChains ('[-interactive]')` • Using Jython list: `AdminTask.listChains (['-interactive'])`

Configuring data access with scripting

Use these topics to learn about using scripting to configure data access.

This topic contains the following tasks:

Configuring a JDBC provider using scripting

You can configure a JDBC provider using the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

1. There are two ways to perform this task. Perform one of the following:
   - Using the AdminTask object:
     - Using Jacl:
       ```
       $AdminTask createJDBCProvider {-interactive}
       ```
     - Using Jython:
       ```
       AdminTask.createJDBCProvider (['-interactive'])
       ```
   - Using the AdminConfig object:
     a. Identify the parent ID and assign it to the node variable. The following example uses the node configuration object as the parent. You can modify this example to use the cell, cluster, server, or application configuration object as the parent.

– Using Jacl:

```
set node [$AdminConfig  getid  /Cell:mycell/Node:mynode/]
```

– Using Jython:

```
node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
print node
```

Example output:

```
mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
```

b. Identify the required attributes:

– Using Jacl:

```
$AdminConfig required JDBCProvider
```

– Using Jython:

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

Example output:

```
Attribute     Type
name     String
implementationClassName   String
```

c. Set up the required attributes and assign it to the jdbcAttrs variable. You can modify the following example to setup non-required attributes for JDBC provider.

– Using Jacl:

```
set n1 [list name JDBC1]
set implCN [list implementationClassName myclass]
set  jdbcAttrs [list  $n1  $implCN]
```

Example output:

```
{name {JDBC1}} {implementationClassName {myclass}}
```

– Using Jython:

```
n1 = ['name', 'JDBC1']
implCN = ['implementationClassName', 'myclass']
jdbcAttrs = [n1,  implCN]
print jdbcAttrs
```

Example output:

```
[['name', 'JDBC1'], ['implementationClassName', 'myclass']]
```

d. Create a new JDBC provider using node as the parent:

– Using Jacl:

```
$AdminConfig create JDBCProvider $node $jdbcAttrs
```

– Using Jython:

```
AdminConfig.create('JDBCProvider', node, jdbcAttrs)
```

Example output:

```
JDBC1(cells/mycell/nodes/mynode|resources.xml#JDBCProvider_1)
```

2. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
3. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

If you modify the class path or native library path of a JDBC provider: After saving your changes (and synchronizing the node in a network deployment environment), you must restart every application server within the scope of that JDBC provider for the new configuration to work. Otherwise, you receive a data source failure message.

Configuring new data sources using scripting

You can configure new data sources using scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

In WebSphere Application Server, any JDBC driver properties that are required by your database vendor must be set as data source properties. Consult the article Vendor-specific data sources minimum required settings in the *Troubleshooting and support* PDF to see a list of these properties and setting options, ordered by JDBC provider type.

There are two ways to perform this task; use either of the following wsadmin scripting objects:

- AdminTask object
- AdminConfig object

AdminConfig gives you more configuration control than the AdminTask object. When you create a data source using AdminTask, you supply universally required properties only, such as a JNDI name for the data source. (Consult the article “Commands for the JDBCProviderManagement group of the AdminTask object” on page 616 for more information.) Other properties that are required by your JDBC driver are assigned default values by Application Server. You cannot use AdminTask commands to set or edit these properties; you must use AdminConfig commands.

- Using the AdminConfig object to configure a new data source:
  1. Identify the parent ID, which is the name and location of the JDBC provider that supports your data source.
     - Using Jacl:

       ```
       set newjdbc [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/]
       ```

     - Using Jython:

       ```
       newjdbc = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/')
       print newjdbc
       ```

       Example output:

       ```
       JDBC1(cells/mycell/nodes/mynode|resources.xml#JDBCProvider_1)
       ```

  2. Obtain the required attributes.

     Tip: If the database vendor-required properties (which are referenced in the article Vendor-specific data sources minimum required settings) are not displayed in the resulting list of required attributes, script these properties as data source *custom properties* after you create the data source.

     - Using Jacl:

       ```
       $AdminConfig required DataSource
       ```

     - Using Jython:

       ```
       print AdminConfig.required('DataSource')
       ```

       Example output:

       ```
       Attribute   Type
       name    String
       ```

  3. Set up the required attributes.
     - Using Jacl:

       ```
       set name [list name DS1]
       set dsAttrs [list $name]
       ```

     - Using Jython:

       ```
       name = ['name', 'DS1']
       dsAttrs = [name]
       ```

  4. Create the data source.
     - Using Jacl:

       ```
       set newds [$AdminConfig create DataSource $newjdbc $dsAttrs]
       ```

– Using Jython:

```
newds = AdminConfig.create('DataSource', newjdbc, dsAttrs)
print newds
```

Example output:

```
DS1(cells/mycell/nodes/mynode|resources.xml#DataSource_1)
```

- Using the AdminTask object to configure a new data source:
  – Using Jacl:

```
$AdminTask createDatasource {-interactive}
```

  – Using Jython:

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

- Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
- Synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Consult your database documentation to see if the vendor recommends setting additional properties. Script them as data source custom properties. See the article “Configuring new data source custom properties using scripting” on page 556.

Configuring new connection pools using scripting

You can use scripting and the wsadmin tool to configure new connection pools.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps:

1. Identify the parent ID:
   - Using Jacl:

```
set newds [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/]
```

   - Using Jython:

```
newds = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/')
```

   Example output:

```
DS1(cells/mycell/nodes/mynode|resources.xml$DataSource_1)
```

2. Creating connection pool:
   - Using Jacl:

```
$AdminConfig create ConnectionPool $newds {}
```

   - Using Jython:

```
print AdminConfig.create('ConnectionPool', newds, [])
```

   Example output:

```
(cells/mycell/nodes/mynode|resources.xml#ConnectionPool_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Changing connection pool settings with the wsadmin tool

You can use the wsadmin scripting tool to change connection pool settings.

The WebSphere Application Server wsadmin tool provides the ability to run scripts. You can use the wsadmin tool to manage a WebSphere Application Server installation, as well as configuration, application deployment, and server run-time operations. The WebSphere Application Server only supports the Jacl and Jython scripting languages. To learn more about the wsadmin tool see Starting the wsadmin scripting client.

To use the wsadmin tool to change connection pool settings:

1. Launch a scripting command. There are several options for you to run scripting commands, ranging from running them interactively to running them in a profile.

   To change connection pool settings, you use the *getAttribute* and *setAttribute* commands, run against the various settings objects.

   For example, to change the connection timeout setting, the commands are:

   ```
   $AdminControl getAttribute $objectname connectionTimeout
   $AdminControl setAttribute $objectname connectionTimeout 200
   ```

   where:
   - $ is a Jacl operator for substituting a variable name with its value
   - getAttribute and setAttribute are the commands
   - connectionTimeout is the object whose value you are resetting
2. For Jacl code examples of each of the connection pool settings, see "Example: Changing connection pool settings with the wsadmin tool"

Example: Changing connection pool settings with the wsadmin tool

You can use the wsadmin tool to change connection pool settings.

The WebSphere Application Server wsadmin tool provides the ability to run scripts. The WebSphere Application Server only supports the Jacl and Jython scripting languages.

You must start the wsadmin scripting client before you perform any other task using scripting.

Connection Timeout

The Connection timeout can be changed at any time while the pool is active. All connection requests that are waiting for a connection are changed to the new value minus the time already waited, and are returned to the wait state if there are no available connections.

For example, if the connection timeout is changed to 300 seconds and a connection request has waited 100 seconds, the connection request waits for 200 more seconds if no connection becomes available.

```
$AdminControl getAttribute $objectname connectionTimeout
$AdminControl setAttribute $objectname connectionTimeout 200
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Maximum Connections

The maximum connections can be changed at any time except in the case where stuck connection support is active.

If stuck connection support is active, an attempt to change the maximum connections is made. If the attempt fails, an `IllegalState` exception occurs. See the Connection pool advanced settings topic in the *Administering applications and their environment* PDF for more information.

If stuck connection support is not active, the maximum connections are changed to the new value. If the new value is greater than the current value, the number of connections increases to the new value and any requests waiting are notified. If the new value is less than the current value and Aged Timeout or

Reap Time is used, the number of connections decreases to the new value depending on pool activity. If agedTimeout or Reap Time are not used, no automatic attempt will be made to reduce the total number of connections. To manually reduce the number of connections to the new maximum connections, use the Mbean function purgePoolContents.

```
$AdminControl getAttribute $objectname maxConnections
$AdminControl setAttribute $objectname maxConnections 200
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Minimum Connections

The minimum connections can be changed at any time

```
$AdminControl getAttribute $objectname minConnections
$AdminControl setAttribute $objectname minConnections 200
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Reap Time

The reap time can be changed at any time. The reap time interval is changed to the new value at the next interval.

```
$AdminControl getAttribute $objectname reapTime
$AdminControl setAttribute $objectname reapTime 30
```

Unused Timeout

The unused timeout can be changed at any time.

```
$AdminControl getAttribute $objectname unusedTimeout
$AdminControl setAttribute $objectname unusedTimeout 900
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Aged Timeout

The Aged Timeout can be changed at any time.

```
$AdminControl getAttribute $objectname agedTimeout
$AdminControl setAttribute $objectname agedTimeout 900
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Purge Policy

The purge policy can be changed at any time.

```
$AdminControl getAttribute $objectname purgePolicy
$AdminControl setAttribute $objectname purgePolicy "Failing Connection Only"
```

For more information about this setting, see the Connection pool settings topic in the *Administering applications and their environment* PDF.

Surge Protection Support

Surge connection support starts if surgeThreshold is > -1 and surgeCreationInterval is > 0. The surge protection properties can be changed at any time.

```
$AdminControl getAttribute $objectname surgeCreationInterval
$AdminControl setAttribute $objectname surgeCreationInterval 30
$AdminControl getAttribute $objectname surgeThreshold
$AdminControl setAttribute $objectname surgeThreshold 15
```

For more information about this setting, see the Connection pool advanced settings topic in the *Administering applications and their environment* PDF.

Stuck connection Support

An attempt is made to change the stuckTime, stuckTimerTime or stuckThreshold properties. If the attempt fails, an `IllegalState` exception occurs. The pool cannot have any active requests or active connections during this request. For the stuck connection support to start, all three stuck property values must be greater than 0, and the maximum connections value must be > 0.

If the connection pool is stuck, you cannot change the stuck or the maximum connection properties. If you are stuck, there are active connections.

```
$AdminControl getAttribute $objectname stuckTime
$AdminControl setAttribute $objectname stuckTime 30
$AdminControl getAttribute $objectname stuckTimerTime
$AdminControl setAttribute $objectname stuckTimerTime 15
$AdminControl getAttribute $objectname stuckThreshold
$AdminControl setAttribute $objectname stuckThreshold 10
```

For more information about this setting, see the Connection pool advanced settings topic in the *Administering applications and their environment* PDF.

Test Connection Support (This is only supported for a DataSource)

The test connection support starts when the testConnection property is set to true, and the interval is > 0. The test connection properties can be changed at any time.

```
$AdminControl getAttribute $objectname testConnection
$AdminControl setAttribute $objectname testConnection 30
$AdminControl getAttribute $objectname testConnectionInterval
$AdminControl setAttribute $objectname testConnectionInterval 15
```

For more information about this setting, see the Test connection service topic in the *Administering applications and their environment* PDF.

Connection Pool Partition Support

```
$AdminControl invoke $objectname freePoolDistributionTableSize
$AdminControl invoke $objectname numberOfFreePoolPartitions
$AdminControl invoke $objectname numberOfSharedPoolPartitions
$AdminControl invoke $objectname gatherPoolStatisticalData
$AdminControl invoke $objectname enablePoolStatisticalData
```

For more information about this setting, see the Connection pool advanced settings topic in the *Administering applications and their environment* PDF.

Show Pool Information Support

The show pool operations can be changed at any time.

```
$AdminControl invoke $objectname showAllPoolContents
$AdminControl invoke $objectname showPoolContents
$AdminControl invoke $objectname showAllocationHandleList
```

Purge Connection Support

PurgePool can be changed at any time.

```
$AdminControl invoke $objectname purgePoolContents normal
$AdminControl invoke $objectname purgePoolContents immediate
```

Pool Control Support

Pause and resume can be changed at any time.

```
$AdminControl invoke $objectname pause
$AdminControl invoke $objectname resume
```

Example: Connection factory and data source Mbean access using wsadmin

Wsadmin commands are used to access connection factory and data source MBeans. MBeans can retrieve or update properties for a connection factory or data source.

To get a list of J2C connection factory or data source Mbean object names, from the wsadmin command line use one of the following administrative control commands:

```
$AdminControl queryNames *:type=J2CConnectionFactory,*
$AdminControl queryNames *:type=DataSource,*
```

Example output from DataSource query:

```
wsadmin>$AdminControl queryNames *:type=DataSource,*
"WebSphere:name=Default Datasource,process=server1,platform=dynamicproxy,node=
system1Node01,JDBCProvider=Cloudscape JDBC Provider,j2eeType=JDBCDataSource,J2EESe
rver=server1,Server=server1,version=6.0.0.0,type=DataSource,mbeanIdentifier=cell
s/system1Node01Cell/nodes/system1Node01/servers/server1/resources.xml#DataSource
_1094760149902,JDBCResource=Cloudscape JDBC Provider,cell=system1Node01Cell"
```

By using the J2C connection factory or data source MBean object name, you can access the MBean. The name follows the `webSphere:name=` expression. In the following example, for the first set, the default data source name is set on variable name. For the second set, the object name is set on variable *objectName*.

Example using the Default Datasource

```
- wsadmin>set name [list Default Datasource]

Default Datasource


- wsadmin>set objectName [$AdminControl queryNames *:name=$name,*]


"WebSphere:name=Default Datasource,process=server1,platform=dynamicproxy,node=
system1Node01,JDBCProvider=Cloudscape JDBC Provider,j2eeType=JDBCDataSource,J2EESe
rver=server1,Server=server1,version=6.0.0.0,type=DataSource,mbeanIdentifier=cell
s/system1Node01Cell/nodes/system1Node01/servers/server1/resources.xml#DataSource
_1094760149902,JDBCResource=Cloudscape JDBC Provider,cell=system1Node01Cell"
```

Now you can use the objectName for getting help on attributes and operations available for this mbean.

```
$Help attributes $objectName
$Help operations $objectName
```

To get or set an attribute or to use an operation, use one of the following commands:

```
$AdminControl getAttribute $objectName "attribute name"
$AdminControl setAttribute $objectName "attribute name" value
$AdminControl invoke $objectName "operation"
```

Attribute, operation examples:

```
Get and set a attribute maxConnections (Note, if you get no value, a null value, or
a java.lang.IllegalStateException, the connection factory or data source for this
MBean has not been created. The connection factory or data source is created at first
JNDI lookup. For this example, the Default Datasource needs to be used before get,
set and invoke will work.)

wsadmin>$AdminControl getAttribute $objectName maxConnections
10

wsadmin>$AdminControl setAttribute $objectName maxConnections 20

wsadmin>$AdminControl getAttribute $objectName maxConnections
20

Using invoke

wsadmin>$AdminControl invoke $objectName showPoolContents

PoolManager name:DefaultDatasource
PoolManager object:746354514
Total number of connections: 3 (max/min 20/1, reap/unused/aged 180/1800/0,
connectiontimeout/purge 1800/EntirePool)
                            (testConnection/inteval false/0, stuck timer/time
/threshold 0/0/0, surge time/connections 0/-1)
Shared Connection information (shared partitions 200)
  No shared connections

Free Connection information (free distribution table/partitions 5/1)
  (2)(0)MCWrapper id 5c6af75b  Managed connection WSRdbManagedConnectionImpl@4b5
a775b  State:STATE_ACTIVE_FREE
  (2)(0)MCWrapper id 3394375a  Managed connection WSRdbManagedConnectionImpl@328
5f75a  State:STATE_ACTIVE_FREE
  (2)(0)MCWrapper id 4795b75a  Managed connection WSRdbManagedConnectionImpl@46a
4b75a  State:STATE_ACTIVE_FREE

  Total number of connection in free pool: 3
UnShared Connection information
  No unshared connections
```

Here is an example Java program using the wsadmin tool to invoke showPoolContents: the Example: using wsadmin to invoke showPoolContents topic in the *Administering applications and their environment* PDF.

Example: Invoking showPoolContents using the wsadmin tool: The following example Java program uses the wsadmin tool to invoke showPoolContents:

```
/**
 * "This sample program is provided AS IS and may be used, executed, copied and modified without royalty payment
 *  by customer (a) for its own instruction and study, (b) in order to develop applications designed to run with
 *  an IBM WebSphere product, either for customer's own internal use or for redistribution by customer, as part
 *  of such an application, in customer's own products. "

 * Product 5724i63, (C) COPYRIGHT International Business Machines Corp., 2004


 *
 * All Rights Reserved * Licensed Materials - Property of IBM
 *
 * This program will display an active mbean object name and the connection pool.
 *
 * To run this program
```

```
 *
 *  1. call "%WAS_HOME%/bin/setupCmdLine.bat"
 *
 *  2. "%JAVA_HOME%\bin\java" "%CLIENTSAS%" "-Dwas.install.root=%WAS_HOME%" "-Dwas.repository.root=%CONFIG_ROOT%"
 *     -Dcom.ibm.CORBA.BootstrapHost=%COMPUTERNAME% -classpath "%WAS_CLASSPATH%;%WAS_HOME%\lib\admin.jar;%WAS_HOME%
 *   \lib\wasjmx.jar;." ShowPoolContents DataSource_mbean_name
 */

import java.util.Properties;
import java.util.Set;

import javax.management.*;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.exception.ConnectorException;

public class ShowPoolContents {

  private AdminClient adminClient;

  public static void main(String[] args) {
    try {
      String name2 = null;
      String port = null;
      if(args.length <2){
        System.out.println("Enter name for the mbean and port");
        return;
      } else {
        name2 = args[0];
        port = args[1];
        System.out.println("Searching for name"+name2);
      }
      ShowPoolContents ace = new ShowPoolContents();

      // Create an AdminClient
      ace.createAdminClient(port);

      ObjectName[] cfON = ace.queryMBean("DataSource", null);

      if (cfON.length == 0) {
        System.out.println(" *Error : queryMBean did not find any active mbeans of type DataSource.");
        System.out.println(" At first touch of a DataSource, an mbean will be created.  To touch a
         DataSource, use  " + "getConnection");
        return;
      }
      int selectedObjectName = -1;
      String findName = name2;
      for(int i=0;i " + results);

    } catch (Exception e) {
      System.out.println(" *Exception : " + e.toString());
      e.printStackTrace(System.out);
    }
  }

  private void createAdminClient(String port) {
    // Set up a Properties object for the JMX connector attributes
    Properties connectProps = new Properties();
    connectProps.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
    connectProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
    connectProps.setProperty(AdminClient.CONNECTOR_PORT, port);

    // Get an AdminClient based on the connector properties
    try {
      adminClient = AdminClientFactory.createAdminClient(connectProps);
    } catch (ConnectorException e) {
```

```
      System.out.println("Exception creating admin client: " + e);
      System.exit(-1);
    }

    System.out.println("Connected to DeploymentManager");
  }

  // helper method to query mbean by type / name
  public ObjectName[] queryMBean(String type, String name) throws Exception {
    String s = "*:";
    if (type != null)
      s += "type=" + type;

    if (name != null)
      s += ",name=" + name;

    s += ",*";

    System.out.println("queryMBean: " + s);
    ObjectName ion = new ObjectName(s);

    Set set = adminClient.queryNames(ion, null);
    Object[] o = set.toArray();
    ObjectName[] on = new ObjectName[o.length];

    for (int i = 0; i <o.length; i++) {
      on[i] = (ObjectName) o[i];
}

    return on;
  }
}
```

Configuring new data source custom properties using scripting

You can configure new data source custom properties using the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new data source custom property:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set newds [$AdminConfig getid /Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/]
     ```
   - Using Jython:
     ```
     newds = AdminConfig.getid('/Cell:mycell/Node:mynode/JDBCProvider:JDBC1/DataSource:DS1/')
     print newds
     ```

   Example output:

   ```
   DS1(cells/mycell/nodes/mynode|resources.xml$DataSource_1)
   ```
2. Get the J2EE resource property set:
   - Using Jacl:
     ```
     set propSet [$AdminConfig showAttribute $newds propertySet]
     ```
   - Using Jython:
     ```
     propSet = AdminConfig.showAttribute(newds, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_8)
   ```
3. Get required attribute:
   - Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

- Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

Example output:

```
Attribute     Type
name         String
```

4. Set up attributes:
   - Using Jacl:

     ```
     set name [list name RP4]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP4']
     rpAttrs = [name]
     ```

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP4(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_8)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new J2CAuthentication data entries using scripting

You can configure new J2CAuthentication data entries with the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new J2CAuthentication data entry:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set security [$AdminConfig getid /Cell:mycell/Security:/]
     ```

   - Using Jython:

     ```
     security = AdminConfig.getid('/Cell:mycell/Security:/')
     print security
     ```

   Example output:

   ```
   (cells/mycell|security.xml#Security_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required JAASAuthData
     ```

   - Using Jython:

     ```
     print AdminConfig.required('JAASAuthData')
     ```

   Example output:

   ```
   Attribute       Type
   alias           String
   userId          String
   password        String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set alias [list alias myAlias]
     set userid [list userId myid]
     set password [list password secret]
     set jaasAttrs [list $alias $userid $password]
     ```

     Example output:

     ```
     {alias myAlias} {userId myid} {password secret}
     ```

   - Using Jython:

     ```
     alias = ['alias', 'myAlias']
     userid = ['userId', 'myid']
     password = ['password', 'secret']
     jaasAttrs = [alias, userid, password]
     print jaasAttrs
     ```

     Example output:

     ```
     [['alias', 'myAlias'], ['userId', 'myid'], ['password', 'secret']]
     ```

4. Create JAAS auth data:
   - Using Jacl:

     ```
     $AdminConfig create JAASAuthData $security $jaasAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('JAASAuthData', security, jaasAttrs)
     ```

   Example output:

   ```
   (cells/mycell|security.xml#JAASAuthData_2)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WAS40 data sources using scripting

Use scripting to configure a new WAS40 data source.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjdbc [$AdminConfig getid "/JDBCProvider:Cloudscape JDBC Provider/"]
     ```

   - Using Jython:

     ```
     newjdbc = AdminConfig.getid('/JDBCProvider:Cloudscape JDBC Provider/')
     print newjdbc
     ```

   Example output:

   ```
   JDBC1(cells/mycell/nodes/mynode|resources.xml$JDBCProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WAS40DataSource
     ```

   - Using Jython:

     ```
     print AdminConfig.required('WAS40DataSource')
     ```

   Example output:

```
Attribute    Type
name    String
```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name was4DS1]
     set ds4Attrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'was4DS1']
     ds4Attrs = [name]
     ```

4. Create WAS40DataSource:
   - Using Jacl:

     ```
     set new40ds [$AdminConfig create WAS40DataSource $newjdbc $ds4Attrs]
     ```

   - Using Jython:

     ```
     new40ds = AdminConfig.create('WAS40DataSource', newjdbc, ds4Attrs)
     print new40ds
     ```

   Example output:

   ```
   was4DS1(cells/mycell/nodes/mynode|resources.xml#WAS40DataSource_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WAS40 connection pools using scripting

You can use scripting to configure a new WAS40 connection pool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new WAS40 connection pool:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set new40ds [$AdminConfig getid /Cell:mycell/Node:mynode/
     Server:server1/JDBCProvider:JDBC1/WAS40DataSource:was4DS1/]
     ```

   - Using Jython:

     ```
     new40ds = AdminConfig.getid('/Cell:mycell/Node:mynode/
     Server:server1/JDBCProvider:JDBC1/WAS40DataSource:was4DS1/')
     print new40ds
     ```

   Example output:

   ```
   was4DS1(cells/mycell/nodes/mynodes:resources.xml$WAS40DataSource_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WAS40ConnectionPool
     ```

   - Using Jython:

     ```
     print AdminConfig.required('WAS40ConnectionPool')
     ```

   Example output:

   ```
   Attribute       Type
   minimumPoolSize     Integer
   maximumPoolSize     Integer
   connectionTimeout   Integer
   idleTimeout        Integer
   orphanTimeout       Integer
   statementCacheSize  Integer
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set mps [list minimumPoolSize 5]
     set minps [list minimumPoolSize 5]
     set maxps [list maximumPoolSize 30]
     set conn [list connectionTimeout 10]
     set idle [list idleTimeout 5]
     set orphan [list orphanTimeout 5]
     set scs [list statementCacheSize 5]
     set 40cpAttrs [list $minps $maxps $conn $idle $orphan $scs]
     ```

     Example output:

     ```
     {minimumPoolSize 5} {maximumPoolSize 30}
     {connectionTimeout 10} {idleTimeout 5}
     {orphanTimeout 5} {statementCacheSize 5}
     ```

   - Using Jython:

     ```
     minps = ['minimumPoolSize', 5]
     maxps = ['maximumPoolSize', 30]
     conn = ['connectionTimeout', 10]
     idle = ['idleTimeout', 5]
     orphan = ['orphanTimeout', 5]
     scs = ['statementCacheSize', 5]
     cpAttrs = [minps, maxps, conn, idle, orphan, scs]
     print cpAttrs
     ```

     Example output:

     ```
     [[minimumPoolSize, 5], [maximumPoolSize, 30],
     [connectionTimeout, 10], [idleTimeout, 5],
     [orphanTimeout, 5], [statementCacheSize, 5]]
     ```

4. Create was40 connection pool:
   - Using Jacl:

     ```
     $AdminConfig create WAS40ConnectionPool $new40ds $40cpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WAS40ConnectionPool', new40ds, 40cpAttrs)
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode:resources.xml#WAS40ConnectionPool_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WAS40 custom properties using scripting

You can use scripting and the wsadmin tool to configure a new WAS40 custom property.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new WAS40 custom properties:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set new40ds [$AdminConfig getid /Cell:mycell/Node:mynode/
     JDBCProvider:JDBC1/WAS40DataSource:was4DS1/]
     ```

   - Using Jython:

     ```
     new40ds = AdminConfig.getid('/Cell:mycell/Node:mynode/
     JDBCProvider:JDBC1/WAS40DataSource:was4DS1/')
     print new40ds
     ```

Example output:

```
was4DS1(cells/mycell/nodes/mynodes|resources.xml$WAS40DataSource_1)
```

2. Get required attributes:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newds propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newds, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_9)
   ```

3. Get required attribute:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute    Type
   name         String
   ```

4. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name RP5]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP5']
     rpAttrs = [name]
     ```

5. Create J2EE Resource Property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP5(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_9)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new J2C resource adapters using scripting

Use scripting to configure new J2C resource adapters in WebSphere Application Server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new J2C resource adapter:

1. Identify the parent ID and assign it to the node variable. The following example uses the node configuration object as the parent. You can modify this example to use the cell, cluster, server, or application configuration object as the parent.
   - Using Jacl:

     ```
     set node [$AdminConfig getid /Cell:mycell/Node:mynode/]
     ```

- Using Jython:

```
node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
print node
```

Example output:

```
mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
```

2. Identify the required attributes:
   - Using Jacl:

```
$AdminConfig required J2CResourceAdapter
```

   - Using Jython:

```
print AdminConfig.required('J2CResourceAdapter')
```

   Example output:

```
Attribute     Type
name         String
```

3. Set up the required attributes:
   - Using Jacl:

```
set rarFile /currentScript/cicseci.rar
set option  [list  -rar.name  RAR1]
```

   - Using Jython:

```
rarFile = '/currentScript/cicseci.rar'
option  = '[-rar.name  RAR1]'
```

4. Create a resource adapter:
   - Using Jacl:

```
set newra [$AdminConfig installResourceAdapter $rarFile mynode $option]
```

   - Using Jython:

```
newra = AdminConfig.installResourceAdapter(rarFile, 'mynode', option)
print newra
```

   Example output:

```
RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring custom properties for J2C resource adapters using scripting

You can configure custom properties for J2C resource adapters with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new custom property for a J2C resource adapters:

1. Identify the parent ID and assign it to the newra variable.
   - Using Jacl:

```
set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
```

   - Using Jython:

```
newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
print newra
```

   Example output:

```
RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
```

2. Get the J2EE resource property set:
   - Using Jacl:
     ```
     set propSet [$AdminConfig showAttribute $newra propertySet]
     ```
   - Using Jython:
     ```
     propSet = AdminConfig.showAttribute(newra, 'propertySet')
     print propSet
     ```

   Example output:
   ```
   (cells/mycell/nodes/mynode|resources.xml#PropertySet_8)
   ```
3. Identify the required attributes:
   - Using Jacl:
     ```
     $AdminConfig required J2EEResourceProperty
     ```
   - Using Jython:
     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:
   ```
   Attribute     Type
   name         String
   ```
4. Set up the required attributes:
   - Using Jacl:
     ```
     set name [list name RP4]
     set rpAttrs [list $name]
     ```
   - Using Jython:
     ```
     name = ['name', 'RP4']
     rpAttrs = [name]
     ```
5. Create a J2EE resource property:
   - Using Jacl:
     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```
   - Using Jython:
     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:
   ```
   RP4(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_8)
   ```
6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new J2C connection factories using scripting

Use scripting and the wsadmin tool to configure new J2C connection factories.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new J2C connection factory:

1. Identify the parent ID and assign it to the newra variable.
   - Using Jacl:
     ```
     set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
     ```
   - Using Jython:
     ```
     newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
     print newra
     ```

   Example output:

```
RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
```

2. There are two ways to configure a new J2C connection factory. Perform one of the following:
   - Using the AdminTask object:
     a. List the connection factory interfaces:
        - Using Jacl:

          ```
          $AdminTask listConnectionFactoryInterfaces $newra
          ```

        - Using Jacl:

          ```
          AdminTask.listConnectionFactoryInterfaces(newra)
          ```

          Example output:

          ```
          javax.sql.DataSource
          ```

     b. Create a J2CConnectionFactory:
        - Using Jacl:

          ```
          $AdminTask createJ2CConnectionFactory $newra { -name cf1
          -jndiName eis/cf1 -connectionFactoryInterface
          avax.sql.DataSource
          ```

        - Using Jacl:

          ```
          AdminTask.createJ2CConnectionFactory(newra, '['-name', 'cf1',
          '-jndiName', 'eis/cf1', '-connectionFactoryInterface',
          'avax.sql.DataSource']')
          ```

   - Using the AdminConfig object:
     a. Identify the required attributes:
        - Using Jacl:

          ```
          $AdminConfig required J2CConnectionFactory
          ```

        - Using Jython:

          ```
          print AdminConfig.required('J2CConnectionFactory')
          ```

          Example output:

          ```
          Attribute Type
          connectionDefinition ConnectionDefinition@
          ```

     b. If your resource adapter is JCA1.5 and you have multiple connection definitions defined, it is required that you specify the ConnectionDefinition attribute. If your resource adapter is JCA1.5 and you have only one connection definition defined, it will be picked up automatically. If your resource adapter is JCA1.0, you do not need to specify the ConnectionDefinition attribute. Perform the following command to list the connection definitions defined by the resource adapter:
        - Using Jacl:

          ```
          $AdminConfig list ConnectionDefinition $newra
          ```

        - Using Jython:

          ```
          print AdminConfig.list('ConnectionDefinition', $newra)
          ```

     c. Set up the required attributes:
        - Using Jacl:

          ```
          set name [list name J2CCF1]
          set j2ccfAttrs [list $name]
          set jname [list jndiName eis/j2ccf1]
          ```

        - Using Jython:

          ```
          name = ['name', 'J2CCF1']
          j2ccfAttrs = [name]
          jname = ['jndiName', eis/j2ccf1]
          ```

     d. If you are specifying the ConnectionDefinition attribute, also set up the following:
        - Using Jacl:

          ```
          set cdattr [list connectionDefinition $cd]
          ```

– Using Jython:

```
cdattr = ['connectionDefinition', $cd]
```

e. Create a J2C connection factory:

– Using Jacl:

```
$AdminConfig create J2CConnectionFactory $newra $j2ccfAttrs
```

– Using Jython:

```
print AdminConfig.create('J2CConnectionFactory', newra, j2ccfAttrs)
```

Example output:

```
J2CCF1(cells/mycell/nodes/mynode|resources.xml#J2CConnectionFactory_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new J2C authentication data entries using scripting

You can use scripting to configure a new J2C authentication data entry.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new J2C authentication data entry:

1. Identify the parent ID and assign it to the security variable.
   - Using Jacl:

     ```
     set security [$AdminConfig getid /Security:mysecurity/]
     ```

   - Using Jython:

     ```
     security = AdminConfig.getid('/Security:mysecurity/')
     ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required JAASAuthData
     ```

   - Using Jython:

     ```
     print AdminConfig.required('JAASAuthData')
     ```

   Example output:

   ```
   Attribute     Type
   alias         String
   userId          String
   password          String
   ```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set alias [list alias myAlias]
     set userid [list userId myid]
     set password [list password secret]
     set jaasAttrs [list $alias $userid $password]
     ```

     Example output:

     ```
     {alias myAlias} {userId myid} {password secret}
     ```

   - Using Jython:

     ```
     alias = ['alias', 'myAlias']
     userid = ['userId', 'myid']
     password = ['password', 'secret']
     jaasAttrs = [alias, userid, password]
     ```

     Example output:

```
[[alias, myAlias], [userId, myid], [password, secret]]
```

4. Create JAAS authentication data:
   - Using Jacl:
     ```
     $AdminConfig create JAASAuthData $security $jaasAttrs
     ```
   - Using Jython:
     ```
     print AdminConfig.create('JAASAuthData', security, jaasAttrs)
     ```

   Example output:
   ```
   (cells/mycell/nodes/mynode|resources.xml#JAASAuthData_2)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new J2C activation specifications using scripting

You can configure new J2C activation specifications using scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a J2C activation specifications:

1. Identify the parent ID and assign it to the `newra` variable.
   - Using Jacl:
     ```
     set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
     ```
   - Using Jython:
     ```
     newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
     print newra
     ```

   Example output:
   ```
   RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
   ```
2. There are two ways to configure a new J2C administrative object. Perform one of the following:
   - Using the AdminTask object:
     a. List the administrative object interfaces:

        Using Jacl:
        ```
        $AdminTask listMessageListenerTypes $newra
        ```
        Using Jython:
        ```
        AdminTask.listMessageListenerTypes(newra)
        ```
        Example output:
        ```
        javax.jms.MessageListener
        ```
     b. Create a J2C administrative object:

        Using Jacl:
        ```
        $AdminTask createJ2CActivationSpec $newra { -name ac1
        -jndiName eis/ac1 -message ListenerType
        javax.jms.MessageListener}
        ```
        Using Jython:
        ```
        AdminTask.createJ2CActivationSpec(newra, ['-name', 'ao1',
        '-jndiName', 'eis/ao1', '-message', 'ListenerType',
        'javax.jms.MessageListener'])
        ```
   - Using the AdminConfig object:
     a. Using Jacl:
        ```
        $AdminConfig required J2CActivationSpec
        ```

Using Jython:

```
print AdminConfig.required('J2CActivationSpec')
```

Example output:

```
Attribute Type
activationSpec ActivationSpec@
```

b. If your resource adapter is JCA V1.5 and you have multiple activation specifications defined, it is required that you specify the activation specification attribute. If your resource adapter is JCA V1.5 and you have only one activation specification defined, it will be picked up automatically. If your resource adapter is JCA V1.0, you do not need to specify the activationSpec attribute. Perform the following command to list the activation specifications defined by the resource adapter:

Using Jacl:

```
$AdminConfig list ActivationSpec $newra
```

Using Jython:

```
print AdminConfig.list('ActivationSpec', $newra)
```

c. Set the administrative object that you need to a variable:

Using Jacl:

```
set ac ActivationSpecID
set name [list name J2CAC1]
set jname [jndiName eis/j2cac1]
set j2cacAttrs [list $name $jname]
```

Using Jython:

```
ac = ActivationSpecID
name = ['name', 'J2CAC1']
jname = ['jndiName', 'eis/j2cac1']
j2cacAttrs = [name, jname]
```

d. If you are specifying the ActivationSpec attribute, also set up the following:

Using Jacl:

```
set cdcttr [list activationSpec $ac]
```

Using Jython:

```
cdattr = ['activationSpec', ac]
```

e. Create a J2C activation specification object:

Using Jacl:

```
$AdminConfig create J2CActivationSpec $newra $j2cacAttrs
```

Using Jython:

```
print AdminConfig.create('J2CActivationSpec', newra,j2cacAttrs)
```

Example output:

```
J2CAC1(cells/mycell/nodes/mynode|resources.xml#J2CActivationSpec_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new J2C administrative objects using scripting

You can use scripting and the wsadmin tool to configure new J2C administrative objects.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a J2C administrative object:

1. Identify the parent ID and assign it to the `newra` variable.

- Using Jacl:

  ```
  set newra [$AdminConfig getid /Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/]
  ```

- Using Jython:

  ```
  newra = AdminConfig.getid('/Cell:mycell/Node:mynode/J2CResourceAdapter:RAR1/')
  print newra
  ```

  Example output:

  ```
  RAR1(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)
  ```

2. There are two ways to configure a new J2C administrative object. Perform one of the following:
   - Using the AdminTask object:
     a. List the administrative object interfaces:

        Using Jacl:

        ```
        $AdminTask listAdminObjectInterfaces $newra
        ```

        Using Jython:

        ```
        AdminTask.listAdminObjectInterfaces(newra)
        ```

        Example output:

        ```
        com.ibm.test.message.FVTMessageProvider
        ```

     b. Create a J2C administrative object:

        Using Jacl:

        ```
        $AdminTask createJ2CAdminObject $newra { -name ao1 -jndiName eis/ao1
        -adminObjectInterface com.ibm.test.message.FVTMessageProvider }
        ```

        Using Jython:

        ```
        AdminTask.createJ2CAdminObject(newra, ['-name', 'ao1', '-jndiName', 'eis/ao1',
        '-adminObjectInterface', 'com.ibm.test.message.FVTMessageProvider'])
        ```

   - Using the AdminConfig object:
     a. Using Jacl:

        ```
        $AdminConfig required J2CAdminObject
        ```

        Using Jython:

        ```
        print AdminConfig.required('J2CAdminObject')
        ```

        Example output:

        ```
        Attribute Type
        adminObject AdminObject@
        ```

     b. If your resource adapter is JCA V1.5 and you have multiple administrative objects defined, it is required that you specify the administrative object attribute. If your resource adapter is JCA V1.5 and you have only one administrative object defined, it will be picked up automatically. If your resource adapter is JCA V1.0, you do not need to specify the administrative object attribute. Perform the following command to list the administrative objects defined by the resource adapter:

        Using Jacl:

        ```
        $AdminConfig list AdminObject $newra
        ```

        Using Jython:

        ```
        print AdminConfig.list('AdminObject', $newra)
        ```

     c. Set the administrative objects that you need to a variable:

        Using Jacl:

        ```
        set ao AdminObjectId
        set name [list name J2CAO1]
        set jname [jndiName eis/j2cao1]
        set j2caoAttrs [list $name $jname]
        ```

        Using Jython:

```
ao = AdminObjectId
name = ['name', 'J2CAO1']
set jname = ['jndiName', eis/j2cao1]
j2caoAttrs = [name, jname]
```

d. If you are specifying the AdminObject attribute, also set up the following:

Using Jacl:

```
set cdattr [list adminObject $ao]
```

Using Jython:

```
cdattr = ['adminObject', ao]
```

e. Create a J2C administrative object:

Using Jacl:

```
$AdminConfig create J2CAdminObject $newra $j2caoAttrs
```

Using Jython:

```
print AdminConfig.create('J2CAdminObject', newra, j2caoAttrs)
```

Example output:

```
J2CAO1(cells/mycell/nodes/mynode|resources.xml#J2CAdminObject_1)
```

3. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
4. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Testing data source connections using scripting

You can test connections for data sources with the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to test a data source to ensure a connection to the database.

1. Identify the DataSourceCfgHelper MBean and assign it to the dshelper variable.
   - Using Jacl:

     ```
     set ds [$AdminConfig getid /DataSource:DS1/]
     $AdminControl testConnection $ds
     ```

   - Using Jython:

     ```
     ds = AdminConfig.getid('/DataSource:DS1/')
     AdminControl.testConnection(ds)
     ```

   Example output:

   ```
   WASX7217I: Connection to provided datasource was successful.
   ```

2. Test the connection. The following example invokes the testConnectionToDataSource operation on the MBean, passing in the classname, userid, password, database name, JDBC driver class path, language, and country.
   - Using Jacl:

     ```
     $AdminControl invoke $dshelper testConnectionToDataSource
     "COM.ibm.db2.jdbc.DB2XADataSource db2admin db2admin
     {{databaseName sample}} /sqllib/java/db2java.zip en US"
     ```

   - Using Jython:

     ```
     print AdminControl.invoke(dshelper, 'testConnectionToDataSource',
     'COM.ibm.db2.jdbc.DB2XADataSource dbuser1 dbpwd1
     "{{databaseName jtest1}}" /sqllib/java12/db \"\" \"\"')
     ```

   Example output:

   ```
   WASX7217I: Connection to provided data source was successful.
   ```

Commands for the EventServiceDBCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the EventServiceDBCommands group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

configEvent ServiceDB 2DB	The **configEvent ServiceDB 2DB** command creates the event service database and data sources for DB2 on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/*dbtype* if this parameter is not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventServiceDB2DB {-createDB true -overrideDataSource true -nodeName nodename -serverName servername -jdbcClassPath c:\sqllib\java -dbUser db2inst1 -dbPassword dbpassword -dbHostName hostname -dbPort 50000 }` • Using Jython string: `AdminTask.configEventServiceDB2DB ('[-createDB true -overrideDataSource true -nodeName nodename -serverName servername -jdbcClassPath c:\sqllib\java -dbUser db2inst1 -dbPassword dbpassword -dbHostName hostname -dbPort 50000 ]')` • Using Jython list: `AdminTask.configEventServiceDB2DB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-jdbcClassPath', 'c:\sqllib\java', '-dbUser', 'db2inst1', '-dbPassword', 'dbpassword ', '-dbHostName', 'hostname', '-dbPort', '50000 '])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventServiceDB2DB -interactive` • Using Jython string: `AdminTask.configEventServiceDB2DB('[-interactive]')` • Using Jython list: `AdminTask.configEventServiceDB2DB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- jdbcClassPath** The path to the JDBC driver. Specify only the path to the driver file; do not include the file name in the path. This parameter is required. **- dbNodeName** The DB2 node name (this must be 8 characters or less). This node must be already catalogued and configured to communicate with the DB2 server. This parameter must be set if the current machine is configured as a DB2 client and the parameter **createDB** is set to `true`.	

			- dbHostName The host name of the machine where the database server is installed. This parameter is required. **- dbPort** DB2 instance port. The default value is 50 000 if not specified. **- dbName** The database name to be created. The default value is `event` if not specified. **- dbUser** DB2 user ID that has privileges to create and drop the databases. The default value is `db2inst1` if not specified. **- dbPassword** DB2 password. This parameter is required. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/db2 if this parameter is not specified. • Returns: None	

configEvent ServiceDB2 iSeriesDB	The **configEvent ServiceDB2 iSeriesDB** command generates the DDL database scripts, creates the event service database for DB2 iSeries on the native platform, and creates data sources on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** When this parameter is set to `true`, the command removes any existing event service data source at the specified scope before creating a new one. When this parameter is set to `false`, the command does not create an event service data source at the specified scope if another event service data source is found at the same scope. The default value is `false` if not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventService DB2iSeriesDB {createDB true -overrideDataSource true -nodeName nodename -serverName servername -dbUser db2user -dbPassword dbpassword -nativeJdbcClassPath /myDB2 ClassPath -collection event}` • Using Jython string: `AdminTask.configEventService DB2iSeriesDB('[-createDB true -overrideDataSource true -nodeName nodename -serverName servername -nativeJdbcClass Path /myDB2ClassPath -colle ction event]')` • Using Jython list: `AdminTask.configEventService DB2iSeriesDB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-nativeJdbcClassPath', '/myDB2ClassPath', '-collec tion', 'event'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventService DB2iSeriesDB -interactive` • Using Jython string: `AdminTask.configEventService DB2iSeriesDB('[-interactive]')` • Using Jython list: `AdminTask.configEventService DB2iSeriesDB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- toolboxJdbcClassPath** The path to the IBM Toolbox for Java DB2 JDBC driver. Specify only the path to the driver file; do not include the file name. You must specify either this parameter or the **jdbcClassPath** parameter. **- nativeJdbcClassPath** The path to the DB2 for iSeries native JDBC driver. Specify only the path to the driver file; do not include the file name in the path. You must specify either this parameter or the **toolboxJdbcClassPath** parameter.	

			- dbHostName The host name of the machine where the DB2 for iSeries database server is installed. This parameter is required if you are using the IBM Toolbox for Java DB2 JDBC driver. **- dbName** The DB2 for iSeries database name. The default value is `*LOCAL` if not specified. **- collection** DB2 for iSeries library SQL collection. The maximum length for the collection name is 10 characters. The default value is an empty string if not specified. **- dbUser** DB2 user ID that has privileges to create and drop the databases. This parameter is required. **- dbPassword** Password of the database user ID. This parameter is required. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/db2iseries if this parameter is not specified. • Returns: None	

configEvent ServiceDB 2ZOSDB	The **configEvent ServiceDB 2ZOSDB** command creates the event service database and data sources for DB2 z/OS on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** When this parameter is set to `true`, the command removes any existing event service data source at the specified scope before creating a new one. When this parameter is set to `false`, the command does not create an event service data source at the specified scope if another event service data source is found at the same scope. The default value is `false` if not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventService DB2ZOSDB {-createDB true -overrideDataSource true -nodeName nodename -server Name servername -jdbcClassPath c:\sqllib\java -dbUser db2user -dbPassword dbpassword -dbHost Name hostname -dbPort 5027 -dbSubSystemName db2zos81 -dbAliasName event -storageGroup sysdeflt -bufferPool4K BP9 -bufferPool8K BP8K9 -bufferPool 16K BP16K9}` `$AdminTask removeEventService DB2ZOSDB {-removeDB true -node Name nodename -serverName server name -dbAliasName event -dbUser db2user -dbPassword dbpassword }` • Using Jython string: `AdminTask.configEventServiceDB2 ZOSDB(["-createDB", "true", "-overrideDataSource", "true", "-nodeName", "nodename", "-server Name", "servername", "-jdbc ClassPath", "c:\sqllib\java", "-dbUser", "db2user", "-dbPassword", "dbpassword", "-dbHostName", "hostname", "-dbPort", "5027", "-dbSubSystemName", "db2zos81", "-dbAliasName", "event", "-storage Group", "sysdeflt", "-bufferPool 4K" "BP9", "-bufferPool8K", "BP8K9", "-bufferPool16K", "BP16K9"])` `AdminTask.removeEventServiceDB2 ZOSDB(["-removeDB", "true", "-nodeName", "nodename", "-server Name", "servername", "-dbAlias Name", "event", "-dbUser", "db2user", "-dbPassword", "dbpass word"])` • Using Jython list: `AdminTask.configEventServiceDB2 ZOSDB('[-createDB true -overrid eDataSource true -nodeName node name -serverName servername -jdbcClassPath c:\sqllib\java -dbUser db2user -dbPassword dbpassword -dbHostName hostname -dbPort 5027 -dbSubSystemName db2zos81 -dbAliasName event -storageGroup sysdeflt -buffer Pool4K BP9 -bufferPool8K BP8K9 -bufferPool16K BP16K9]')` `AdminTask.removeEventServiceDB2 ZOSDB('[-removeDB true -node Name nodename -serverName servername -dbAliasName event -dbUser db2user -dbPassword dbpassword]')`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- jdbcClassPath** The path to the JDBC driver. Specify only the path to the driver file; do not include the file name in the path. This parameter is required. **- dbHostName** The host name of the machine where the database server is installed. This parameter is required.	**Interactive mode example usage:** • Using Jacl: `$AdminTask configEventServ iceDB2ZOSDB -interactive` • Using Jython string: `AdminTask.configEventServ iceDB2ZOSDB('[-interactive]')` • Using Jython list: `AdminTask.configEventServi ceDB2ZOSDB ['-interactive'])`

			- dbPort DB2 for z/OS instance port. The default value is 5027 if not specified. **- dbAliasName** The name of the cataloged database on the DB2 client machine. This parameter is required when the createDB parameter is set to `true`. **- dbSubSystemName** The DB2 z/OS subsystem that you want to create. This parameter is required. **- eventDBName** The event database name that you want to create. The default value is `event` if you do not specify a value. **- eventCatalogDBName** The event catalog database that you want to create. The default value is `eventcat` if you do not specify a value. **- dbDiskSizeInMB** Specify the disk size in MB for the event service database. This value must be at least 10 MB. The default value is 100 MB if not specified.	

			- dbUser DB2 user ID that has privileges to create and drop the databases. This parameter is required. **- dbPassword** Password of the database user ID. This parameter is required. **- storageGroup** The storage group for the event database and the event catalog database. The storage group must already be created and active. **- bufferPool4K** The name of the 4K buffer pool. This buffer pool must be active before the database DDL scripts can be run. **- bufferPool8K** The name of the 8K buffer pool. This buffer pool must be active before the dabase DDL scripts can be run. **- bufferPool16K** The name of the 16K buffer pool. This buffer pool must be active before the database DDL scripts can be run. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/db2zos if this parameter is not specified. • Returns: None	

configEventServiceDerbyDB	The **configEventServiceDerbyDB** command creates the event service database and data sources for Derby on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** When this parameter is set to `true`, the command removes any existing event service data source at the specified scope before creating a new one. When this parameter is set to `false`, the command does not create an event service data source at the specified scope if another event service data source is found at the same scope. The default value is `false` if not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventServiceDerbyDB {-createDB true -overrideDataSource true -nodeName nodename -serverName servername}` • Using Jython string: `AdminTask.configEventServiceDerbyDB( '[-createDB true -overrideDataSource true -nodeName nodename -serverName servername]')` • Using Jython list: `AdminTask.configEventServiceDerbyDB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'nodename', '-serverName', 'servername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventServiceDerbyDB -interactive` • Using Jython string: `AdminTask.configEventServiceDerbyDB('[-interactive]')` • Using Jython list: `AdminTask.configEventServiceDerbyDB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- dbHostName** The host name of the network Derby database. To create the Derby network data source, specify this parameter and the **dbPort** parameter. To create the Derby local data source, do not specify this parameter and the **dbPort** parameter. **- dbPort** The port number of the network Derby database. To create the Derby network data source, specify this parameter and the **dbHostName** parameter. To create the Derby local data source, do not specify this parameter and the **dbHostName** parameter.	

			- dbName The database name to be created. The default value is `event` if not specified. **- dbUser** The user ID used by the data source for the Derby database authentication. This parameter is optional when the WebSphere domain security is disabled. If you specify this parameter, you also must specify the **dbPassword** parameter. This parameter is required when the WebSphere domain security is enabled. **- dbPassword** The password used by the data source for the Derby database authentication. This parameter is optional when the WebSphere domain security is disabled. If you specify this parameter, you also must specify the **dbUser** parameter. This parameter is required when the WebSphere domain security is enabled. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/derby if this parameter is not specified. • Returns: None	

configEventServiceInformixDB	The **configEventServiceInformixDB** command creates the event service database and data sources for Informix on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/*dbtype* if this parameter is not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventService InformixDB {-createDB true -overrideDataSource true -nodeName nodename -serverName servername -jdbcClassPath "c:\program files\ibm\informix\jdbc\lib" -dbInformixDir "c:\program files\ibm\informix" -dbUser informix -dbPassword dbpassword -dbHostName hostname -dbPort 1526 -dbServerName ol_server }` • Using Jython string: `AdminTask.configEventService InformixDB('[-createDB true -overrideDataSource true -nodeName nodename -serverName servername -jdbcClassPath "c:\program files\ibm\informix\jdbc\lib" -dbInformixDir "c:\program files\ibm\informix" -dbUser informix -dbPassword dbpassword -dbHostName hostname -dbPort 1526 -dbServerName ol_server]')` • Using Jython list: `AdminTask.configEventService InformixDB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-jdbcClassPath', 'c:\program files\ibm\informix\jdbc\lib', '-dbInformixDir', 'c:\program files\ibm\informix', '-dbUser', 'informix ', '-dbPassword', 'dbpassword', '-dbHostName', 'hostname', '-dbPort', '1526', '-dbServerName', 'ol_server'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventServiceInformixDB -interactive` • Using Jython string: `AdminTask.configEventServiceInformixDB('[-interactive]')` • Using Jython list: `AdminTask.configEventServiceInformixDB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- jdbcClassPath** The path to the JDBC driver. Specify only the path to the driver file; do not include the file name in the path. This parameter is required. **- dbInformixDir** The directory where the Informix database is installed. This parameter must be specified when the parameter **createDB** is set to `true`. This parameter is required.	

			- dbHostName The host name of the machine where the database server is installed. This parameter is required. **- dbServerName** Informix server instance name (for example, `ol_servername`). This parameter is required. **- dbPort** Informix instance port. The default value is 1526 if not specified. **- dbName** The database name to be created. The default value is `event` if not specified. **- dbUser** The Informix database schema user ID that will own the event service database tables. The WebSphere data source uses this user ID to authenticate the Informix database connection. This parameter is required. **- dbPassword** The password of the schema user ID that owns the event service Informix tables. The WebSphere data source uses this password to authenticate the Informix database connection. This parameter is required.	

			- ceiInstancePrefix The command uses the event service instance name to group the database files in a directory with unique names. The default value is `ceiinst1` if not specified. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/informix if this parameter is not specified. • Returns: None	

configEventServiceOracleDB	The **configEventServiceOracleDB** command creates the event service tables and data sources for Oracle on a server or cluster. The command does not create the database; the Oracle SID must already exist.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/*dbtype* if this parameter is not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventServiceOracleDB {-createDB true -overrideDataSource true -nodeName nodename -serverName servername -jdbcClassPath c:\oracle\ora92\jdbc\lib -oracleHome c:\oracle\ora92 -dbUser ceiuser -dbPassword ceipassword -dbHostName hostname -dbPort 1521 -sysUser sys -sysPassword syspassword}` • Using Jython string: `AdminTask.configEventServiceOracleDB('[-createDB true -overrideDataSource true -nodeName nodename -serverName servername -jdbcClassPath c:\oracle\ora92\jdbc\lib -oracleHome c:\oracle\ora92 -dbUser ceiuser -dbPassword ceipassword -dbHostName hostname -dbPort 1521 -sysUser sys -sysPassword syspassword]')` • Using Jython list: `AdminTask.configEventServiceOracleDB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-jdbcClassPath', 'c:\oracle\ora92\jdbc\lib', '-oracleHome', 'c:\oracle\ora92', '-dbUser', 'ceiuser', '-dbPassword', 'ceipassword', '-dbHostName', 'hostname', '-dbPort', '1521', '-sysUser', 'sys', '-sysPassword', 'syspassword'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventServiceOracleDB -interactive` • Using Jython string: `AdminTask.configEventServiceOracleDB('[-interactive]')` • Using Jython list: `AdminTask.configEventServiceOracleDB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- jdbcClassPath** The path to the JDBC driver. Specify only the path to the driver file; do not include the file name in the path. This parameter is required. **- oracleHome** The ORACLE_HOME directory. This parameter must be set when the parameter **createDB** is set to `true`.	

			- dbHostName The host name of the machine where the Oracle database server is installed. The default value is `localhost` if not specified. **- dbPort** Oracle instance port. The default value is 1521 if not specified. **- dbName** The Oracle system identifier (SID). The SID must already exist and must be available for the event service command to create the tables and populate the tables with data. The default value is `orcl` if not specified. **- dbUser** The Oracle schema user ID that will own the event service Oracle tables. The user ID will be created during the database creation; the WebSphere data source uses this user ID to authenticate the Oracle database connection. The default value is `ceiuser` if not specified. **- dbPassword** The password of the schema user ID. The password will be created during the database creation; the WebSphere data source uses this password to authenticate the Oracle database connection. This parameter is required.	

			- sysUser Oracle sys user ID. This must be a user that has SYSDBA privileges. The default value is `sys` if not specified. **- sysPassword** The password for the user specified by the **sysUser** parameter. The default value is an empty string if not specified. **- ceiInstancePrefix** The command uses the event service instance name to group the database files in a directory with unique names. The default value is `ceiinst1` if not specified. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/oracle if this parameter is not specified. • Returns: None	

configEvent ServiceSQL ServerDB	The **configEvent ServiceSQL ServerDB** command creates the event service database and data sources for SQL Server on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/*dbtype* if this parameter is not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventService SQLServerDB {-createDB true -overrideDataSource true -nodeName nodename -server Name servername -dbUser cei user -dbPassword ceipassword -dbServerName sqlservername -dbHostName hostname -dbPort 1433 -saUser sa -saPassword sapassword}` • Using Jython string: `AdminTask.configEventService SQLServerDB('[-createDB true -overrideDataSource true -nodeName nodename -server Name servername -dbUser cei user -dbPassword ceipassword -dbServerName sqlservername -dbHostName hostname -dbPort 1433 -saUser sa -saPassword sapassword]')` • Using Jython list: `AdminTask.configEventService SQLServerDB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'node name', '-serverName', 'server name', '-dbUser', 'ceiuser', '-dbPassword', 'ceipassword', '-dbServerName', 'sqlserver name', '-dbHostName', 'host name', '-dbPort', '1433', '-saUser', 'sa', '-saPassword', 'sapassword'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventServi ceSQLServerDB -interactive` • Using Jython string: `AdminTask.configEventServi ceSQLServerDB('[-interactive]')` • Using Jython list: `AdminTask.configEventService SQLServerDB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- dbServerName** The server name of the SQL Server database. This parameter must be set when the parameter **createDB** is set to `true`. **- dbHostName** The host name of the machine where the SQL Server database is running.	

			- dbPort SQL Server instance port. The default value is 1433 if not specified. **- dbName** The database name to be created. The default value is `event` if not specified. **- dbUser** The SQL Server user ID that will own the event service tables. The default value is `ceiuser` if not specified. **- dbPassword** The password of the SQL Server user ID specified by the **dbUser** parameter. This parameter is required. **- saUser** User ID that has privileges to create and drop databases and users. This parameter is required when the **createDB** parameter is set to `true`. The default value is `sa` if not specified. **- saPassword** The sa password. You must not specify this parameter if the sa user ID does not have a password.	

			- ceiInstancePrefix The command uses the event service instance name to group the database files in a directory with unique names. The default value is `ceiinst1` if not specified. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/sqlserver if this parameter is not specified. • Returns: None	

configEventServiceSybaseDB	The **configEventServiceSybaseDB** command creates the event service database and data sources for Sybase on a server or cluster.	None	• Parameters: **- createDB** The command generates the DDL database scripts and creates the database when this parameter is set to `true`. The command only generates the DDL database scripts when this parameter is set to `false`. To create the database, the current machine must be already configured to run the database commands. The default value is `false` if not specified. **- overrideDataSource** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/*dbtype* if this parameter is not specified. **- nodeName** The name of the node that contains the server where the event service data source should be created. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask configEventService SybaseDB {-createDB true -overrideDataSource true -nodeName nodename -server Name servername -dbUser cei user -dbPassword ceipassword -createLogin true -dbServer Name sybaseservername -dbHostName hostname -dbPort 5000 -saUser sa -saPassword sapassword -firstDeviceNumber 10 -dbCacheSizeInMB 10 -dbDishSizeInMB 100 -dbInstall Dir c:\sybase}` • Using Jython string: `AdminTask.configEventService SybaseDB('[-createDB true -overrideDataSource true -node Name nodename -serverName servername -dbUser ceiuser -dbPassword ceipassword -createLogin true -dbServer Name sybaseservername -dbHost Name hostname -dbPort 5000 -saUser sa -saPassword sapass word -firstDeviceNumber 10 -dbCacheSizeInMB 10 -dbDish SizeInMB 100 -dbInstallDir c:\sybase]')` • Using Jython list: `AdminTask.configEventService SybaseDB(['-createDB', 'true', '-overrideDataSource', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-dbUser', 'ceiuser', '-dbPass word', 'ceipassword', '-create Login', 'true', '-dbServerName', 'sybaseservername', '-dbHost Name', 'hostname', '-dbPort', '5000', '-saUser', 'sa', '-saPassword', 'sapassword', '-firstDeviceNumber', '10', '-dbCacheSizeInMB', '10', '-dbDishSizeInMB', '100', '-dbInstallDir', 'c:\sybase'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask configEventSer viceSybaseDB -interactive` • Using Jython string: `AdminTask.configEventServi ceSybaseDB('[-interactive]')` • Using Jython list: `AdminTask.configEventServi ceSybaseDB(['-interactive'])`

			- serverName The name of the server where the event service data source should be created. If this parameters is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be created. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- jdbcClassPath** The path to the JDBC driver. Specify only the path to the driver file; do not include the file name in the path. This parameter is required. **- dbInstallDir** The directory where the Sybase database is installed. This parameter must be set when the parameter **createDB** is set to `true`.	

			- dbServerName The name of the Sybase server instance. The server is defined in the Sybase configuration. This parameter is required. **- dbHostName** The host name of the machine where Sybase is running. The default value is `localhost` if not specified. **- dbPort** Sybase instance port. The default value is 5000 if not specified. **- dbName** The database name to be created. The default value is `event` if not specified. **- firstDeviceNumber** The event database creates six devices. This parameter identifies the value of the first device number that should be assigned to the new devices. This parameter must be set when the parameter **createDB** is set to `true`. The default value is 10 if not specified. **- dbCacheSizeInMB** The memory cache size will be used for transaction logs. This parameter must be set when the parameter **createDB** is set to `true`. The lowest valid value is 10. The default value is 10 MB if not specified.	

			- dbDiskSizeInMB The database size in MB to be created for the event service. This parameter must be set when the parameter **createDB** is set to `true`. The lowest valid value is 100. The default value is 100 MB if not specified. **- dbUser** The user ID that will own the event service Sybase tables. The WebSphere data source uses this user ID to authenticate the Sybase database connection. The default value is `ceiuser` if not specified. **- dbPassword** The password of the user ID that owns the event service Sybase tables. The WebSphere data source uses this password to authenticate the Sybase database connection. This parameter is required. **- createLogin** The **configEventServiceSybaseDB** command creates the login user ID that will own the event service Sybase tables when this parameter is set to `true`. The command does not create the user ID when the parameter is set to `false`. The default value is `true` if not specified.	

			- saUser The Sybase sa userid that has privileges to create tables and users. The default value is `sa` if not specified. **- saPassword** The password of the Sybase sa user ID. You must not specify this parameter if the sa user ID does not have a password. **- ceiInstancePrefix** The command uses the event service instance name to group the database files in a directory with unique names. The default value is `ceiinst1` if not specified. **- outputScriptDir** Optional database script output directory. When this parameter is specified, the command generates the event service database scripts in the specified directory. If the specified directory does not contain a full path, the command creates the specified directory in *app_server_root*/profiles/*profile_name*/bin. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/sybase if this parameter is not specified. • Returns: None	

removeEventServiceDB2DB	The **removeEventServiceDB2DB** command removes the event service database and data sources for DB2 from a server or cluster.	None	• Parameters: **- removeDB** The command removes the database when this parameter is set to `true` and does not remove the database when set to `false`. To remove the database, the current machine must already configured to run the database commands. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventServiceDB2DB {-removeDB true -nodeName nodename -serverName servername -dbUser db2inst1 -dbPassword dbpassword }` • Using Jython string: `AdminTask.removeEventServiceDB2DB('[-removeDB true -nodeName nodename -serverName servername -dbUser db2inst1 -dbPassword dbpassword]')` • Using Jython list: `AdminTask.removeEventServiceDB2DB(['-removeDB', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-dbUser', 'db2inst1', '-dbPassword', 'dbpassword'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServiceDB2DB -interactive` • Using Jython string: `AdminTask.removeEventServiceDB2DB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServiceDB2DB(['-interactive'])`

			- dbUser DB2 user ID that has privileges to create and drop the databases. This parameter must be set when the parameter **removeDB** is set to `true`. The default value is `db2inst1` if not specified. **- dbPassword** DB2 password. This parameter must be set when the parameter **removeDB** is set to true. **- dbScriptDir** The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/db2. • Returns: None	

removeEventServiceDB2iSeriesDB	The **removeEventServiceDB2iSeriesDB** command removes the DB2 for iSeries data sources from a server or cluster. The database must be removed manually.	None	• Parameters: **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventServiceDB2iSeriesDB {-nodeName nodename -serverName servername }` • Using Jython string: `AdminTask.removeEventServiceDB2iSeriesDB('[-nodeName nodename -serverName servername]')` • Using Jython list: `AdminTask.removeEventServiceDB2iSeriesDB(['-nodeName', 'nodename', '-serverName', 'servername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServiceDB2iSeriesDB -interactive` • Using Jython string: `AdminTask.removeEventServiceDB2iSeriesDB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServiceDB2iSeriesDB(['-interactive'])`

removeEventServiceDB2ZOSDB	The **removeEventServiceDB2ZOSDB** command removes the event service database and data sources for DB2 z/OS from a server or cluster.	None	• Parameters: **- removeDB** The command removes the database when this parameter is set to `true` and does not remove the database when set to `false`. To remove the database, the current machine must already configured to run the database commands. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventService DB2ZOSDB {-removeDB true -nodeName nodename -server Name servername -dbUser db2user -dbPassword dbpassword` • Using Jython string: `AdminTask.removeEventService DB2ZOSDB('[-removeDB true -nodeName nodename -server Name servername -dbUser db2user -dbPassword dbpass word]')` • Using Jython list: `AdminTask.removeEventService DB2ZOSDB(['-removeDB', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-dbUser', 'db2user', '-dbPass word', 'dbpassword'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServ iceDB2ZOSDB -interactive` • Using Jython string: `AdminTask.removeEventServi ceDB2ZOSDB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServi ceDB2ZOSDB(['-interactive'])`

			- dbName The DB2 database name. On the DB2 client machine, it is the name of the catalogued database. On the native z/OS machine, it is the name of the database subsystem. This parameter must be set when the parameter **removeDB** is set to `true`. The default value is `event` if not specified. **- dbUser** DB2 user ID that has privileges to create and drop the databases. This parameter must be set when the parameter **removeDB** is set to true. **- dbPassword** DB2 password. This parameter must be set when the parameter **removeDB** is set to true. **- dbScriptDir** The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/db2zos. • Returns: None	

removeEventServiceDerbyDB	The **removeEventServiceDerbyDB** command removes the Event Service database and data source for Derby from a server or cluster.	None	• Parameters: **- removeDB** The command removes the database when this parameter is set to `true` and does not remove the database when set to `false`. To remove the database, the current machine must already configured to run the database commands. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventServiceDerbyDB {-removeDB true -nodeName nodename -serverName servername}` • Using Jython string: `AdminTask.removeEventServiceDerbyDB('[-removeDB true -nodeName nodename -serverName servername]')` • Using Jython list: `AdminTask.removeEventServiceDerbyDB(['-removeDB', 'true', '-nodeName', 'nodename', '-serverName', 'servername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServiceDerbyDB -interactive` • Using Jython string: `AdminTask.removeEventServiceDerbyDB( '[-interactive]')` • Using Jython list: `AdminTask.removeEventServiceDerbyDB(['-interactive'])`

			- clusterName The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- dbScriptDir** The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/derby. • Returns: None	

removeEventServiceInformixDB	The **removeEventServiceInformixDB** command removes the event service database and data sources for Informix from a server or cluster.	None	• Parameters: **- removeDB** The command removes the database when this parameter is set to `true` and does not remove the database when set to `false`. To remove the database, the current machine must already configured to run the database commands. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventService InformixDB {-removeDB true -nodeName nodename -server Name servername}` • Using Jython string: `AdminTask.removeEventService InformixDB('[-removeDB true -nodeName nodename -server Name servername]')` • Using Jython list: `AdminTask.removeEventService InformixDB(['-removeDB', 'true', '-nodeName', 'node name', '-serverName', 'servername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServi ceInformixDB -interactive` • Using Jython string: `AdminTask.removeEventServi ceInformixDB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServi ceInformixDB(['-interactive'])`

			- clusterName The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- dbScriptDir** The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/informix. • Returns: None	

removeEventServiceOracleDB	The **removeEventServiceOracleDB** command removes the event service tables and data sources for Oracle from a server or cluster. The command does not remove the database.	None	• Parameters: **- removeDB** The command removes the event service tables when this parameter is set to `true` and does not remove the tables when set to `false`. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventServiceOracleDB {-removeDB true -nodeName nodename -serverName servername -sysUser sys -sysPassword syspassword}` • Using Jython string: `AdminTask.removeEventServiceOracleDB('[-removeDB true -nodeName nodename -serverName servername -sysUser sys -sysPassword syspassword]')` • Using Jython list: `AdminTask.removeEventServiceOracleDB(['-removeDB', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-sysUser', 'sys', '-sysPassword', 'syspassword'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServiceOracleDB -interactive` • Using Jython string: `AdminTask.removeEventServiceOracleDB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServiceOracleDB(['-interactive'])`

			- clusterName The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- sysUser** Oracle database sys user ID. The default value is `sys` if not specified. **- sysPassword** The password for the user specified by the sysUser parameter. **- dbScriptDir** The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/oracle. • Returns: None	

removeEventServiceSQLServerDB	The removeEventServiceOra cleDB command removes the event service database and data sources for SQL Server from a server or cluster.	None	• Parameters: **- removeDB** The command removes the database when this parameter is set to `true` and does not remove the database when set to `false`. To remove the database, the current machine must already configured to run the database commands. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventService SQLServerDB {-removeDB true -nodeName nodename -serverName servername -dbUser ceiuser -saUser sa -saPassword sapassword -dbServerName sqlservername}` • Using Jython string: `AdminTask.removeEventService SQLServerDB('[-removeDB true -nodeName nodename -serverName servername -dbUser ceiuser -saUser sa -saPassword sapassword -dbServerName sqlservername]')` • Using Jython list: `AdminTask.removeEventService SQLServerDB(['-removeDB', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-dbUser', 'ceiuser', '-saUser','sa', '-saPassword', 'sapassword', '-dbServerName', 'sqlservername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServiceSQLServerDB -interactive` • Using Jython string: `AdminTask.removeEventServiceSQLServerDB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServiceSQLServerDB(['-interactive'])`

			- clusterName The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- dbServerName** The server name of the SQL Server database. This parameter must be set when the parameter **removeDB** is set to `true`. **- dbUser** SQL Server user ID that owns the event service tables. The default value is `ceiuser` if not specified. **- saUser** User ID that has privileges to drop the databases and users. The default value is `sa` if not specified. **- saPassword** The password specified by the **saUser** parameter. This parameter is required when the parameter **removeDB** is set to `true`.	
			- dbScriptDir The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/sqlserver. • Returns: None	

removeEventServiceSybaseDB	The **removeEventServiceSybaseDB** command removes the event service database and data sources for Sybase from a server or cluster.	None	• Parameters: **- removeDB** The command removes the database when this parameter is set to `true` and does not remove the database when set to `false`. To remove the database, the current machine must already configured to run the database commands. **- nodeName** The name of the node that contains the server where the event service data source should be removed. If this parameter is specified, then the **serverName** parameter must be set. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service data source should be removed. If this parameter is specified without the **nodeName** parameter, the command will use the node name of the current WebSphere profile. You must not specify this parameter if the **clusterName** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventService SybaseDB {-removeDB true -nodeName nodename -server Name servername -dbUser cei user -removeLogin true -saUser sa -saPassword sapassword -dbServerName sybaseservername}` • Using Jython string: `AdminTask.removeEventService SybaseDB('[-removeDB true -nodeName nodename -server Name servername -dbUser cei user -removeLogin true -sa User sa -saPassword sapass word -dbServerName sybase servername]')` • Using Jython list: `AdminTask.removeEventService SybaseDB(['-removeDB', 'true', '-nodeName', 'nodename', '-serverName', 'servername', '-dbUser', 'ceiuser', '-remove Login', 'true', '-saUser', 'sa', '-saPassword', 'sapass word', '-dbServerName', 'sybaseservername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServ iceSybaseDB -interactive` • Using Jython string: `AdminTask.removeEventServ iceSybaseDB('[-interactive]')` • Using Jython list: `AdminTask.removeEventServi ceSybaseDB(['-interactive'])`

			- clusterName The name of the cluster where the event service data source should be removed. If this parameter is specified, then the **serverName** and **nodeName** parameters must not be set. You must not specify this parameter if the **serverName** and **nodeName** parameters are specified. **- dropLogin** The **removeEventServiceSybaseDB** command drops the user ID that owns the event service tables when this parameter is set to `true`. The command will not drop the user ID when this parameter is set to `false`. This parameter must be set when the parameter **removeDB** is set to `true`. The default value is `true` if not specified. **- dbServerName** The name of the Sybase server instance. The server is defined in the Sybase configuration. This parameter is required when the **removeDB** parameter is set to `true`.	

			- dbUser The user ID that owns the event service tables. The default value is `ceiuser` if not specified. **- saUser** The Sybase sa userid that has privileges to drop tables and users. The default value is `sa` if not specified. **- saPassword** The password of the Sybase sa user ID. You must not specify this parameter if the sa user ID does not have a password. **- dbScriptDir** The directory containing the database scripts generated by the event service database configuration command. If specified, the command will run the scripts in this directory to remove the event service database. The default database script output directory is *app_server_root*/profiles/*profile*/databases/event/*node*/*server*/dbscripts/sybase. • Returns: None	

Commands for the JDBCProviderManagement group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the JDBCProviderManagement group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

create Datasource	The **create Datasource** command creates a new data source to access the back end data store. Application components use the data source to access connection instances to your database.	JDBC Provider Object ID - The configuration object of the JDBC provider to which the new data source will belong.	• Parameters: **- name** The name of the data source. (String, required) **- jndiName** The Java Naming and Directory Interface (JNDI) name. (String, required) **- description** The description of the data source. (String, optional) **- category** The category that you can use to classify a group of data sources. (String, optional) **- dataStoreHelperClassName** The name of the DataStoreHelper implementation class that extends the capabilities of the selected JDBC driver implementation class to perform data-specific functions. WebSphere Application Server provides a set of DataStoreHelper implementation classes for each of the JDBC provider drivers it supports. (String, required) **- componentManagedAuthenticationAlias** The alias used for database authentication at run time. This alias is only used when the application resource reference is using res-auth=Application. (String, optional) **- containerManagedPersistence** Specifies if the data source is used for container managed persistence for enterprise beans. The default value is true. (Boolean, optional)	**Interactive mode example usage:** • Using Jacl: `$AdminTask createDatasource {-interactive}` • Using Jython string: `AdminTask.createDatasource ('[-interactive]')` • Using Jython list: `AdminTask.createDatasource (['-interactive'])`

			• **Parameters for step one:** ***configureResourceProperties*** This command step configures the resource properties that are required by the data source. (Required) You can specify the following parameters for this step: **name** The name of the resource property. (String, required) **type** The type of the resource property. (String, required) **value** The value of the resource property. (String, required) • Returns: The object ID of the created data source configuration object.	

createJDBC Provider	The **createJDBC Provider** command creates a new Java database connectivity provider (JDBC) that you can use to connect to a relational database for data access.	None	• Parameters: **- scope** The scope for the new JDBC provider. The default is `none`. (String, required) **- databaseType** The type of database that will be used by the JDBC provider. For example, DB2, Derby, Informix, Oracle, and so on. (String, required) **- providerType** The JDBC provider type that will be used by the JDBC provider. (String, required) **- implementationType** The implementation type for this JDBC provider. Use `Connection pool data source` if your application runs in a single phase or a local transaction. Otherwise, use `XA data source` to run in a global transaction. (String, required) **- name** The name of the JDBC provider. The default is the value from the provider template. (String, optional) **- description** The description for the JDBC provider. (String, optional) **- implementationClassName** Specifies the Java class name for the JDBC driver implementation. (String, optional) **- classpath** Specifies a list of paths or JAR file names that form the location for the resource provider classes. (String, optional)	**Interactive mode example usage:** • Using Jacl: `$AdminTask createJDBCProvider {-interactive}` • Using Jython string: `AdminTask.createJDBCProvider ('[-interactive]')` • Using Jython list: `AdminTask.createJDBCProvider (['-interactive'])`

			- nativePath Specifies a list of paths that form the location for the resource provider native libraries. (String, optional) • Returns: The ID of the JDBC provider object that you created.	
listDatasources	Use the **listData sources** command to list data sources that are contained in the specified scope.	None	• Parameters: **- scope** The scope for the data sources that will be listed. The valid values include cell, node, server, or cluster. (String, optional) • Returns: A list of data sources.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listDatasources {-interactive}` • Using Jython string: `AdminTask.listDatasources ('[-interactive]')` • Using Jython list: `AdminTask.listDatasources (['-interactive'])`
listJDBC Providers	The **listJDBC Providers** command lists JDBC providers that are contained in the specified scope.	None	• Parameters: **- scope** The scope for the JDBC providers that will be listed. The valid values include cell, node, server, or cluster. (String, optional) • Returns: A list of JDBC providers.	**Interactive mode example usage:** • Using Jacl: `$AdminTask listJDBCProviders {-interactive}` • Using Jython string: `AdminTask.listJDBCProviders ('[-interactive]')` • Using Jython list: `AdminTask.listJDBCProviders (['-interactive'])`

Configuring messaging with scripting

Use these topics to learn about configuring messaging with scripting and the wsadmin tool.

This topic contains the following tasks:

Configuring the message listener service using scripting

Use scripting to configure the message listener service.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure the message listener service for an application server:

1. Identify the application 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 message listener service belonging to the server and assign it to the mls variable:
   - Using Jacl:

     ```
     set mls [$AdminConfig list MessageListenerService $server]
     ```

   - Using Jython:

     ```
     mls = AdminConfig.list('MessageListenerService', server)
     print mls
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode/servers/server1|server.xml#MessageListenerService_1)
   ```

3. Modify various attributes with one of the following examples:
   - This example command changes the thread pool attributes:
     - Using Jacl:

       ```
       $AdminConfig modify $mls {{threadPool {{inactivityTimeout 4000}
       {isGrowable true} {maximumSize 100} {minimumSize 25}}}}
       ```

     - Using Jython:

       ```
       AdminConfig.modify(mls, [['threadPool', [['inactivityTimeout', 4000],
       ['isGrowable',  'true'], ['maximumSize', 100], ['minimumSize', 25]]]])
       ```

   - This example modifies the property of the first listener port:
     - Using Jacl:

       ```
       set lports [$AdminConfig showAttribute $mls listenerPorts]
       set lport [lindex $lports 0]
       $AdminConfig modify $lport {{maxRetries 2}}
       ```

     - Using Jython:

       ```
       lports = AdminConfig.showAttribute(mls, 'listenerPorts')
       cleanLports = lports[1:len(lports)-1]
       lport = cleanLports.split(" ")[0]
       AdminConfig.modify(lport, [['maxRetries', 2]])
       ```

   - This example adds a listener port:
     - Using Jacl:

       ```
       set new [$AdminConfig create ListenerPort $mls {{name my}
       {destinationJNDIName di} {connectionFactoryJNDIName jndi/fs}}]
       $AdminConfig create StateManageable $new {{initialState START}}
       ```

     - Using Jython:

       ```
       new = AdminConfig.create('ListenerPort', mls, [['name', 'my'],
       ['destinationJNDIName', 'di'], ['connectionFactoryJNDIName', 'jndi/fsi']])
       print new
       print AdminConfig.create('StateManageable', new, [['initialState', 'START']])
       ```

     Example output:

     ```
     my(cells/mycell/nodes/mynode/servers/server1:server.xml#ListenerPort_1079471940692)
     (cells/mycell/nodes/mynode/servers/server1:server.xml#StateManageable_107947182623)
     ```

4. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
5. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new JMS providers using scripting

You can use the wsadmin tool and scripting to configure a new JMS provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new JMS provider:

1. Identify the parent ID:

- Using Jacl:

```
set node [$AdminConfig getid /Cell:mycell/Node:mynode/]
```

- Using Jython:

```
node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
print node
```

Example output:

```
mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
```

2. Get required attributes:
   - Using Jacl:

```
$AdminConfig required JMSProvider
```

   - Using Jython:

```
print AdminConfig.required('JMSProvider')
```

   Example output:

```
Attribute        Type
name     String
externalInitialContextFactory    String
externalProviderURL               String
```

3. Set up required attributes:
   - Using Jacl:

```
set name [list name JMSP1]
set extICF [list externalInitialContextFactory
"Put the external initial context factory here"]
set extPURL [list externalProviderURL "Put the external provider URL here"]
set jmspAttrs [list $name $extICF $extPURL]
```

   - Using Jython:

```
name = ['name', 'JMSP1']
extICF = ['externalInitialContextFactory',
"Put the external initial context factory here"]
extPURL = ['externalProviderURL', "Put the external provider URL here"]
jmspAttrs = [name, extICF, extPURL]
print jmspAttrs
```

   Example output:

```
{name JMSP1} {externalInitialContextFactory {Put the external
initial context factory here }} {externalProviderURL
{Put the external provider URL here}}
```

4. Create the JMS provider:
   - Using Jacl:

```
set newjmsp [$AdminConfig create JMSProvider $node $jmspAttrs]
```

   - Using Jython:

```
newjmsp = AdminConfig.create('JMSProvider', node, jmspAttrs)
print newjmsp
```

   Example output:

```
JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new JMS destinations using scripting

You can use scripting and the wsadmin tool to configure a new JMS destination.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new JMS destination:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1]
     ```
   - Using Jython:
     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:
   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```
2. Get required attributes:
   - Using Jacl:
     ```
     $AdminConfig required GenericJMSDestination
     ```
   - Using Jython:
     ```
     print AdminConfig.required('GenericJMSDestination')
     ```

   Example output:
   ```
   Attribute       Type
   name       String
   jndiName    String
   externalJNDIName  String
   ```
3. Set up required attributes:
   - Using Jacl:
     ```
     set name [list name JMSD1]
     set jndi [list jndiName jms/JMSDestination1]
     set extJndi [list externalJNDIName jms/extJMSD1]
     set jmsdAttrs [list $name $jndi $extJndi]
     ```
   - Using Jython:
     ```
     name = ['name', 'JMSD1']
     jndi = ['jndiName', 'jms/JMSDestination1']
     extJndi = ['externalJNDIName', 'jms/extJMSD1']
     jmsdAttrs = [name, jndi, extJndi]
     print jmsdAttrs
     ```

   Example output:
   ```
   {name JMSD1} {jndiName jms/JMSDestination1} {externalJNDIName jms/extJMSD1}
   ```
4. Create generic JMS destination:
   - Using Jacl:
     ```
     $AdminConfig create GenericJMSDestination $newjmsp  $jmsdAttrs
     ```
   - Using Jython:
     ```
     print AdminConfig.create('GenericJMSDestination', newjmsp,  jmsdAttrs)
     ```

   Example output:
   ```
   JMSD1(cells/mycell/nodes/mynode|resources.xml#GenericJMSDestination_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new JMS connections using scripting

Use scripting and the wsadmin tool to configure a new JMS connection.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new JMS connection:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:myNode/JMSProvider:JMSP1]
     ```
   - Using Jython:
     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:
   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```
2. Get required attributes:
   - Using Jacl:
     ```
     $AdminConfig required GenericJMSConnectionFactory
     ```
   - Using Jython:
     ```
     print AdminConfig.required('GenericJMSConnectionFactory')
     ```

   Example output:
   ```
   Attribute       Type
   name       String
   jndiName    String
   externalJNDIName  String
   ```
3. Set up required attributes:
   - Using Jacl:
     ```
     set name [list name JMSCF1]
     set jndi [list jndiName jms/JMSConnFact1]
     set extJndi [list externalJNDIName jms/extJMSCF1]
     set jmscfAttrs [list $name $jndi $extJndi]
     ```

     Example output:
     ```
     {name JMSCF1} {jndiName jms/JMSConnFact1} {externalJNDIName jms/extJMSCF1}
     ```
   - Using Jython:
     ```
     name = ['name', 'JMSCF1']
     jndi = ['jndiName', 'jms/JMSConnFact1']
     extJndi = ['externalJNDIName', 'jms/extJMSCF1']
     jmscfAttrs = [name, jndi, extJndi]
     print jmscfAttrs
     ```

     Example output:
     ```
     [[name, JMSCF1], [jndiName, jms/JMSConnFact1], [externalJNDIName, jms/extJMSCF1]]
     ```
4. Create generic JMS connection factory:
   - Using Jacl:
     ```
     $AdminConfig create GenericJMSConnectionFactory $newjmsp $jmscfAttrs
     ```
   - Using Jython:
     ```
     print AdminConfig.create('GenericJMSConnectionFactory', newjmsp, jmscfAttrs)
     ```

   Example output:
   ```
   JMSCF1(cells/mycell/nodes/mynode|resources.xml#GenericJMSConnectionFactory_1)
   ```
5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WebSphere queue connection factories using scripting

You can use scripting and the wsadmin tool to configure new queue connection factories in WebSphere Application Server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new WebSphere queue connection factory:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASQueueConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('WASQueueConnectionFactory')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name WASQCF]
     set jndi [list jndiName jms/WASQCF]
     set mqcfAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name WASQCF} {jndiName jms/WASQCF}
     ```

   - Using Jython:

     ```
     name = ['name', 'WASQCF']
     jndi = ['jndiName', 'jms/WASQCF']
     mqcfAttrs = [name, jndi]
     print mqcfAttrs
     ```

     Example output:

     ```
     [[name, WASQCF], [jndiName, jms/WASQCF]]
     ```

4. Create was queue connection factories:
   - Using Jacl:

     ```
     $AdminConfig create WASQueueConnectionFactory $newjmsp $mqcfAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASQueueConnectionFactory', newjmsp, mqcfAttrs)
     ```

   Example output:

   ```
   WASQCF(cells/mycell/nodes/mynode|resources.xml#WASQueueConnectionFactory_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.

6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WebSphere topic connection factories using scripting

Use scripting and the wsadmin tool to configure new WebSphere topic connection factories.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new WebSphere topic connection factory:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```
   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```
2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASTopicConnectionFactory
     ```
   - Using Jython:

     ```
     print AdminConfig.required('WASTopicConnectionFactory')
     ```

   Example output:

   ```
   Attribute            Type
   name                 String
   jndiName             String
   port                 ENUM(DIRECT, QUEUED)
   ```
3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name WASTCF]
     set jndi [list jndiName jms/WASTCF]
     set port [list port QUEUED]
     set mtcfAttrs [list $name $jndi $port]
     ```

     Example output:

     ```
     {name WASTCF} {jndiName jms/WASTCF} {port QUEUED}
     ```
   - Using Jython:

     ```
     name = ['name', 'WASTCF']
     jndi = ['jndiName', 'jms/WASTCF']
     port = ['port', 'QUEUED']
     mtcfAttrs = [name, jndi, port]
     print mtcfAttrs
     ```

     Example output:

     ```
     [[name, WASTCF], [jndiName, jms/WASTCF], [port, QUEUED]]
     ```
4. Create was topic connection factories:
   - Using Jacl:

     ```
     $AdminConfig create WASTopicConnectionFactory $newjmsp $mtcfAttrs
     ```
   - Using Jython:

     ```
     print AdminConfig.create('WASTopicConnectionFactory', newjmsp, mtcfAttrs)
     ```

   Example output:

   ```
   WASTCF(cells/mycell/nodes/mynode|resources.xml#WASTopicConnectionFactory_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WebSphere queues using scripting

You can use scripting to configure a new WebSphere queue.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new WebSphere queue:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASQueue
     ```

   - Using Jython:

     ```
     print AdminConfig.required('WASQueue')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name WASQ1]
     set jndi [list jndiName jms/WASQ1]
     set wqAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name WASQ1} {jndiName jms/WASQ1}
     ```

   - Using Jython:

     ```
     name = ['name', 'WASQ1']
     jndi = ['jndiName', 'jms/WASQ1']
     wqAttrs = [name, jndi]
     print wqAttrs
     ```

     Example output:

     ```
     [[name, WASQ1], [jndiName, jms/WASQ1]]
     ```

4. Create was queue:
   - Using Jacl:

     ```
     $AdminConfig create WASQueue $newjmsp $wqAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASQueue', newjmsp, wqAttrs)
     ```

   Example output:

   ```
   WASQ1(cells/mycell/nodes/mynode|resources.xml#WASQueue_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new WebSphere topics using scripting

You can configure new WebSphere topics using the wsadmin tool and scripting.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new WebSphere topic:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1/')
     print newjmsp
     ```

     Example output:

     ```
     JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
     ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required WASTopic
     ```

   - Using Jython:

     ```
     print AdminConfig.required('WASTopic')
     ```

     Example output:

     ```
     Attribute       Type
     name       String
     jndiName    String
     topic        String
     ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name WAST1]
     set jndi [list jndiName jms/WAST1]
     set topic [list topic "Put your topic here"]
     set wtAttrs [list $name $jndi $topic]
     ```

     Example output:

     ```
     {name WAST1} {jndiName jms/WAST1} {topic {Put your topic here}}
     ```

   - Using Jython:

     ```
     name = ['name', 'WAST1']
     jndi = ['jndiName', 'jms/WAST1']
     topic = ['topic', "Put your topic here"]
     wtAttrs = [name, jndi, topic]
     print wtAttrs
     ```

     Example output:

     ```
     [[name, WAST1], [jndiName, jms/WAST1], [topic, "Put your topic here"]]
     ```

4. Create was topic:
   - Using Jacl:

     ```
     $AdminConfig create WASTopic $newjmsp $wtAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('WASTopic', newjmsp, wtAttrs)
     ```

Example output:

```
WAST1(cells/mycell/nodes/mynode|resources.xml#WASTopic_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new MQ connection factories using scripting

You can use scripting to configure a new MQ connection factory.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new MQ connection factory:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:WebSphere MQ JMS Provider/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:WebSphere MQ JMS Provider')
     print newjmsp
     ```

   Example output:

   ```
   WebSphere MQ JMS Provider(cells/mycell/nodes/mynode|resources.xml#builtin_mqprovider)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQConnectionFactory')
     ```

   Example output:

   ```
   ttribute       Type
   name           String
   jndiName          String
   connectionPool     ConnectionPool
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name MQCF]
     set jndi [list jndiName jms/MQCF]
     set mqcfAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name MQCF} {jndiName jms/MQCF}
     ```

   - Using Jython:

     ```
     name = ['name', 'MQCF']
     jndi = ['jndiName', 'jms/MQCF']
     mqcfAttrs = [name, jndi]
     print mqcfAttrs
     ```

     Example output:

     ```
     [[name, MQCF], [jndiName, jms/MQCF]]
     ```

4. Set up a template:
   - Using Jacl:

     ```
     set template [lindex [$AdminConfig listTemplates MQConnectionFactory] 0]
     ```

   - Using Jython:

```
import java
lineseparator = java.lang.System.getProperty('line.separator')
template = AdminConfig.listTemplates('MQConnectionFactory').split(lineseparator)[0]
print template
```

Example output:

```
Example non-XA WMQ ConnectionFactory(templates/
system:JMS-resource-provider-templates.xml
#MQConnectionFactory_3)
```

5. Create MQ connection factory:
   - Using Jacl:

     ```
     $AdminConfig createUsingTemplate MQConnectionFactory
     $newjmsp $mqcfAttrs $template
     ```

   - Using Jython:

     ```
     print AdminConfig.createUsingTemplate('MQConnectionFactory',
     newjmsp, mqcfAttrs, template)
     ```

   Example output:

   ```
   MQCF(cells/mycell/nodes/mynode:resources.xml#MQConnectionFactory_1)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new MQ queue connection factories using scripting

You can use scripting to configure a new MQ queue connection factory.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new MQ queue connection factory:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQQueueConnectionFactory
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQQueueConnectionFactory')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name MQQCF]
     set jndi [list jndiName jms/MQQCF]
     set mqqcfAttrs [list $name $jndi]
     ```

Example output:

```
{name MQQCF} {jndiName jms/MQQCF}
```

- Using Jython:

```
name = ['name', 'MQQCF']
jndi = ['jndiName', 'jms/MQQCF']
mqqcfAttrs = [name, jndi]
print mqqcfAttrs
```

Example output:

```
[[name, MQQCF], [jndiName, jms/MQQCF]]
```

4. Set up a template:
   - Using Jacl:

```
set template [lindex [$AdminConfig listTemplates MQQueueConnectionFactory] 0]
```

   - Using Jython:

```
import java
lineseparator = java.lang.System.getProperty('line.separator')
template = AdminConfig.listTemplates('MQQueueConnectionFactory').split(lineseparator)[0]
print template
```

   Example output:

```
Example non-XA WMQ QueueConnectionFactory(templates/
system:JMS-resource-provider-templates.xml
#MQQueueConnectionFactory_3)
```

5. Create MQ queue connection factory:
   - Using Jacl:

```
$AdminConfig createUsingTemplate MQQueueConnectionFactory
$newjmsp $mqqcfAttrs $template
```

   - Using Jython:

```
print AdminConfig.createUsingTemplate('MQQueueConnectionFactory',
newjmsp, mqqcfAttrs, template)
```

   Example output:

```
MQQCF(cells/mycell/nodes/mynode:resources.xml#MQQueueConnectionFactory_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new MQ topic connection factories using scripting

Use scripting and the wsadmin tool to configure a new MQ topic connection factory.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new MQ topic connection factory:

1. Identify the parent ID:
   - Using Jacl:

```
set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
```

   - Using Jython:

```
newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
print newjmsp
```

   Example output:

```
JMSP1(cells/mycell/nodes/mynode:resources.xml#JMSProvider_1)
```

2. Get required attributes:

- Using Jacl:

```
$AdminConfig required MQTopicConnectionFactory
```

- Using Jython:

```
print AdminConfig.required('MQTopicConnectionFactory')
```

Example output:

```
Attribute       Type
name       String
jndiName    String
```

3. Set up required attributes:
   - Using Jacl:

```
set name [list name MQTCF]
set jndi [list jndiName jms/MQTCF]
set mqtcfAttrs [list $name $jndi]
```

Example output:

```
{name MQTCF} {jndiName jms/MQTCF}
```

   - Using Jython:

```
name = ['name', 'MQTCF']
jndi = ['jndiName', 'jms/MQTCF']
mqtcfAttrs = [name, jndi]
print mqtcfAttrs
```

Example output:

```
[[name, MQTCF], [jndiName, jms/MQTCF]]
```

4. Set up a template:
   - Using Jacl:

```
set template [lindex [$AdminConfig listTemplates MQTopicConnectionFactory] 0]
```

   - Using Jython:

```
import java
lineseparator = java.lang.System.getProperty('line.separator')
template = AdminConfig.listTemplates('MQTopicConnectionFactory').split(lineseparator)[0]
print template
```

Example output:

```
Example non-XA WMQ TopicConnectionFactory(templates/system:
JMS-resource-provider-templates.xml
#MQTopicConnectionFactory_5)
```

5. Create mq topic connection factory:
   - Using Jacl:

```
$AdminConfig create MQTopicConnectionFactory $newjmsp $mqtcfAttrs $template
```

   - Using Jython:

```
print AdminConfig.create('MQTopicConnectionFactory', newjmsp, mqtcfAttrs, template)
```

Example output:

```
MQTCF(cells/mycell/nodes/mynode:resources.xml#MQTopicConnectionFactory_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new MQ queues using scripting

You can use scripting and the wsadmin tool to configure a new MQ queue.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new MQ queue:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQQueue
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQQueue')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   jndiName    String
   baseQueueName  String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name MQQ]
     set jndi [list jndiName jms/MQQ]
     set baseQN [list baseQueueName "Put the base queue name here"]
     set mqqAttrs [list $name $jndi $baseQN]
     ```

     Example output:

     ```
     {name MQQ} {jndiName jms/MQQ} {baseQueueName {Put the base queue name here}}
     ```

   - Using Jython:

     ```
     name = ['name', 'MQQ']
     jndi = ['jndiName', 'jms/MQQ']
     baseQN = ['baseQueueName', "Put the base queue name here"]
     mqqAttrs = [name, jndi, baseQN]
     print mqqAttrs
     ```

     Example output:

     ```
     [[name, MQQ], [jndiName, jms/MQQ], [baseQueueName, "Put the base queue name here"]]
     ```

4. Set up a template:
   - Using Jacl:

     ```
     set template [lindex [$AdminConfig listTemplates MQQueue] 0]
     ```

   - Using Jython:

     ```
     import java
     lineseparator = java.lang.System.getProperty('line.separator')
     template = AdminConfig.listTemplates('MQQueue').split(lineseparator)[0]
     print template
     ```

   Example output:

   ```
   Example.JMS.WMQ.Q1(templates/system:JMS-resource-provider-
   templates.xml#MQQueue_1)
   ```

5. Create MQ queue factory:
   - Using Jacl:

     ```
     $AdminConfig createUsingTemplate MQQueue $newjmsp $mqqAttrs $template
     ```

   - Using Jython:

     ```
     print AdminConfig.createUsingTemplate('MQQueue', newjmsp, mqqAttrs, template)
     ```

Example output:

```
MQQ(cells/mycell/nodes/mynode|resources.xml#MQQueue_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new MQ topics using scripting

You can use scripting to configure a new MQ topic.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new MQ topic:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newjmsp [$AdminConfig getid /Cell:mycell/Node:mynode/JMSProvider:JMSP1/]
     ```

   - Using Jython:

     ```
     newjmsp = AdminConfig.getid('/Cell:mycell/Node:myNode/JMSProvider:JMSP1')
     print newjmsp
     ```

   Example output:

   ```
   JMSP1(cells/mycell/nodes/mynode|resources.xml#JMSProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MQTopic
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MQTopic')
     ```

   Example output:

   ```
   Attribute            Type
   name                 String
   jndiName             String
   baseTopicName        String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name MQT]
     set jndi [list jndiName jms/MQT]
     set baseTN [list baseTopicName "Put the base topic name here"]
     set mqtAttrs [list $name $jndi $baseTN]
     ```

     Example output:

     ```
     {name MQT} {jndiName jms/MQT} {baseTopicName {Put the base topic name here}}
     ```

   - Using Jython:

     ```
     name = ['name', 'MQT']
     jndi = ['jndiName', 'jms/MQT']
     baseTN = ['baseTopicName', "Put the base topic name here"]
     mqtAttrs = [name, jndi, baseTN]
     print mqtAttrs
     ```

     Example output:

     ```
     [[name, MQT], [jndiName, jms/MQT], [baseTopicName, "Put the base topic name here"]]
     ```

4. Create MQ topic factory:
   - Using Jacl:

     ```
     $AdminConfig create MQTopic $newjmsp $mqtAttrs
     ```

- Using Jython:

  ```
  print AdminConfig.create('MQTopic', newjmsp, mqtAttrs)
  ```

  Example output:

  ```
  MQT(cells/mycell/nodes/mynode|resources.xml#MQTopic_1)
  ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Commands for the JCA management group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the JCA management group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
copyResourceAdapter	Use the **copyResourceAdapter** command to create a Java 2 Connector (J2C) resource adapter under the scope that you specify.	J2C Resource Adapter object ID	• Parameters: **- name** Indicates the name of the new J2C resource adapter. This parameter is required. **- scope** Indicates the scope object ID. This parameter is required. **- useDeepCopy** If you set this parameter to `true`, all of the J2C connection factory, J2C activation specification, and J2C administrative objects will be copied to the new J2C resource adapter (deep copy). If you set this parameter to `false`, the objects are not created (shallow copy). The default is `false`. • Returns: J2C resource adapter object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask copyResourceAdapter $ra [subst {-name newRA -scope $scope}]` • Using Jython string: `AdminTask.copyResourceAdapter (ra, '[-name newRA -scope scope]')` • Using Jython list: `AdminTask.copyResourceAdapter (ra, ['-name', 'newRA', '-scope', 'scope'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask copyResourceAdapter {-interactive}` • Using Jython string: `AdminTask.copyResourceAdapter ('[-interactive]')` • Using Jython list: `AdminTask.copyResourceAdapter (['-interactive'])`

createJ2C Activation Spec	Use the **createJ2C Activation Spec** command to create a Java 2 Connector (J2C) activation specification under a J2C resource adapter and the attributes that you specify. Use the messageListenerType parameter to indicate the activation specification that is defined for the J2C resource adapter.	J2C Resource Adapter object ID	• Parameters: **- messageListenerType** Identifies the activation specification for the J2C activation specification to be created. Use this parameter to identify the activation specification template for the J2C resource adapter that you specify. **- name** Indicates the name of the J2C activation specification that you are creating. **- jndiName** Indicates the name of the Java Naming and Directory Interface (JNDI). **- destinationJndiName** Indicates the name of the Java Naming and Directory Interface (JNDI) of corresponding destination. **- authenticationAlias** Indicates the authentication alias of the J2C activation specification that you are creating. **- description** Description of the created J2C activation specification. • Returns: J2CActivationSpec object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask createJ2CActivationSpec $ra {-name J2CActSpec -jndiName eis/ActSpec1 -messageListenerType javax.jms.MessageListener }` • Using Jython string: `AdminTask.createJ2CActivationSpec(ra, '[-name J2CActSpec -jndiName eis/ActSpec1 -messageListenerType javax.jms.MessageListener]')` • Using Jython list: `AdminTask.createJ2CActivationSpec(ra, ['-name', 'J2CActSpec', '-jndiName', 'eis/ActSpec1', '-messageListenerType', 'javax.jms.MessageListener'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createJ2CActivationSpec {-interactive}` • Using Jython string: `AdminTask.createJ2CActivationSpec ('[-interactive]')` • Using Jython list: `AdminTask.createJ2CActivationSpec (['-interactive'])`

createJ2C AdminObject	Use the **createJ2C AdminObject** command to create an administrative object under a resource adapter with attributes that you specify. Use the administrative object interface to indicate the administrative object that is defined in the resource adapter.	J2C Resource Adapter object ID	• Parameters: **-adminObjectInterface** Specifies the administrative object interface to identify the administrative object for the resource adapter that you specify. This parameter is required. **-name** Indicates the name of the administrative object. **-jndiName** Specifies the name of the Java Naming and Directory Interface (JNDI). **-description** Description of the created J2C admin object. • Returns: J2CAdminObject object ID	**Batch mode example usage:** • Using Jacl: `$AdminTask createJ2CAdmin Object $ra {-adminObject Interface fvt.adapter.mess age.FVTMessageProvider -name J2CA01 -jndiName eis/J2CA01}` • Using Jython string: `AdminTask.createJ2CAdminOb ject(ra, '[-adminObjectIn terface fvt.adapter.messag e.FVTMessageProvider -name J2CA01 -jndiName eis/J2CA01]')` • Using Jython list: `AdminTask.createJ2CAdminObj ect(ra, ['-adminObjectInte rface', 'fvt.adapter.messa ge.FVTMessageProvider', '-name', 'J2CA01', '-jndiName', 'eis/J2CA01'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createJ2CAdmin Object {-interactive}` • Using Jython string: `AdminTask.createJ2CAdmin Object ('[-interactive]')` • Using Jython list: `AdminTask.createJ2CAdmin Object (['-interactive'])`

createJ2C Connection Factory	Use the **createJ2C Connection Factory** command to create a Java 2 connection factory under a Java 2 resource adapter and the attributes that you specify. Use the connection factory interfaces to indicate the connection definitions that are defined for the Java 2 resource adapter.	J2C Resource Adapter object ID	• Parameters: **-connectionFactoryInterface** Identifies the connection definition for the Java 2 resource adapter that you specify. This parameter is required. **-name** Indicates the name of the connection factory. **-jndiName** Indicates the name of the Java Naming and Directory Interface (JNDI). **-description** Description of the created J2C connection factory. • Returns: The J2C connection factory object ID.	**Batch mode example usage:** • Using Jacl: `$AdminTask createJ2CConnectionFactory $ra {-connectionFactoryInterfaces javax.sql.DataSource -name J2CCF1 -jndiName eis/J2CCF1}` • Using Jython string: `AdminTask.createJ2CConnectionFactory(ra, '[-connectionFactoryInterfaces javax.sql.DataSource -name J2CCF1 -jndiName eis/J2CCF1]')` • Using Jython list: `AdminTask.createJ2CConnectionFactory(ra, ['-connectionFactoryInterfaces', 'javax.sql.DataSource', '-name', 'J2CCF1', '-jndiName', 'eis/J2CCF1'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createJ2CConnectionFactory {-interactive}` • Using Jython string: `AdminTask.createJ2CConnectionFactory ('[-interactive]')` • Using Jython list: `AdminTask.createJ2CConnectionFactory (['-interactive'])`
listAdminObjectInterfaces	Use the **listAdminObjectInterfaces** command to list the administrative object interfaces that are defined under the resource adapter that you specify.	J2C Resource Adapter object ID	• Parameters: None • Returns: A list of administrative object interfaces.	**Batch mode example usage:** • Using Jacl: `$AdminTask listAdminObjectInterfaces $ra` • Using Jython string: `AdminTask.listAdminObjectInterfaces(ra)` • Using Jython list: `AdminTask.listAdminObjectInterfaces(ra)` **Interactive mode example usage:** • Using Jacl: `$AdminTask listAdminObjectInterfaces {-interactive}` • Using Jython string: `AdminTask.listAdminObjectInterfaces ('[-interactive]')` • Using Jython list: `AdminTask.listAdminObjectInterfaces (['-interactive'])`

listConnectionFactoryInterfaces	Use the **listConnectionFactoryInterfaces** command to list all of the connection factory interfaces that are defined under the Java 2 connector resource adapter that you specify.	J2C Resource Adapter object ID	• Parameters: None • Returns: A list of connection factory interfaces.	**Batch mode example usage:** • Using Jacl: `$AdminTask listConnectionFactoryInterfaces $ra` • Using Jython string: `AdminTask.listConnectionFactoryInterfaces(ra)` • Using Jython list: `AdminTask.listConnectionFactoryInterfaces(ra)` **Interactive mode example usage:** • Using Jacl: `$AdminTask listConnectionFac toryInterfaces {-interactive}` • Using Jython string: `AdminTask.listConnectionFac toryInterfaces ('[-intera ctive]')` • Using Jython list: `AdminTask.listConnectionFac toryInterfaces (['-interac tive'])`
listJ2CActivationSpecs	Use the **listJ2CActivation Specs** command to list the activation specifications that are contained under the resource adapter and message listener type that you specify.	J2C Resource Adapter object ID	• Parameters: **-messageListenerType** Specifies the message listener type for the resource adapter for which you are making a list. This parameter is required. • Returns: A list of activation specifications that has specified messageListener type.	**Batch mode example usage:** • Using Jacl: `$AdminTask listJ2CActivation Specs $ra {-messageListener Type javax.jms.MessageLis tener}` • Using Jython string: `AdminTask.listJ2CActivation Specs(ra, '[-messageListener Type javax.jms.MessageLis tener]')` • Using Jython list: `AdminTask.listJ2CActivation Specs(ra, ['-messageListener Type', 'javax.jms.Message Listener'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listJ2CActivat ionSpecs {-interactive}` • Using Jython string: `AdminTask.listJ2CActivat ionSpecs ('[-interactive]')` • Using Jython list: `AdminTask.listJ2CActivati onSpecs (['-interactive'])`

listJ2CAdminObjects	Use the **listJ2CAdminObjects** command to list administrative objects that contain the administrative object interface that you specify.	J2C Resource Adapter object ID	• Parameters: **-adminObjectInterface** Specifies the administrative object interface for which you want to list. This parameter is required. • Returns: A list of administrative objects that has specified adminObjectInterface.	**Batch mode example usage:** • Using Jacl: `$AdminTask listJ2CAdminObjects $ra {-adminObjectInterface fvt.adaptor.message.FVTMessageProvider}` • Using Jython string: `AdminTask.listJ2CAdminObjects(ra, '[-adminObjectInterface fvt.adaptor.message.FVTMessageProvider]')` • Using Jython list: `AdminTask.listJ2CAdminObjects(ra, ['-adminObjectInterface', 'fvt.adaptor.message.FVTMessageProvider'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listJ2CAdminObjects {-interactive}` • Using Jython string: `AdminTask.listJ2CAdminObjects ('[-interactive]')` • Using Jython list: `AdminTask.listJ2CAdminObjects (['-interactive'])`

listJ2CConnectionFactories	Use the **listJ2CConnectionFactories** command to list the Java 2 connector connection factories under the resource adapter and connection factory interface that you specify.	J2C Resource Adapter object ID	• Parameters: **-connectionFactoryInterface** Indicates the name of the connection factory that you want to list. This parameter is required. • Returns: A list of J2C connectionFactory that has the specified connectionFactoryInterface.	**Batch mode example usage:** • Using Jacl: `$AdminTask listJ2CConnectionFactories $ra {-connectionFactoryInterface javax.sql.DataSource}` • Using Jython string: `AdminTask.listJ2CConnectionFactories(ra, '[-connectionFactoryInterface javax.sql.DataSource]')` • Using Jython list: `AdminTask.listJ2CConnectionFactories(ra, ['-connectionFactoryInterface', 'javax.sql.DataSource'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask listJ2CConnectionFactories {-interactive}` • Using Jython string: `AdminTask.listJ2CConnectionFactories ('[-interactive]')` • Using Jython list: `AdminTask.listJ2CConnectionFactories (['-interactive'])`
listMessageListenerTypes	Use the **listMessageListenerTypes** command to list the message listener types that are defined under the resource adapter that you specify.	J2C Resource Adapter object ID	• Parameters: None • Returns: A list of message listener types.	**Batch mode example usage:** • Using Jacl: `$AdminTask listMessageListenerTypes $ra` • Using Jython string: `AdminTask.listMessageListenerTypes(ra)` • Using Jython list: `AdminTask.listMessageListenerTypes(ra)` **Interactive mode example usage:** • Using Jacl: `$AdminTask listMessageListenerTypes {-interactive}` • Using Jython string: `AdminTask.listMessageListenerTypes ('[-interactive]')` • Using Jython list: `AdminTask.listMessageListenerTypes (['-interactive'])`

Configuring mail, URLs, and resource environment entries with scripting

Use scripting to configure mail, URLs, and resource environment entries.

This topic contains the following tasks:

Configuring new mail providers using scripting

You can use scripting and the wsadmin tool to configure new mail providers.

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

Perform the following steps to configure a new mail provider:

1. Identify the parent ID:
   - Using Jacl:
     ```
     set node [$AdminConfig getid /Cell:mycell/Node:mynode/]
     ```
   - Using Jython:
     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:

   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```
2. Get required attributes:
   - Using Jacl:
     ```
     $AdminConfig required MailProvider
     ```
   - Using Jython:
     ```
     print AdminConfig.required('MailProvider')
     ```

   Example output:

   ```
   Attribute       Type
   name       String
   ```
3. Set up required attributes:
   - Using Jacl:
     ```
     set name [list name MP1]
     set mpAttrs [list $name]
     ```
   - Using Jython:

```
name = ['name', 'MP1']
mpAttrs = [name]
```

4. Create the mail provider:
   - Using Jacl:

     ```
     set newmp [$AdminConfig create MailProvider $node $mpAttrs]
     ```

   - Using Jython:

     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

     Example output:

     ```
     MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
     ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new mail sessions using scripting

You can use scripting and the wsadmin tool to configure new mail sessions.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new mail session:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
     ```

   - Using Jython:

     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

     Example output:

     ```
     MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
     ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required MailSession
     ```

   - Using Jython:

     ```
     print AdminConfig.required('MailSession')
     ```

     Example output:

     ```
     Attribute       Type
     name       String
     jndiName       String
     ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set name [list name MS1]
     set jndi [list jndiName mail/MS1]
     set msAttrs [list $name $jndi]
     ```

     Example output:

     ```
     {name MS1} {jndiName mail/MS1}
     ```

   - Using Jython:

     ```
     name = ['name', 'MS1']
     jndi = ['jndiName', 'mail/MS1']
     msAttrs = [name, jndi]
     print msAttrs
     ```

Example output:

```
[[name, MS1], [jndiName, mail/MS1]]
```

4. Create the mail session:
   - Using Jacl:

     ```
     $AdminConfig create MailSession $newmp $msAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('MailSession', newmp, msAttrs)
     ```

   Example output:

   ```
   MS1(cells/mycell/nodes/mynode|resources.xml#MailSession_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new protocols using scripting

You can configure new protocols with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new protocol:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
     ```

   - Using Jython:

     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

   Example output:

   ```
   MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
   ```

2. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required ProtocolProvider
     ```

   - Using Jython:

     ```
     print AdminConfig.required('ProtocolProvider')
     ```

   Example output:

   ```
   Attribute       Type
   protocol      String
   classname       String
   ```

3. Set up required attributes:
   - Using Jacl:

     ```
     set protocol [list protocol "Put the protocol here"]
     set classname [list classname "Put the class name here"]
     set ppAttrs [list $protocol $classname]
     ```

     Example output:

     ```
     {protocol protocol1} {classname classname1}
     ```

   - Using Jython:

     ```
     protocol = ['protocol', "Put the protocol here"]
     classname = ['classname', "Put the class name here"]
     ppAttrs = [protocol, classname]
     print ppAttrs
     ```

Example output:

```
[[protocol, protocol1], [classname, classname1]]
```

4. Create the protocol provider:
   - Using Jacl:

     ```
     $AdminConfig create ProtocolProvider $newmp $ppAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('ProtocolProvider', newmp, ppAttrs)
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#ProtocolProvider_4)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new custom properties using scripting

You can use scripting and the wsadmin tool to configure new custom properties.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new custom property:

1. Identify the parent ID:
   - Using Jacl:

     ```
     set newmp [$AdminConfig getid /Cell:mycell/Node:mynode/MailProvider:MP1/]
     ```

   - Using Jython:

     ```
     newmp = AdminConfig.create('MailProvider', node, mpAttrs)
     print newmp
     ```

   Example output:

   ```
   MP1(cells/mycell/nodes/mynode|resources.xml#MailProvider_1)
   ```

2. Get the J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newmp propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newmp, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#PropertySet_2)
   ```

3. Get required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

   Example output:

   ```
   Attribute       Type
   name      String
   ```

4. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name CP1]
     set cpAttrs [list $name]
     ```

Example output:

```
{name CP1}
```

- Using Jython:

```
name = ['name', 'CP1']
cpAttrs = [name]
print cpAttrs
```

Example output:

```
[[name, CP1]]
```

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $cpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, cpAttrs)
     ```

   Example output:

   ```
   CP1(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_2)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new resource environment providers using scripting

You can use the wsadmin tool and scripting to configure new resources environment providers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new resource environment provider:

1. Identify the parent ID and assign it to the node variable.
   - Using Jacl:

     ```
     set node [$AdminConfig  getid  /Cell:mycell/Node:mynode/]
     ```

   - Using Jython:

     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:

   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required ResourceEnvironmentProvider
     ```

   - Using Jython:

     ```
     print AdminConfig.required('ResourceEnvironmentProvider')
     ```

   Example output:

   ```
   Attribute     Type
   name     String
   ```

3. Set up the required attributes and assign it to the repAttrs variable:
   - Using Jacl:

     ```
     set n1 [list name REP1]
     set repAttrs [list $name]
     ```

   - Using Jython:

```
n1 = ['name', 'REP1']
repAttrs = [n1]
```

4. Create a new resource environment provider:
   - Using Jacl:

     ```
     set newrep [$AdminConfig create ResourceEnvironmentProvider $node $repAttrs]
     ```

   - Using Jython:

     ```
     newrep = AdminConfig.create('ResourceEnvironmentProvider', node, repAttrs)
     print newrep
     ```

     Example output:

     ```
     REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
     ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring custom properties for resource environment providers using scripting

You can use scripting to configure custom properties for a resource environment provider.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new custom property for a resource environment provider:

1. Identify the parent ID and assign it to the newrep variable.
   - Using Jacl:

     ```
     set newrep [$AdminConfig getid /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
     ```

   - Using Jython:

     ```
     newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
     print newrep
     ```

     Example output:

     ```
     REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
     ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

   - Using Jython:

     ```
     print AdminConfig.required('J2EEResourceProperty')
     ```

     Example output:

     ```
     Attribute     Type
     name     String
     ```

3. Set up the required attributes and assign it to the repAttrs variable:
   - Using Jacl:

     ```
     set name [list name RP]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP']
     rpAttrs = [name]
     ```

4. Get the J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newrep propertySet]
     ```

- Using Jython:

```
propSet = AdminConfig.showAttribute(newrep, 'propertySet')
print propSet
```

Example output:

```
(cells/mycell/nodes/mynode|resources.xml#PropertySet_1)
```

5. Create a J2EE resource property:
   - Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

   - Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

   Example output:

```
RP(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new referenceables using scripting

You can use scripting and the wsadmin tool to configure new referenceables.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new referenceable:

1. Identify the parent ID and assign it to the newrep variable.
   - Using Jacl:

```
set newrep [$AdminConfig  getid  /Cell:mycell/Node:mynode/
ResourceEnvironmentProvider:REP1/]
```

   - Using Jython:

```
newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/
ResourceEnvironmentProvider:REP1/')
print newrep
```

   Example output:

```
REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
```

2. Identify the required attributes:
   - Using Jacl:

```
$AdminConfig required Referenceable
```

   - Using Jython:

```
print AdminConfig.required('Referenceable')
```

   Example output:

```
Attribute     Type
factoryClassname   String
classname   String
```

3. Set up the required attributes:
   - Using Jacl:

```
set fcn [list factoryClassname REP1]
set cn [list classname NM1]
set refAttrs [list $fcn $cn]
```

   - Using Jython:

```
fcn = ['factoryClassname', 'REP1']
cn = ['classname', 'NM1']
refAttrs = [fcn, cn]
print refAttrs
```

Example output:

```
{factoryClassname {REP1}} {classname {NM1}}
```

4. Create a new referenceable:
   - Using Jacl:

     ```
     set newref [$AdminConfig create Referenceable  $newrep  $refAttrs]
     ```

   - Using Jython:

     ```
     newref = AdminConfig.create('Referenceable',  newrep,  refAttrs)
     print newref
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#Referenceable_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new resource environment entries using scripting

You can use scripting and the wsadmin tool to configure a new resource environment entry.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new resource environment entry:

1. Identify the parent ID and assign it to the newrep variable.
   - Using Jacl:

     ```
     set newrep [$AdminConfig  getid  /Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/]
     ```

   - Using Jython:

     ```
     newrep = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvironmentProvider:REP1/')
     print newrep
     ```

   Example output:

   ```
   REP1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvironmentProvider_1)
   ```

2. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required ResourceEnvEntry
     ```

   - Using Jython:

     ```
     print AdminConfig.required('ResourceEnvEntry')
     ```

   Example output:

   ```
   Attribute     Type
   name      String
   jndiName   String
   referenceable   Referenceable@
   ```

3. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name REE1]
     set jndiName [list jndiName myjndi]
     set newref [$AdminConfig getid /Cell:mycell/Node:mynode/Referenceable:/]
     set ref [list referenceable $newref]
     set reeAttrs [list $name $jndiName $ref]
     ```

- Using Jython:

```
name = ['name', 'REE1']
jndiName = ['jndiName', 'myjndi']
newref = AdminConfig.getid('/Cell:mycell/Node:mynode/Referenceable:/')
ref = ['referenceable', newref]
reeAttrs = [name, jndiName, ref]
```

4. Create the resource environment entry:
   - Using Jacl:

```
$AdminConfig create ResourceEnvEntry $newrep $reeAttrs
```

   - Using Jython:

```
print AdminConfig.create('ResourceEnvEntry', newrep, reeAttrs)
```

   Example output:

```
REE1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvEntry_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring custom properties for resource environment entries using scripting

You can use scripting to configure a new custom property for a resource environment entry.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new custom property for a resource environment entry:

1. Identify the parent ID and assign it to the newree variable.
   - Using Jacl:

```
set newree [$AdminConfig  getid  /Cell:mycell/Node:mynode/ResourceEnvEntry:REE1/]
```

   - Using Jython:

```
newree = AdminConfig.getid('/Cell:mycell/Node:mynode/ResourceEnvEntry:REE1/')
print newree
```

   Example output:

```
REE1(cells/mycell/nodes/mynode|resources.xml#ResourceEnvEntry_1)
```

2. Create the J2EE custom property set:
   - Using Jacl:

```
set propSet [$AdminConfig showAttribute $newree propertySet]
```

   - Using Jython:

```
propSet = AdminConfig.showAttribute(newree, 'propertySet')
print propSet
```

   Example output:

```
(cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_5)
```

3. Identify the required attributes:
   - Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

   - Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

   Example output:

```
Attribute     Type
name      String
```

4. Set up the required attributes:
   - Using Jacl:
     ```
     set name [list name RP1]
     set rpAttrs [list $name]
     ```
   - Using Jython:
     ```
     name = ['name', 'RP1']
     rpAttrs = [name]
     ```
5. Create the J2EE custom property:
   - Using Jacl:
     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```
   - Using Jython:
     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:
   ```
   RPI(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_1)
   ```
6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new URL providers using scripting

You can use scripting and the wsadmin tool to configure new URL providers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new URL provider:

1. Identify the parent ID and assign it to the node variable.
   - Using Jacl:
     ```
     set node [$AdminConfig  getid  /Cell:mycell/Node:mynode/]
     ```
   - Using Jython:
     ```
     node = AdminConfig.getid('/Cell:mycell/Node:mynode/')
     print node
     ```

   Example output:
   ```
   mynode(cells/mycell/nodes/mynode|node.xml#Node_1)
   ```
2. Identify the required attributes:
   - Using Jacl:
     ```
     $AdminConfig required URLProvider
     ```
   - Using Jython:
     ```
     print AdminConfig.required('URLProvider')
     ```

   Example output:
   ```
   Attribute     Type
   streamHandlerClassName   String
   protocol       String
   name         String
   ```
3. Set up the required attributes:
   - Using Jacl:
     ```
     set name [list name URLP1]
     set shcn [list streamHandlerClassName "Put the stream handler classname here"]
     set protocol [list protocol "Put the protocol here"]
     set urlpAttrs [list $name $shcn $protocol]
     ```

Example output:

```
{name URLP1} {streamHandlerClassName {Put the stream handler classname here}}
{protocol {Put the protocol here}}
```

- Using Jython:

```
name = ['name', 'URLP1']
shcn = ['streamHandlerClassName', "Put the stream handler classname here"]
protocol = ['protocol', "Put the protocol here"]
urlpAttrs = [name, shcn, protocol]
print urlpAttrs
```

Example output:

```
[[name, URLP1], [streamHandlerClassName, "Put the stream handler classname here"],
[protocol, "Put the protocol here"]]
```

4. Create a URL provider:
   - Using Jacl:

     ```
     $AdminConfig create URLProvider $node $urlpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('URLProvider', node, urlpAttrs)
     ```

   Example output:

   ```
   URLP1(cells/mycell/nodes/mynode|resources.xml#URLProvider_1)
   ```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring custom properties for URL providers using scripting

You can use scripting to configure custom properties for URL providers.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure custom properties for URL providers:

1. Identify the parent ID and assign it to the newurlp variable.
   - Using Jacl:

     ```
     set newurlp [$AdminConfig  getid  /Cell:mycell/Node:mynode/URLProvider:URLP1/]
     ```

   - Using Jython:

     ```
     newurlp = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/')
     print newurlp
     ```

   Example output:

   ```
   URLP1(cells/mycell/nodes/mynode|resources.xml#URLProvider_1)
   ```

2. Get the J2EE resource property set:
   - Using Jacl:

     ```
     set propSet [$AdminConfig showAttribute $newurlp propertySet]
     ```

   - Using Jython:

     ```
     propSet = AdminConfig.showAttribute(newurlp, 'propertySet')
     print propSet
     ```

   Example output:

   ```
   (cells/mycell/nodes/mynode|resources.xml#PropertySet_7)
   ```

3. Identify the required attributes:
   - Using Jacl:

     ```
     $AdminConfig required J2EEResourceProperty
     ```

- Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

Example output:

```
Attribute     Type
name          String
```

4. Set up the required attributes:
   - Using Jacl:

```
set name [list name RP2]
set rpAttrs [list $name]
```

   - Using Jython:

```
name = ['name', 'RP2']
rpAttrs = [name]
```

5. Create a J2EE resource property:
   - Using Jacl:

```
$AdminConfig create J2EEResourceProperty $propSet $rpAttrs
```

   - Using Jython:

```
print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
```

   Example output:

```
RP2(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_1)
```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring new URLs using scripting

You can use scripting and the wsadmin tool to configure new URLs.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following example to configure a new URL:

1. Identify the parent ID and assign it to the newurlp variable.
   - Using Jacl:

```
set newurlp [$AdminConfig  getid  /Cell:mycell/Node:mynode/URLProvider:URLP1/]
```

   - Using Jython:

```
newurlp = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/')
print newurlp
```

   Example output:

```
URLP1(cells/mycell/nodes/mynode|resources.xml#URLProvider_1)
```

2. Identify the required attributes:
   - Using Jacl:

```
$AdminConfig required URL
```

   - Using Jython:

```
print AdminConfig.required('URL')
```

   Example output:

```
Attribute     Type
name          String
spec          String
```

3. Set up the required attributes:

- Using Jacl:

```
set name [list name URL1]
set spec [list spec "Put the spec here"]
set urlAttrs [list $name $spec]
```

  Example output:

```
{name URL1} {spec {Put the spec here}}
```

- Using Jython:

```
name = ['name', 'URL1']
spec = ['spec', "Put the spec here"]
urlAttrs = [name, spec]
```

  Example output:

```
[[name, URL1], [spec, "Put the spec here"]]
```

4. Create a URL:
   - Using Jacl:

```
$AdminConfig create URL $newurlp $urlAttrs
```

   - Using Jython:

```
print AdminConfig.create('URL', newurlp, urlAttrs)
```

   Example output:

```
URL1(cells/mycell/nodes/mynode|resources.xml#URL_1)
```

5. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Configuring custom properties for URLs using scripting

Use the wsadmin tool and scripting to set custom properties for URLs.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to configure a new custom property for a URL:

1. Identify the parent ID and assign it to the newurl variable.
   - Using Jacl:

```
set newurl [$AdminConfig getid /Cell:mycell/Node:mynode/URLProvider:URLP1/URL:URL1/]
```

   - Using Jython:

```
newurl = AdminConfig.getid('/Cell:mycell/Node:mynode/URLProvider:URLP1/URL:URL1/')
print newurl
```

   Example output:

```
URL1(cells/mycell/nodes/mynode|resources.xml#URL_1)
```

2. Create a J2EE resource property set:
   - Using Jacl:

```
set propSet [$AdminConfig showAttribute $newurl propertySet]
```

   - Using Jython:

```
propSet = AdminConfig.showAttribute(newurl, 'propertySet')
print propSet
```

   Example output:

```
(cells/mycell/nodes/mynode|resources.xml#J2EEResourcePropertySet_7)
```

3. Identify the required attributes:
   - Using Jacl:

```
$AdminConfig required J2EEResourceProperty
```

- Using Jython:

```
print AdminConfig.required('J2EEResourceProperty')
```

Example output:

```
Attribute     Type
name         String
```

4. Set up the required attributes:
   - Using Jacl:

     ```
     set name [list name RP3]
     set rpAttrs [list $name]
     ```

   - Using Jython:

     ```
     name = ['name', 'RP3']
     rpAttrs = [name]
     ```

5. Create a J2EE resource property:
   - Using Jacl:

     ```
     $AdminConfig create J2EEResourceProperty $propSet $rpAttrs
     ```

   - Using Jython:

     ```
     print AdminConfig.create('J2EEResourceProperty', propSet, rpAttrs)
     ```

   Example output:

   ```
   RP3(cells/mycell/nodes/mynode|resources.xml#J2EEResourceProperty_7)
   ```

6. Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.
7. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more information.

Commands for the provider group of the AdminTask object

Use the commands in the provider group to create or delete a provider. For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the provider group of the AdminTask object:

Table 23.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
createProvider	The **createProvider** command creates a provider in the model given the ProviderInfo.	None	• Parameters: None • Returns: true if the provider was created, false if not.	**Interactive mode example usage:** • Using Jacl: `$AdminTask createProvider {-interactive}` • Using Jython string: `AdminTask.createProvider ('[-interactive]')` • Using Jython list: `AdminTask.createProvider (['-interactive'])`

Table 23. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deleteProvider	The **deleteProvider** command deletes a provider from the model given the providerName.	None	• Parameters: None • Returns: true if the provider was deleted, false if not.	**Interactive mode example usage:** • Using Jacl: `$AdminTask deleteProvider {-interactive}` • Using Jython string: `AdminTask.deleteProvider ('[-interactive]')` • Using Jython list: `AdminTask.deleteProvider (['-interactive'])`
getProviderInfo	The **getProviderInfo** command returns a ProviderInfo object defined in the model given the providerName.	None	• Parameters: None • Returns: A ProviderInfo object defined in the model given the providerName.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getProviderInfo {-interactive}` • Using Jython string: `AdminTask.getProviderInfo ('[-interactive]')` • Using Jython list: `AdminTask.getProviderInfo (['-interactive'])`
getProviderInstance	The **getProviderInstance** command returns a java.security.Provider for the providerName specified.	None	• Parameters: None • Returns: A java.security.Provider for the providerName specified.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getProviderInsta nce {-interactive}` • Using Jython string: `AdminTask.getProviderIns tance ('[-interactive]')` • Using Jython list: `AdminTask.getProviderInst ance (['-interactive'])`

Table 23. (continued)

Command name:	Description:	Target object:	Parameters and return values:	Examples:
getProviders	The **getProviders** command returns a List of ProviderInfo objects from the model.	None	• Parameters: None • Returns: A list of ProviderInfo objects from the model.	**Interactive mode example usage:** • Using Jacl: `$AdminTask getProviders {-interactive}` • Using Jython string: `AdminTask.getProviders ('[-interactive]')` • Using Jython list: `AdminTask.getProviders (['-interactive'])`

Using the thin administrative client with the wsadmin tool

With the thin administrative client, you can run the wsadmin tool or a standalone administrative Java program with only a couple of JAR files. This leads to a reduction in the amount of time that it takes for the wsadmin tool to start and improved performance.

For tracing and logging information for the thin administrative client, see the Enabling trace on client and standalone applications article in the *Troubleshooting and support* PDF.

1. Make the thin administrative client JAR files available by copying them from a WebSphere Application Server environment to an environment outside of WebSphere Application Server. The thin administrative client JAR files are located in one of the following locations:
   - The *AppServer*`/runtimes` directory.
   - The *AppClient*`/runtimes` directory, if you optionally selected the thin administrative client when you installed the application client.

   You must have a special license agreement to download these JAR files.
2. Use the 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 or pull them in as bundles for an OSGi environment. For more information see the "Compiling an application in an OSGi environment using scripting" and the "Compiling an application in a non-OSGi environment using scripting" on page 660 articles.
3. Launch the thin wsadmin client or run the wsadmin tool remotely in a Java 2 Platform, Standard Edition environment.
4. Deploy the stand alone JAR files and the administrative client applications and start the applications outside of a WebSphere Application Server environment.

Compiling an application in an OSGi environment using scripting

To compile an application in an OSGi environment, the application must be contained in an OSGi bundle. Perform the following steps:

1. Create a `plugins` directory that includes the JAR files that you will need. Include the following JAR files in the CLASSPATH:

```
com.ibm.ws.admin.client_6.1.0.jar
org.eclipse.core.runtime_3.1.0.jar
org.eclipse.jdt.core_3.1.0.jar
org.eclipse.osgi_3.1.0.jar
org.eclipse.update.configurator_3.1.0.jar
```

By convention, all of the OSGi bundles are placed in the `plugins` directory.

2. Create an application class that implements the IPlatformRunnable interface. For example:

```
package ...


import org.eclipse.core.runtime.IPlatformRunnable;

public class ThinClientTest  implements IPlatformRunnable
{


    public ThinClientTest  () {

        System.out.println("ThinClientTest has been constructed");
    }

    public Object run(Object o) throws Exception
    {
                new ThinClientTest().executeUtility();
                return null;
    }
    private void executeUtility() throws Exception
    {

      . . .
    }
```

3. Create a bundle activator for the bundle. For example:

```
package com.ibm.ws;

import java.util.Set;
import java.util.List;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceListener;
import org.osgi.framework.ServiceEvent;

/**
 * This class implements a simple bundle that utilizes the OSGi
 * framework's event mechanism to listen for service events. Upon
 * receiving a service event, it prints out the event's details.
**/
public class JWSClientBundleActivator implements BundleActivator, ServiceListener
{
    /**
     * Implements BundleActivator.start(). Prints a message and adds itself to the bundle context
     * as a service listener.
     * @param context the framework context for the bundle.
    **/
    public void start(BundleContext context) throws Exception
    {
        System.out.println("Starting");
         context.addServiceListener(this);
        System.out.println("Starting to listen for service events.");
    }

    /**
     * Implements BundleActivator.stop(). Prints
     * a message and removes itself from the bundle context as a
```

```
     * service listener.
     * @param context the framework context for the bundle.
    **/
    public void stop(BundleContext context) throws Exception
    {
        context.removeServiceListener(this);
        System.out.println("Stopped listening for service events.");
        // Note: It is not required that we remove the listener here,
        // since the framework will do it automatically anyway.
    }

    /**
     * Implements ServiceListener.serviceChanged().
     * Prints the details of any service event from the framework.
     * @param event the fired service event.
    **/
    public void serviceChanged(ServiceEvent event)
    {
        String[] objectClass = (String[])
            event.getServiceReference().getProperty("objectClass");

        if (event.getType() == ServiceEvent.REGISTERED)
        {
            System.out.println(
                "Ex1: Service of type " + objectClass[0] + " registered.");
        }
        else if (event.getType() == ServiceEvent.UNREGISTERING)
        {
            System.out.println(
                "Ex1: Service of type " + objectClass[0] + " unregistered.");
        }
        else if (event.getType() == ServiceEvent.MODIFIED)
        {
            System.out.println(
                "Ex1: Service of type " + objectClass[0] + " modified.");
        }
    }
}
```

4. Create the `MANIFEST.MF` application. For example:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Client1
Bundle-SymbolicName: com.ibm.ws; singleton=true
Bundle-Version: 1.0.0
Bundle-Vendor: IBM
Bundle-Localization: plugin
Eclipse-AutoStart: true
Bundle-Activator: com.ibm.ws.JWSClientBundleActivator
DynamicImport-Package: *
```

5. Create the `plugin.xml` file. For example:

```
<?xml version= "1.0" encoding="UTF-8"?>
<?eclipse version= "3.0"?>
<plugin
   id="com.ibm.ws"
   name="Client1"
   version="1.0.0"
   provider-name="IBM"
>
   <extension id="Client1" name="Client1"
   point="org.eclipse.core.runtime.applications">
     <application>
       <run class="name_of_your_client_class" />
     </application>
   </extension>
</plugin>
```

6. Compile the application and the bundle activator. For example:

```
CLASSPATH=/opt/resources/com.ibm.ws.admin.client_6.1.0.jar:${CLASSPATH}
cd ${ HOME}/com/ibm/ws
${JAVA_HOME}/bin/javac  ThinClientTest.java
${JAVA_HOME}/bin/javac  JWSClientBundleActivator.java
```

7. Package the `MANIFEST.MF` application and the `plugin.xml` files into an OSGi bundle. You perform do this step using the JAR utility to turn the `MANIFEST.MF` application, the `plugin.xml` file, and the Java classes for the application into a JAR file.

Compiling an application in a non-OSGi environment using scripting

To use the thin administrative client JAR files to compile an application, perform the following steps:

1. Include the `com.ibm.ws.admin.client_6.1.0.jar` JAR file in the CLASSPATH.
2. Compile the application.

For example, the JAR file is located in the *app_server_root*`/opt/resources` directory and you want to compile the ThinAdminClientApplication.java file in the current directory. Use the following to compile the file:

```
export CLASSPATH=/opt/resources/com.ibm.ws.admin.client_6.1.0.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 JAR files that support administrative client functions that you can use with IBM Developer Kits For the Java Platform. For more information about thin administrative clients, see the Application client functions article in the *Developing and deploying applications* PDF.

For tracing and logging information for the thin administrative client, see the Enabling trace on client and standalone applications article in the *Troubleshooting and support* PDF.

1. Obtain the thin administrative client JAR files from the WebSphere Application Server Network Deployment installation from one of the following places:
   - The *app_server_root*`/runtimes` directory.
   - The *app_client_root*`/runtimes` directory, if you optionally installed this JAR file when you installed the application client.

   Copy the JAR files into the environment from where you want to use them.
2. Set or export the classpath to include the *app_server_root*`/runtimes/com.ibm.ws.admin.client_6.1.0.jar` file on the remote client machine.
3. Run the setupCmdLine.sh or the setupCmdLine.bat file.
4. Copy the wsadmin.sh or the wsadmin.bat file from the server machine. This file does not ship with the application client installation.
5. Start the wsadmin tool in either an OSGi or a non-OSGi environment.
   a. Modify the wsadmin.sh or the wsadmin.bat file to include the following settings:

```
JAVA_HOME: location of the JDK/JRE installation
CLIENTSAS: location of the sas.client.props file
CLIENTSOAP: location of the soap.client.props file
JAASSOAP: location of the wsjaas_client.conf file
TAC_CLASSPATH: location of the thin administrative client JAR.
If you use the Jython language, use the jython.jar file.
If you use the Jacl file, use the jacl.jar and tcljava.jar files.
```

   The additional JAR files are also located in the run time directory.

b. Use the `startup.jar` file to start the application. There are several properties that you can set. You can place these properties in environment variables which you can use on the command line to invoke the administrative client application. Then you can place these properties directly on the command line and invoke Java using the `-D` argument. For example:

- Using the SOAP connector. The SOAP connector settings, including the security settings, are stored in a file pointed to by the com.ibm.SOAP.ConfigURL property. Application code can override some of the settings through properties that you pass in to create the administrative client. The following example application code initializes the properties used to create the connector:

```
Properties props = new Properties();
props.setProperty(AdminClient.CONNECTOR_HOST, "host_name");
props.setProperty(AdminClient.CONNECTOR_PORT, "port");
props.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient. CONNECTOR_TYPE_SOAP);
props.setProperty(AdminClient.CONNECTOR_SECURITY_ENABLED, "false");
```

  The following example starts the user program:

```
#!/bin/sh
CONNECTORPROPS=-Dcom.ibm.SOAP.ConfigURL=location of soap.client.props
JAVA_HOME=location of Java
TAC_CLASSPATH=location of the thin client JAR

"${JAVA_HOME}/bin/java"  \
-Djava.ext.dirs="${JAVA_HOME}/jre/lib/ext"  \
-classpath "${TAC_CLASSPATH}" ${CONNECTORPROPS} UserProgram
```

- Using the RMI connector. The RMI settings used by the connector, including security settings, are stored in a file pointed to by the com.ibm.CORBA.ConfigURL property. Application code can override some of the settings through properties that you pass in to create the administrative client. The following example application code initializes the properties used to create the administrative client:

```
Properties props = new Properties();
props.setProperty(AdminClient.CONNECTOR_HOST, "host_name");
props.setProperty(AdminClient.CONNECTOR_PORT, "port");
props.setProperty(AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_RMI);
props.setProperty(AdminClient.CONNECTOR_SECURITY_ENABLED, "false");
```

  The following example starts the application:

```
#!/bin/sh
CONNECTORPROPS=-Dcom.ibm.CORBA.configURL=location of sas.client.props
JAVA_HOME=location of Java
TAC_CLASSPATH=location of thin client JAR

"${JAVA_HOME}/bin/java"  \
-Djava.ext.dirs="${JAVA_HOME}/jre/lib/ext"  \
-classpath "${TAC_CLASSPATH}"  ${CONNECTORPROPS}  UserProgram
```

- Launching an OSGI application. For example, assuming that the supporting JAR files are in the *app_server_root*/plugins directory and you put the `startup.jar` file in the *app_server_root*/lib/startup.jar directory, you can run the application in the bundle using the following:

```
#!/bin/bash
#


. setupCmdLine.bsh
################################################
set -x
cd ${HOME}

trap ERR
set -x
"${JAVA_HOME}/bin/java" \
      -Dosgi.install.area=${HOME} \
```

```
        -Dosgi.configuration.area=${HOME}/configuration \
        -Dosgi.debug=${HOME}/.options \
        "-classpath" "${HOME}/lib/startup.jar" \
        "-Dwas.install.root=${HOME}" \
        org.eclipse.core.launcher.Main -nosplash -debug -clean \
        -application \
        <app name>
exit $?
```

Auditing invocations of the wsadmin tool

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:

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” on page 678 article. For more information about starting the wsadmin tool, see the “Starting the wsadmin scripting client” on page 135 article. 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.

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
```

Troubleshooting with scripting

Use these topics to learn more about troubleshooting with scripting.

This topic contains the following tasks:

- "Enabling the Runtime Performance Advisor tool using scripting" on page 666

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 information see the Tracing and logging configuration article and the Log level settings article in the *Troubleshooting and support* PDF.

Tracing operations with the wsadmin tool

You can enable tracing with scripting and the wsadmin tool.

Before starting this task, the wsadmin tool must be running. See the "Starting the wsadmin scripting client" on page 135 article for more information.

Perform the following steps to trace operations:

Enable wsadmin client tracing with the following command:

- Using Jacl:

  ```
  $AdminControl trace com.ibm.*=all=enabled
  ```

- Using Jython:

  ```
  AdminControl.trace('com.ibm.*=all=enabled')
  ```

where:

`$`	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.*=all=enabled`	indicates to turn on tracing

The following command disables tracing:

- Using Jacl:

  ```
  $AdminControl trace com.ibm.*=all=disabled
  ```

- Using Jython:

  ```
  AdminControl.trace('com.ibm.*=all=disabled')
  ```

where:

`$`	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.*=all=disabled`	indicates to turn off tracing

The trace command changes the trace settings for the current session. You can change this setting persistently by editting 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.

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.

Configuring traces using scripting

Use the wsadmin tool and scripting to configure traces for a configured server.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to set the trace for a configured server:

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” on page 116 article for more information.
6. In a network deployment environment only, synchronize the node. See the “Synchronizing nodes with the wsadmin tool” on page 99 article for more 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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Perform the following steps to turn traces on and off in server processes:

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 starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

The following example produces a Java core file. You can use this file for problem determination.

- 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')
  ```
  For information about the environment variables that deal with dumping threads, refer to the Java Diagnostics guide.

Setting up profile scripts to make tracing easier using scripting

You can use scripting and the wsadmin tool to set up profile scripts to facilitate tracing.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

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 “Starting the wsadmin scripting client” on page 135 article.

Enabling the Runtime Performance Advisor tool using scripting

You can configure the Runtime Performance Advisor using the wsadmin tool or the administrative console.

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

The Runtime Performance Advisor tool provides advice to help tune systems for optimal performance. See the Using the Runtime Performance Advisor article in the *Tuning guide* PDF for more information on how to enable this tool using the administrative console. The recommendations display as text in the SystemOut.log file.

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 RPAr 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:

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
```

After completing the previous steps, start the server and monitor RPA.

Commands for the EventServiceCommands group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the EventServiceCommands group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:
deployEvent Service	The **deployEvent Service** command deploys and configures the event service on a server or cluster.	None	• Parameters: **- nodeName** The name of the node where the event service should be deployed. If this parameter is specified, then the serverName parameter must be specified. You must not specify this parameter if the clusterName parameter is specified. **- serverName** The name of the server where the event service should be deployed. You must specify this parameter if the **nodeName** parameter is specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service should be deployed. You must not specify this parameter if the **nodeName** or **serverName** parameter are specified. **- enable** Set this parameter to `true` if you want the event service to be started after the next restart of the WebSphere server. The default value is `true`. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask deployEventService {-nodeName nodename -server Name servername}` `$AdminTask deployEventService {-clusterName clustername -enab le false}` • Using Jython string: `AdminTask.deployEventService ('[-nodeName nodename -server Name servername]')` `AdminTask.deployEventService ('[-clusterName clustername -enable false]')` • Using Jython list: `AdminTask.deployEventService (['-nodeName', 'nodename', '-serverName', '-servername'])` `AdminTask.deployEventService (['-clusterName', 'clustername', '-enable', 'false]')` **Interactive mode example usage:** • Using Jacl: `$AdminTask deployEventS ervice {-interactive}` • Using Jython string: `AdminTask.deployEvent Service('[-interactive]')` • Using Jython list: `AdminTask.deployEvent Service(['-interactive'])`

deployEvent ServiceMdb	The **deployEvent ServiceMdb** command deploys the event service MDB to a server or cluster.	None	• Parameters: **- nodeName** The name of the node where the event service MDB should be deployed. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service MDB should be deployed. You must specify this parameter if the **nodeName** parameter is specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service MDB should be deployed. You must not specify this parameter if the **nodeName** and **serverName** parameters are specified. **- applicationName** The name of the event service MDB application to be deployed on a server or cluster. **- listenerPort** The name of the listener port where the event service MDB should publish the events. The listener port must already be created. You must not specify this parameter if the **activationSpec** parameter is specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask deployEventService Mdb {-applicationName appname -nodeName nodename -serverName servername -listenerPort lpname}` `$AdminTask deployEventService Mdb {-applicationName appname -clusterName clustername -act ivationSpec asjndiname -qcfJnd iName qcfjndiname}` • Using Jython string: `AdminTask.deployEventServiceMdb ('[-applicationName appname -nodeName nodename -serverName servername -listenerPort lpname]')` `AdminTask.deployEventServiceMdb ('[-applicationName appname -clusterName clustername -acti vationSpec asjndiname -qcfJndi Name qcfjndiname]')` • Using Jython list: `AdminTask.deployEventServiceMdb (['-applicationName', 'appname', '-nodeName', 'nodename', '-ser verName', '-servername', ' -listenerPort', 'lpname'])` `AdminTask.deployEventServiceMdb (['-applicationName', 'appname', '-clusterName', 'clustername', '-activationSpec', 'asjndiname', '-qcfJndiName', 'qcfjndiname'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask deployEvent ServiceMdb {-interactive}` • Using Jython string: `AdminTask.deployEventSe rviceMdb('[-interactive]')` • Using Jython list: `AdminTask.deployEventSer viceMdb(['-interactive'])`

			- activationSpec The JNDI name of activation specification where the event service MDB should publish the events. The activation specification must already be created. You must not specify this parameter if the **listenerPort** parameter is specified. **- qcfJndiName** The JNDI name of the JMS queue connection factory object to be used by the event service MDB. You must specify this parameter if the **activationSpec** parameter is specified. • Returns: None	

disableEvent Service	The **disableEvent Service** command disables the event service from starting after the next restart of the WebSphere server(s) designated by the nodeName, serverName, or clusterName parameters.	None	• Parameters: **- nodeName** The name of the node where the event service should be disabled. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service should be disabled. You must specify this parameter if the **nodeName** parameter is specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service should be disabled. You must not specify this parameter if the **nodeName** and **serverName** parameters are specified. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask disableEvent Service {-nodeName node name -serverName servername}` `$AdminTask disableEventServ ice {-clusterName clustername}` • Using Jython string: `AdminTask.disableEventService ('[-nodeName nodename -server Name servername]')` `AdminTask.disableEventService ('[-clusterName clustername]')` • Using Jython list: `AdminTask.disableEventService (['-nodeName', 'nodename', '-serverName', '-servername'])` `AdminTask.disableEventService (['-clusterName', 'clustername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask disableEvent Service {-interactive}` • Using Jython string: `AdminTask.disableEvent Service('[-interactive]')` • Using Jython list: `AdminTask.disableEvent Service(['-interactive'])`

enableEvent Service	The **enableEvent Service** command enables the event service to be started after the next restart of the WebSphere server(s) designated by the nodeName, serverName, or clusterName parameters.	None	• Parameters: **- nodeName** The name of the node where the event service should be enabled. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service should be enabled. You must specify this parameter if the **nodeName** parameter is specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service should be enabled. You must not specify this parameter if the **nodeName** and **serverName** parameters are specified. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask enableEventService {-nodeName nodename -serverName servername}` `$AdminTask enableEventService {-clusterName clustername}` • Using Jython string: `AdminTask.enableEventService ('[-nodeName nodename -server Name servername]')` `AdminTask.enableEventService ('[-clusterName clustername]')` • Using Jython list: `AdminTask.enableEventService (['-nodeName', 'nodename', '-serverName', '-servername'])` `AdminTask.enableEventService (['-clusterName', 'clustername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask enableEvent Service {-interactive}` • Using Jython string: `AdminTask.enableEvent Service('[-interactive]')` • Using Jython list: `AdminTask.enableEventS ervice(['-interactive'])`

removeEvent Service	The **removeEvent Service** command removes the event service from a server or cluster.	None	• Parameters: **- nodeName** The name of the node from which the event service should be removed. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server from which the event service should be removed. You must specify this parameter if the **nodeName** parameter is specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster from which the event service should be removed. You must not specify this parameter if the **nodeName** and **serverName** parameters are specified. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventSer vice {-nodeName nodename -serverName servername}` `$AdminTask removeEventSer vice {-clusterName cluster name}` • Using Jython string: `AdminTask.removeEventService ('[-nodeName nodename -server Name servername]')` `AdminTask.removeEventService ('[-clusterName clustername]')` • Using Jython list: `AdminTask.removeEventService (['-nodeName', 'nodename', '-serverName', '-servername'])` `AdminTask.removeEventService (['-clusterName', 'clustername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEvent Service {-interactive}` • Using Jython string: `AdminTask.removeEvent Service('[-interactive]')` • Using Jython list: `AdminTask.removeEvent Service(['-interactive'])`

removeEvent ServiceMdb	The **removeEvent ServiceMdb** command removes the event service MDB from a server or cluster.	None	• Parameters: **- nodeName** The name of the node where the event service MDB should be removed. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service MDB should be removed. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service MDB should be removed. You must not specify this parameter if the **nodeName** and **serverName** parameters are specified. **- applicationName** The name of the event service MDB application to be removed from a server or cluster. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask removeEventService Mdb {-applicationName appname -nodeName nodename -serverName servername}` `$AdminTask removeEventService Mdb {-applicationName appname -clusterName clustername}` • Using Jython string: `AdminTask.removeEventService Mdb('[-applicationName appname -nodeName nodename -serverName servername]')` `AdminTask.removeEventServiceMdb ('[-applicationName appname -clusterName clustername]')` • Using Jython list: `AdminTask.removeEventServiceMdb (['-applicationName', 'appname', '-nodeName', 'nodename', '-server Name', 'servername'])` `AdminTask.removeEventServiceMdb (['-applicationName', 'appname', '-clusterName', 'clustername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask removeEventServi ceMdb {-interactive}` • Using Jython string: `AdminTask.removeEventServi ceMdb('[-interactive]')` • Using Jython list: `AdminTask.removeEventServ iceMdb(['-interactive'])`

setEvent Service JmsAuth Alias	The **setEvent Service JmsAuth Alias** command updates the authentication alias used by the event service JMS objects on a server or cluster. If the JMS authentication alias does not exist, it is created.	None	• Parameters: **- nodeName** The name of the node where the event service JMS authentication alias should be updated. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** The name of the server where the event service JMS authentication alias should be updated. If this parameter is specified, then the **serverName** parameter must be specified. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** The name of the cluster where the event service JMS authentication alias should be updated. You must not specify this parameter if the **nodeName** and **serverName** parameters are specified. **- userName** The name of the user to be used in the update of the event service JMS authentication alias on a server or cluster. **- password** The password of the user to be used in the update of the event service JMS authentication alias on a server or cluster. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setEventService JmsAuthAlias{-nodeName node name -serverName servername username -password pwd}` `$AdminTask setEventServiceJ msAuthAlias{-clusterName clustername -userName user name -password pwd}` • Using Jython string: `AdminTask.setEventServiceJms AuthAlias('[-nodeName nodename -serverName servername -user Name username -password pwd]')` `AdminTask.setEventServiceJms AuthAlias('[-clusterName clust ername -userName username -password pwd]')` • Using Jython list: `AdminTask.setEventServiceJmsAuth Alias(['-nodeName', 'nodename', '-serverName', '-servername', '-userName', 'username', '-pass word', 'pwd'])` `AdminTask.setEventServiceJmsAuth Alias(['-clusterName', 'cluster name', '-userName', 'username', '-password', 'pwd'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setEventService JmsAuthAlias {-interactive}` • Using Jython string: `AdminTask.setEventService JmsAuthAlias( '[-interactive]')` • Using Jython list: `AdminTask.setEventServiceJms AuthAlias(['-interactive'])`

showEvent ServiceStatus	The **showEvent ServiceStatus** command returns the status of the event service in a server or cluster. If the task is executed with no parameters, the status of all event services is displayed. To filter the list of event services to be displayed, provide nodeName, serverName, or clusterName.	None	• Parameters: **- nodeName** Use this parameter to display only the status of the event services that belong to the specified node. You must not specify this parameter if the **clusterName** parameter is specified. **- serverName** Use this parameter to display only the status of the event services that belong to the specified server. You can use this parameter with the **nodeName** parameter to display the status of the event service belonging to the specified node and server. You must not specify this parameter if the **clusterName** parameter is specified. **- clusterName** Use this parameter to only display the status of the event services that belong to the specified cluster. You must not specify this parameter if the **nodeName** or **serverName** parameters are specified. • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask showEventService Status {-nodeName nodename -serverName servername}` `$AdminTask showEventService Status {-clusterName cluster name}` • Using Jython string: `AdminTask.showEventService Status('[-nodeName nodename -serverName servername]')` `AdminTask.showEventService Status('[-clusterName clustername]')` • Using Jython list: `AdminTask.showEventService Status(['-nodeName', 'nodename', '-serverName', '-servername'])` `AdminTask.showEventServiceStatus (['-clusterName', 'clustername'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask showEventServi ceStatus {-interactive}` • Using Jython string: `AdminTask.showEventServic eStatus('[-interactive]')` • Using Jython list: `AdminTask.showEventService Status(['-interactive'])`

Commands for the AdministrationReports group of the AdminTask object

For more information about the AdminTask object, see the Commands for the AdminTask object article.

The following commands are available for the AdministrationReports group of the AdminTask object:

Command name:	Description:	Target object:	Parameters and return values:	Examples:

reportConfigInconsistencies	Use the **reportConfigInconsistencies** command to create a report of inconsistencies in the system configuration.	None	• Parameters: None • Returns: A report that describes inconsistencies found in the system.	**Interactive mode example usage:** • Using Jacl: `$AdminTask reportConfigInconsistencies {-interactive}` • Using Jython string: `AdminTask.reportConfigInconsistencies ('[-interactive]')` • Using Jython list: `AdminTask.reportConfigInconsistencies (['-interactive'])` **Example output:** `Configuration consistency report for cell yardbirdCell` `cells/yardbirdCell/test.xml is a zero length file.` `cells/yardbirdCell/nodes/DummyNode does not contain a serverindex.xml document.` `cells/yardbirdCell/applications/Test.ear/deployments/Test does not contain a deployment.xml document.` `3 consistency problems were found.`
reportConfiguredPorts	Use the **reportConfiguredPorts** command to create a report of all the ports configured in the cell.	None	• Parameters: None • Returns: A report that describes the port usage in the system.	**Interactive mode example usage:** • Using Jacl: `$AdminTask reportConfiguredPorts {-interactive}` • Using Jython string: `AdminTask.reportConfiguredPorts ('[-interactive]')` • Using Jython list: `AdminTask.reportConfiguredPorts (['-interactive'])` **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`

Scripting reference material

The following tasks are reference material for the wsadmin tool and scripting.

This topic contains the following tasks:

Wsadmin tool

The WebSphere Application Server wsadmin tool runs scripts. You can use the wsadmin tool to manage WebSphere Application Server as well as the configuration, application deployment, and server run-time operations.

You can run wsadmin commands from WebSphere Application Server on a distributed platform and use that distributed command to administer WebSphere Application Server on a z/OS platform.

The options for the wsadmin tool are case insensitive.

The command-line invocation syntax for the wsadmin scripting client is as follows:

```
wsadmin.sh [-h(help)]

[-?]

[-c <commands>]

[-p <properties_file_name>]

[-profile <profile_script_name>]

[-profileName <profile_name>]

[-f <script_file_name>]

[-javaoption java_option]

[-lang language]

[-wsadmin_classpath classpath]

[-conntype SOAP [-host host_name]

 [-port port_number] [-user userid] [-password password] |

 RMI [-host host_name] [-port port_number] [-user userid] [-password password] | NONE]
[-jobid string]

[-tracefile trace_file]

[-appendtrace true/false]
[script parameters]
```

Where *script parameters* represent any arguments other than the ones listed previously. The argc variable contains the argument number, and the argv variable contains the contents.

Options

-c Designates to run a single command.

Multiple -c options can exist on the command line. They run in the order that you designate. You must save after using this command.

-f Designates a script to run.

Only one -f option can exist on the command line.

-javaoption
Specifies a valid Java standard or a non-standard option. Multiple -javaoption options can exist on the command line.

To shorten the length of the command, type it the following way:

```
wsadmin.sh -javaoption java_option  java_option
```

instead of the following:

```
wsadmin.sh -javaoption java_option -javaoption java_option
```

-lang
Specifies the language of the script file, the command, or an interactive shell. The possible languages include: Jacl and Jython. The options for the -lang argument include: 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, 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, an error message generates.

-p
Specifies a properties file.

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 specify on the command line. The first level is the installation default, `wsadmin.properties`, which is located in the WebSphere Application Server `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 that the environment variable `WSADMIN_PROPERTIES` points to.

Multiple -p options can exist on the command line. They invoke in the order that you supply them.

-profile
Specifies a profile script.

The profile script runs before other commands, or scripts. If you specify `-c`, the profile script runs before it invokes this command. If you specify `-f`, 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 invoke in the order that you supply them.

-profileName
Specifies the profile from which the wsadmin tool will run. Specify this option if one the following reasons apply:

- You run the wsadmin tool from the `WAS_HOME/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.

  Note: WebSphere Application Server for z/OS does not support user-created profiles; only the default profile is used.

-?
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`, and `NONE`.

The options SOAP and RMI for the `-conntype` argument also include: host, port, user, and password.

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.

-wsadmin_classpath
Use this option to make additional classes available to your scripting process.

Follow this option with a class path string. For example:

```
/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 hostname to which wsadmin should attempt to connect. The default wsadmin.properties file located in the properties directory of each WebSphere 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.

Warning: On UNIX system, the use of -password option may result in security exposure as the password information becomes visible to the system status program such as ps command which can be invoked by other user to display all the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the soap.client.props file for SOAP connector or sas.client.props file for RMI connector. The soap.client.props and sas.client.props files are located in the properties directory of your WebSphere profile.

-username
Specify a user name to be used by the connector to connect to the server if security is enabled in the server.

-port
Specify a port to be used by the connector. The default wsadmin.properties file located in the properties directory of each WebSphere Application Server profile provides a value in the port property to connect to the local server.

-jobid
Specifies a jobID string so that you can keep track of each invocation of the wsadmin tool for auditing purposes. The jobID string (jobID=xxxx) displays 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 following example specifies the jobID 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:

```
wsadmin.sh -f test1.jacl -profile setup.jacl -conntype SOAP
-port mymachinesoapportnumber -host mymachine
```

Using Jython:

```
wsadmin.sh -lang jython -f  test1.py -profile setup.py -conntype
SOAP -port mymachinesoapportnumber -host mymachine
```

Initial and maximum Java heap size
Using Jacl:

```
wsadmin.sh -javaoption -Xms128m -Xmx256m -f test.jacl
```

Using Jython:

```
wsadmin.sh -lang jython -javaoption -Xms128m -Xmx256m -f test.py
```

RMI connection with security
Using Jacl:

```
wsadmin.sh -conntype RMI -port  rmiportnumber -userid userid
-password password
```

Using Jython:

```
wsadmin.sh -lang jython -conntype RMI -port  rmiportnumber -userid userid
-password password
```

Warning: On UNIX system, the use of -password option may result in security exposure as the password information becomes visible to the system status program such as ps command which can be invoked by other user to display all the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the soap.client.props file for SOAP connector or sas.client.props file for RMI connector. The soap.client.props and sas.client.props files are located in the properties directory of your WebSphere profile.

Local mode of operation to perform a single command
Using Jacl:

```
wsadmin.sh -conntype NONE -c "\$AdminApp uninstall app"
```

or

```
wsadmin.sh -conntype NONE -c '$AdminApp uninstall app'
```

Using Jython:

```
wsadmin.sh -lang jython -conntype NONE -c "\AdminApp.uninstall('app')"
```

or

```
wsadmin.sh -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.sh -c "\$AdminApp install /myApps/App1.ear {-appname appl1}"
  wsadmin.sh -c "\$AdminApp install /myApps/App2.ear {-appname appl2}"
  wsadmin.sh -c "\$AdminApp install /myApps/App3.ear {-appname appl3}"
  ```

  or

  ```
  wsadmin.sh -c '$AdminApp install /myApps/App1.ear {-appname appl1}'
  wsadmin.sh -c '$AdminApp install /myApps/App2.ear {-appname appl2}'
  wsadmin.sh -c '$AdminApp install /myApps/App3.ear {-appname appl3}'
  ```

  Using Jython:

  ```
  wsadmin.sh -lang jython -c "\AdminApp.install('/myApps/App1.ear', '[-appname appl1]')"
  wsadmin.sh -lang jython -c "\AdminApp.install('/myApps/App2.ear', '[-appname appl2]')"
  wsadmin.sh -lang jython -c "\AdminApp.install('/myApps/App3.ear', '[-appname appl3]')"
  ```

  or

  ```
  wsadmin.sh -lang jython -c 'AdminApp.install('/myApps/App1.ear', '[-appname appl1]')'
  wsadmin.sh -lang jython -c 'AdminApp.install('/myApps/App2.ear', '[-appname appl2]')'
  wsadmin.sh -lang jython -c 'AdminApp.install('/myApps/App3.ear', '[-appname appl3]')'
  ```

  Or, for example, using Jacl, you can create the *appinst.jacl* file that contains the commands:

  ```
  $AdminApp install /myApps/App1.ear {-appname appl1}
  $AdminApp install /myApps/App2.ear {-appname appl2}
  $AdminApp install /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('/myApps/App1.ear', '[-appname appl1]')
  AdminApp.install('/myApps/App2.ear', '[-appname appl2]')
  AdminApp.install('/myApps/App3.ear', '[-appname appl3]')
  ```

  Then invoke this file using the following command: `wsadmin.sh -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 `$AdminControl queryNames *` command performs well. If a scripting client connects to the deployment manager in a multiple machine environment, use a command only if it is necessary for the script to obtain a list of all the MBeans in the system. If you need the MBeans on a node, it is easier to invoke `"$AdminControl queryNames node=mynode,*"`. The JMX system management infrastructure forwards requests to the system to fulfill the first query, `*`. The second query, `node=mynode,*` is targeted to a specific machine.
- 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

The Help object provides general help and dynamic online information about the currently running MBeans. You can 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:

Command name:	Description:	Parameters and return values:	Examples:

AdminApp	Provides a summary of all of the available methods for the AdminApp object.	• Parameters: none • Returns: string	Example usage: Using Jacl: `$Help AdminApp` Using Jython: `print Help.AdminApp()` Example 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 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 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

AdminConfig	Provides a summary of all the available methods for the AdminConfig object.	• Parameters: None • Returns: string	Example usage: Using Jacl: `$Help AdminConfig` Using Jython: `print Help.AdminConfig()` Example 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`

AdminControl	Provides a summary of the help commands and ways to invoke an administrative command.	• Parameters: None • Returns: string	Example usage: Using Jacl: `$Help AdminControl` Using Jython: `print Help.AdminControl()` Example 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

AdminTask	Provides a summary of help commands and ways to invoke an administrative command.	• Parameters: None • Returns: string	Example usage: Using Jacl: `$AdminTask help` Using Jython: `print AdminTask.help()` Example output: `WASX8001I: The AdminTask object enables the available administrative commands. AdminTask commands operate in two modes: the default mode is one which AdminTask communicates with the WebSphere Application 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 administrative commands varies and depends on your WebSphere Application Server 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.

all	Provides a summary of the information that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Name: WebSphere:cell=pongo,name=Trace Service,mbeanIdentifier=cells/ pongo/nodes/pongo/servers/server1/ server.xml#TraceService_1, type=TraceService,node=pongo,process= server1` `Description: null` `Class name: javax.management.modelm bean.RequiredModelMBean` `Attribute Type` `ringBufferSize int` `traceSpecification java.lang.String` `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`

attributes	Provides a summary of all the attributes that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Attribute Type Access` `ringBufferSize java.lang.Integer RW` `traceSpecification string RW`
classname	Provides a class name that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `javax.management.modelmbean.RequiredModelMBean`
constructors	Provides a summary of all of the constructors that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Constructors`
description	Provides a description that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: 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,*'))` Example output: `Managed object for overall server process.`

help	Provides a summary of all the available methods for the Help object.	• Parameters: None • Returns: string	Example output: WASX7028I: The Help object has two purposes: First, provide general help information for the bjects 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

message	Displays information for a message ID.	• Parameters: message ID • Returns: string	Example usage: Using Jacl: `$Help message CNTR0005W` Using Jython: `print Help.message('CNTR0005W')` Example output: `Explanation: The container was unable to passivate an enterprise bean due to exception {2}` `User action: Take action based upon message in exception {2}`
notifications	Provides a summary of all the notifications that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: Using Jacl: `$Help notifications [$AdminControl queryNames type=TraceService,process=server1,node=pongo,*]` Using Jython: `print Help.notifications(AdminControl.queryNames('type=TraceService,process=server1,node=pongo,*'))` Example output: `Notification` `websphere.messageEvent.audit` `websphere.messageEvent.fatal` `websphere.messageEvent.error` `websphere.seriousEvent.info` `websphere.messageEvent.warning` `jmx.attribute.changed`

operations	Provides a summary of all the operations that the MBean defines by name.	• Parameters: name - string • Returns: string	Example usage: Using Jacl: `$Help operations [$AdminControl query Names type=TraceService,process=server1, node=pongo,*]` Using Jython: `print Help.operations(AdminControl.query Names('type=TraceService,process=server1, node=pongo,*'))` Example output: `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)`
operations	Provides the signature of the opname operation for the MBean that is defined by name.	• Parameters: name - string, opname - string • Returns: string	Example usage: Using Jacl: `$Help operations [$AdminControl queryNames type=TraceService,process=server1,node= pongo,*] processDumpString` Using Jython: `print Help.operations(AdminControl.query Names('type=TraceService,process=server1, node=pongo,*'), 'processDumpString')` Example output: `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.`

Related concepts

"Help object for scripted administration" on page 88
The Help object provides general help, online information about running MBeans, and help on messages.

Commands for the AdminConfig object

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, running the AdminConfig tool in local mode 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. In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

The following commands are available for the AdminConfig object:

Command name:	Description:	Parameters and return values:	Examples:
attributes	Returns a list of the top level attributes for a given type.	• Parameters: object type The name of the object type that you input here is the one based on the XML configuration files and does not have to be the same name that the administrative console displays. • Returns: A list of attributes.	Example usage: Using Jacl: `$AdminConfig attributes ApplicationServer` Using Jython: `print AdminConfig.attributes('ApplicationServer')` Example output: `"properties Property*" "serverSecurity ServerSecurity"` `"server Server@" "id Long" "stateManagement StateManageable"` `"name String" "moduleVisibility EEnumLiteral (MODULE,` `COMPATIBILITY, SERVER, APPLICATION)"` `"services Service*"` `"statisticsProvider StatisticsProvider"`

checkin	Checks a file that the document URI describes into the configuration repository. This method only applies to deployment manager configurations.	• Parameters: document URI, filename, opaque object • Returns: None	Example usage: Using Jacl: `$AdminConfig checkin cells/MyCell/Node/MyNode/ serverindex.xml \mydir\myfile $obj` Using Jython: `AdminConfig.checkin('cells/MyCell/Node/MyNode/ serverindex.xml', '\mydir\myfile', obj)` The document URI is relative to the root of the configuration repository, for example, `\WebSphere\AppServer\config`. The file that is specified by `filename` is used as the source of the file to check. The *opaque* object is an object that the **extract** command of the AdminConfig object returns by a prior call.
convertToCluster	Converts a server so that it is the first member of a new server cluster.	• Parameters: server ID, cluster name • Returns: The configuration ID of the new cluster.	Example usage: Using Jacl: `set serverid [$AdminConfig getid /Server:myServer/] $AdminConfig convertToCluster $serverid myCluster` Using Jython: `serverid = AdminConfig.getid('/Server:myServer/') AdminConfig.convertToCluster(serverid, 'myCluster')` Example output: `myCluster(cells/mycell/clusters/ myCluster|cluster.xml#ClusterMember_2`
create	Creates configuration objects.	• Parameters using Jacl: type- string; parent ID- string; attributes- string • Parameters using Jython: type- string; parent ID- string; attributes- string or type- string; parent ID- string; attributes- Jython list • Returns: A string with configuration object names.	The name of the object type that you input here is the one that is based on the XML configuration files. This name does not have to be the same name that the administrative console displays. Example usage: Using Jacl: `set jdbc1 [$AdminConfig getid /JDBCProvider:jdbc1/] $AdminConfig create DataSource $jdbc1 {{name ds1}}` Using Jython with string attributes: `jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/') AdminConfig.create('DataSource', jdbc1, '[[name ds1]]')` Using Jython with object attributes: `jdbc1 = AdminConfig.getid('/JDBCProvider:jdbc1/') AdminConfig.create('DataSource', jdbc1, [['name', 'ds1']])` Example output: `ds1(cells/mycell/nodes/DefaultNode/servers/ server1|resources.xml#DataSource_6)`

createClusterMember	Creates a new server as a member of an existing cluster. This method creates 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.	• Parameters using Jacl: cluster ID- string; node ID- string; member attributes- string • Parameters using Jython: cluster ID- string; node ID- string; member attributes- string or cluster ID- string; node ID- string; member attributes- Jython list • Returns: The configuration ID of the new cluster member.	The name of the object type that you input here is the one that is based on the XML configuration files. This name does not have to be the same name that the administrative console displays. Example usage: Using Jacl: `set clid [$AdminConfig getid /ServerCluster:myCluster/]` `set nodeid [$AdminConfig getid /Node:mynode/]` `$AdminConfig createClusterMember $clid $nodeid {{memberName newMem1} {weight 5}}` Using Jython with string attributes: `clid = AdminConfig.getid('/ServerCluster:myCluster/')` `nodeid = AdminConfig.getid('/Node:mynode/')` `AdminConfig.createClusterMember(clid, nodeid, '[[memberName newMem1] [weight 5]]')` Using Jython with object attributes: `clid = AdminConfig.getid('/ServerCluster:myCluster/')` `nodeid = AdminConfig.getid('/Node:mynode/')` `AdminConfig.createClusterMember(clid, nodeid, [['memberName', 'newMem1'], ['weight', 5]])` Example output: `myCluster(cells/mycell/clusters/myCluster|cluster.xml#ClusterMember_2)`

createDocument	Creates a new document in the configuration repository. The documentURI parameter names the document to create in the repository. The filename parameter must be a valid local file name where the contents of the document exist.	• Parameters: documentURI, filename • Returns: None	Example usage: Using Jacl: `$AdminConfig createDocument cells/mycell/ myfile.xml \mydir\\myfile` Using Jython: `AdminConfig.createDocument('cells/mycell /myfile.xml', '\mydir\myfile')`
createUsing Template	Creates a type of object with the given parent, using a template.	• Parameters using Jacl: type-string; parent id-string; attributes-string; template ID-string • Parameters using Jython: type-string; parent id- string; attributes-string; template ID-string or type-string; parent id-string; attributes-Jython list; template ID-string • Returns: The configuration ID of a new object.	Example usage: Using Jacl: `set node [$AdminConfig getid /Node:`*`mynode`*`/ ]` `set templ [$AdminConfig listTemplates JDBCProvider "DB2 JDBC Provider (XA)"]` `$AdminConfig createUsingTemplate JDBCProvider $node {{name`*`newdriver`*`}} $templ` Using Jython using string attributes: `node = AdminConfig.getid('/Node:mynode/')` `templ = AdminConfig.listTemplates( 'JDBCProvider', "DB2 JDBC Provider (XA)")` `AdminConfig.createUsingTemplate( 'JDBCProvider', node, '[[name newdriver]]', templ)` Using Jython using object attributes: `node = AdminConfig.getid('/Node:mynode/')` `templ = AdminConfig.listTemplates( 'JDBCProvider', "DB2 JDBC Provider (XA)")` `AdminConfig.createUsingTemplate ('JDBCProvider', node, [['name', 'newdriver']], templ)`

defaults	Displays 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.	• Parameters: type The name of the object type that you input here is the one based on the XML configuration files. This name does not have to be the same name that the administrative console displays. • Returns: A string that contains a list of attributes with its type and value.	Example usage: Using Jacl: `$AdminConfig defaults TuningParams` Using Jython: `print AdminConfig.defaults('TuningParams')` Example output: `Attribute Type Default` `usingMultiRowSchema Boolean false` `maxInMemorySessionCount Integer 1000` `allowOverflow Boolean true` `scheduleInvalidation Boolean false` `writeFrequency ENUM` `writeInterval Integer 120` `writeContents ENUM` `invalidationTimeout Integer 30` `invalidationSchedule InvalidationSchedule`
deleteDocument	Deletes a document from the configuration repository. The documentURI parameter names the document to delete from the repository.	• Parameters: documentURI • Returns: None	Example usage: Using Jacl: `$AdminConfig deleteDocument cells/mycell/myfile.xml` Using Jython: `AdminConfig.deleteDocument('cells/mycell/myfile.xml')`
existsDocument	Tests for the existence of a document in the configuration repository. The documentURI parameter names the document to test in the repository.	• Parameters: documentURI • Returns: A true value, if the document exists.	Example usage: Using Jacl: `$AdminConfig existsDocument cells/mycell/myfile.xml` Using Jython: `AdminConfig.existsDocument('cells/mycell/myfile.xml')` Example output: `1`

extract	Extracts 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.	• Parameters: document URI, filename • Returns: An opaque java.lang.Object to use when checking in the file.	Example usage: Using Jacl: `set obj [$AdminConfig extract cells/MyCell/nodes/MyNode/serverindex.xml \mydir\myfile]` Using Jython: `obj = AdminConfig.extract('cells/MyCell/nodes/MyNode/serverindex.xml', '\mydir\myfile')` The document URI is relative to the root of the configuration repository, for example, `/WebSphere/AppServer/config`. If the file that is specified by the filename parameter exists, the extracted file replaces it.
`getCrossDocumentValidationEnabled`	Returns a message with the current cross-document enablement setting. This method returns true if cross-document validation is enabled.	• Parameters: None • Returns: A string that contains the message with the cross-document validation setting.	Example usage: Using Jacl: `$AdminConfig getCrossDocumentValidationEnabled` Using Jython: `print AdminConfig.getCrossDocumentValidationEnabled()` Example output: `WASX7188I: Cross-document validation enablement set to true`
getid	Returns the configuration ID of an object.	• Parameters: containment path • Returns: The configuration ID for an object that is described by the containment path.	Example usage: Using Jacl: `$AdminConfig getid /Cell:testcell/Node:testNode/JDBCProvider:Db2JdbcDriver/` Using Jython: `AdminConfig.getid('/Cell:testcell/Node:testNode/JDBCProvider:Db2JdbcDriver/')` Example output: `Db2JdbcDriver(cells/testcell/nodes/testnode|resources.xml#JDBCProvider_1)`
getObjectName	Returns a string version of the object name for the corresponding running MBean. This method returns an empty string if no corresponding running MBean exists.	• Parameters: configuration ID • Returns: A string that contains the object name.	Example usage: Using Jacl: `set server [$AdminConfig getid /Node:mynode/Server:server1/]` `$AdminConfig getObjectName $server` Using Jython: `server = AdminConfig.getid('/Node:mynode/Server:server1/')` `AdminConfig.getObjectName(server)` Example output: `WebSphere:cell=mycell,name=server1,mbeanIdentifier=cells/mycell/nodes/mynode/servers/server1/server.xml#Server_1,type=Server,node=mynode,process=server1,processType=UnManagedProcess`

getSaveMode	Returns the mode that is used when you invoke a **save** command. Possible values include the following: • overwriteOnConflict - Saves changes even if they conflict with other configuration changes • rollbackOnConflict - Causes a save operation to fail if changes conflict with other configuration changes. This value is the default.	• Parameters: None • Returns: A string that contains the current save mode setting.	Example usage: Using Jacl: `$AdminConfig getSaveMode` Using Jython: `print AdminConfig.getSaveMode()` Example output: `rollbackOnConflict`
getValidationLevel	Returns the validation used when files are extracted from the repository.	• Parameters: None • Returns: A string that contains the validation level.	Example usage: Using Jacl: `$AdminConfig getValidationLevel` Using Jython: `AdminConfig.getValidationLevel()` Example output: `WASX7189I: Validation level set to HIGH`
`get Validation Severity Result`	Returns the number of validation messages with the given severity from the most recent validation.	• Parameters: severity • Returns: A string that indicates the number of validation messages of the given severity.	Example usage: Using Jacl: `$AdminConfig getValidationSeverityResult 1` Using Jython: `AdminConfig.getValidationSeverityResult(1)` Example output: `16`

hasChanges	Returns `true` if unsaved configuration changes exist.	• Parameters: None • Returns: A string that indicates whether unsaved configuration changes exist.	Example usage: Using Jacl: `$AdminConfig hasChanges` Using Jython: `AdminConfig.hasChanges()` Example output: `1`

help	Displays static help information for the AdminConfig object.	• Parameters: None • Returns: A list of options.	Example usage: Using Jacl: `$AdminConfig help` Using Jython: `print AdminConfig.help()` Example output: `WASX7053I: The AdminConfig object communicates with the configuration service in a WebSphere Application Server 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

`installResourceAdapter`	Installs a Java 2 Connector (J2C) resource adapter with the given Resource Adapter Archive (RAR) file name and an option string in the node. The RAR file name is the fully qualified file name that resides in the node that you specify. 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.	• Parameters: RAR file name, node, options • Returns: The configuration ID of the new J2CResourceAdapter object.	Example usage: 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"]')` Example output: `myResourceAdapter(cells/mycell/nodes/mynode|resources.xml#J2CResourceAdapter_1)`

	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 `{}`. 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.		

list	Returns a list of objects of a given type, possibly scoped by a parent.	• Parameters: 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 the administrative console displays. • Returns: A list of objects.	Example usage: Using Jacl: `$AdminConfig list JDBCProvider` Using Jython: `print AdminConfig.list('JDBCProvider')` Example output: `Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xml#JDBCProvider_1)` `Db2JdbcDriver(cells/mycell/nodes/DefaultNode/servers/deploymentmgr|resources.xml#JDBCProvider_1)` `Db2JdbcDriver(cells/mycell/nodes/DefaultNode/servers/nodeAgent|resources.xml#JDBCProvider_1)`
listTemplates	Displays a list of template object IDs.	• Parameters: object type The name of the object type that you input here is the one that is based on the XML configuration files and does not have to be the same name that the administrative console displays. • Returns: A list of template IDs.	Example usage: Using Jacl: `$AdminConfig listTemplates JDBCProvider` Using Jython: `print AdminConfig.listTemplates('JDBCProvider')` This example displays a list of all the JDBCProvider templates that are available on the system.
modify	Supports the modification of object attributes.	• Parameters using Jacl: object-string; attributes-string • Parameters using Jython: object-string; attributes-string or object-string; attributes-Jython list • Returns: None	Example usage: 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	Obtains information about object types.	• Parameters: 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 the administrative console displays. • Returns: A list of object types.	Example usage: Using Jacl: `$AdminConfig parents JDBCProvider` Using Jython: `AdminConfig.parents('JDBCProvider')` Example output: `Cell` `Node` `Server`
queryChanges	Returns a list of unsaved configuration files.	• Parameters: None • Returns: A string that contains a list of files with unsaved changes.	Example usage: Using Jacl: `$AdminConfig queryChanges` Using Jython: `AdminConfig.queryChanges()` Example output: `WASX7146I: The following configuration files contain unsaved changes:` `cells/mycell/nodes/mynode/servers/server1|resources.xml`
remove	Removes a configuration object.	• Parameters: Object • Returns: None	Example usage: Using Jacl: `$AdminConfig remove ds1(cells/mycell/nodes/DefaultNode /servers/ server1:resources.xml#DataSource_6)` Using Jython: `AdminConfig.remove('ds1(cells/mycell/nodes/DefaultNode /servers/ server1:resources.xml#DataSource_6)')`
required	Displays the required attributes that are contained by an object of a certain type.	• Parameters: Type The name of the object type that you input here is the one that is based on the XML configuration files. It does not have to be the same name that the administrative console displays. • Returns: A string that contains a list of the required attributes with its type.	Example usage: Using Jacl: `$AdminConfig required URLProvider` Using Jython: `print AdminConfig.required('URLProvider')` Example output: `Attribute Type` `streamHandlerClassName String` `protocol String`
reset	Resets the temporary workspace that holds updates to the configuration.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminConfig reset` Using Jython: `AdminConfig.reset()`

save	Saves changes in the configuration repository.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminConfig save` Using Jython: `AdminConfig.save()`
`setCross Document Validation Enabled`	Sets the cross-document validation enabled mode. Values include `true` or `false`.	• Parameters: Flag • Returns: None	Example usage: Using Jacl: `$AdminConfig setCrossDocumentValidationEnabled true` Using Jython: `AdminConfig.setCrossDocumentValidationEnabled('true')`
setSaveMode	Toggles the behavior of the **save** command. The default value is `rollbackOnConflict`. When a conflict is discovered 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.	• Parameters: Mode • Returns: None	Example usage: Using Jacl: `$AdminConfig setSaveMode overwriteOnConflict` Using Jython: `AdminConfig.setSaveMode('overwriteOnConflict')`
`setValidation Level`	Sets the validation that is used when files are extracted from the repository. Five validation levels are available: `none`, `low`, `medium`, `high`, or `highest`.	• Parameters: Level • Returns: A string that contains the validation level setting.	Example usage: Using Jacl: `$AdminConfig setValidationLevel high` Using Jython: `AdminConfig.setValidationLevel('high')` Example output: `WASX7189I: Validation level set to HIGH`

show	Returns the top-level attributes of the given object.	• Parameters: Object, attributes • Returns: A string that contains the attribute value.	Example usage: Using Jacl: `$AdminConfig show Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xm#JDBCProvider_1)` Example output with Jacl: `{name "Sample Datasource"} {description "Data source for the Sample entity beans"}` Using Jython: `print AdminConfig.show('Db2JdbcDriver(cells/mycell/nodes/DefaultNode|resources.xm#JDBCProvider_1)')` Example output with Jython: `[name "Sample Datasource"] [description "Data source for the Sample entity beans"]`

showall	Recursively shows the attributes of a given configuration object.	• Parameters: Object, attributes • Returns: A string that contains the attribute value.	Example usage: Using Jacl: `$AdminConfig showall "Default Datasource(cells/mycell/nodes/ DefaultNode/servers/server1:resources.xml#DataSource_1)` Example output with Jacl: `{authMechanismPreference BASIC_PASSWORD} {category default} {connectionPool {{agedTimeout 0} {connectionTimeout 1000} {maxConnections 30} {minConnections 1} {purgePolicy FailingConnectionOnly} {reapTime 180} {unusedTimeout 1800}}} {datasourceHelperClassname com.ibm.websphere. rsadapter.CloudscapeDataStoreHelper} {description "Datasource for the WebSphere Default Application"} {jndiName DefaultDatasource} {name "Default Datasource"} {propertySet {{resourceProperties {{{description "Location of Cloudscape 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 "Cloudscape 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}`  Using Jython: `AdminConfig.showall("Default Datasource(cells/mycell/nodes/ DefaultNode/servers/server1:resources.xml#DataSource_1)")`
			Example output with Jython: `[authMechanismPreference BASIC_PASSWORD] [category default] [connectionPool [[agedTimeout []] [connectionTimeout 1000] [maxConnections 30] [minConnections 1] [purgePolicy FailingConnectionOnly] [reapTime 180] [unusedTimeout 1800]]]`

			[datasourceHelperClassname com.ibm.websphere.rsadapter. CloudscapeDataStoreHelper] [description "Datasource for the WebSphere Default Application"] [jndiName DefaultDatasource] [name "Default Datasource"] [propertySet [[resourceProperties [[[description "Location of Cloudscape default database."] [name databaseName] [type string] [value ${WAS_INSTALL_ROOT}/bin/DefaultDB]] [[name remoteDataSourceProtocol] [type string] [value []]] [[name shutdownDatabase] [type string] [value []]] [[name dataSourceName] [type string] [value []]] [[name description] [type string] [value []]] [[name connectionAttributes] [type string] [value []]] [[name createDatabase] [type string] [value []]]]]]] [provider "Cloudscape JDBC Driver(cells/pongo /nodes/pongo/ servers/server1|resources.xml#JDBCProvider_1)"] [relationalResourceAdapter "WebSphere Relational Resource Adapter(cells/pongo/nodes/pongo/servers/server1| resources.xml#builtin_rra)"] [statementCacheSize 0]
showAttribute	Displays only the value for the single attribute that you specify. The output of this command is different from the output of the **show** command when a single attribute is specified. The **showAttribute** command does not display a list that contains the attribute name and value. It only displays the attribute value.	• Parameters: Configuration ID, attribute • Returns: A string that contains the attribute value.	Example usage: Using Jacl: `set ns [$AdminConfig getid /Node:mynode/]` `$AdminConfig showAttribute $ns hostName` Using Jython: `ns = AdminConfig.getid('/Node:mynode/')` `print AdminConfig.showAttribute(ns, 'hostName')` Example output: `mynode`

types	Returns a list of the configuration object types that you can manipulate.	• Parameters: None • Returns: A list of object types.	Example usage: Using Jacl: `$AdminConfig types` Using Jython: `print AdminConfig.types()` Example output: `AdminService` `Agent` `ApplicationConfig` `ApplicationDeployment` `ApplicationServer` `AuthMechanism` `AuthenticationTarget` `AuthorizationConfig` `AuthorizationProvider` `AuthorizationTableImpl` `BackupCluster` `CMPConnectionFactory` `CORBAObjectNameSpaceBinding` `Cell` `CellManager` `Classloader` `ClusterMember` `ClusteredTarget` `CommonSecureInteropComponent`

`uninstall Resource Adapter`	Uninstalls a Java 2 Connector (J2C) resource adapter with the given J2C resource adapter configuration ID and an option list. One option is valid for this command: * `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. 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.	• Parameters: J2C resource adapter configuration ID, list of options • Returns: The configuration ID of J2CResourceAdapter object that is removed.	Example usage: 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()` Example output: `WASX7397I: The following J2CResourceAdapter objects are removed:` `MyJ2CRA(cells/juniarti/nodes/juniarti|resources.xml #J2CResourceAdapter_1069433028609)`

validate	Invokes validation. This command requests 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 config id parameter.	• Parameters: config id (optional) • Returns: A string that contains results of the validation.	Example usage: Using Jacl: `$AdminConfig validate` Using Jython: `print AdminConfig.validate()` Example output: `WASX7193I: Validation results are logged in \WebSphere5\ AppServer\logs\wsadmin.valout: Total number of messages: 16` `WASX7194I: Number of messages of severity 1: 16`

Commands for the AdminControl object

Use the AdminControl object to invoke operational commands that deal with running objects in the WebSphere 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:

Command name:	Description:	Parameters and return values:	Examples:

completeObjectName	Creates 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 it finds several MBeans that match the fragment, the command returns the first one.	• Parameters: name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set serverON [$AdminControl completeObjectName node=mynode,type=Server,*]` Using Jython: `serverON = AdminControl.completeObjectName('node=mynode,type=Server,*')`
getAttribute	Returns the value of the attribute for the name that you provide.	• Parameters: name-java.lang.String; attribute-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]` `$AdminControl getAttribute $objNameString processType` Using Jython: `objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')` `AdminControl.getAttribute (objNameString, 'processType')`

getAttribute _jmx	Returns the value of the attribute for the name that you provide.	• Parameters: name-ObjectName; attribute-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] set objName [java::new javax.management.Object Name $objNameString] $AdminControl getAttribute_jmx $objName processType` Using Jython: `objNameString = AdminControl.complete ObjectName('WebSphere: type=Server,*') import javax. management as mgmt objName = mgmt.Object Name(objNameString) AdminControl.getAttribute _jmx(objName, 'processType')`
getAttributes	Returns the attribute values for the names that you provide.	• Parameters using Jacl: name-String; attributes-java.lang.String • Parameters using Jython: name-String; attributes-java.lang.String or name-String; attributes-java.lang.Object[] • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] $AdminControl getAttributes $objName String "cellName nodeName"` Using Jython with string attributes: `objNameString = Admin Control.completeObject name('WebSphere:type =Server,*) AdminControl.getAttributes (objNameString, '[cellName nodeName]')` Using Jython with object attributes: `objNameString = Admin Control.completeObject name('WebSphere:type =Server,*) AdminControl.get Attributes(objNameString, ['cellName', 'nodeName'])`

getAttributes _jmx	Returns the attribute values for the names that you provide.	• Parameters: name-ObjectName; attributes-java.lang.String[] • Returns: javax.management. AttributeList	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName WebSphere: type=Server,*] set objName [$AdminControl makeObjectName $objectNameString] set attrs [java::new {String[]} 2 {cellName nodeName}] $AdminControl getAttributes _jmx $objName $attrs` Using Jython: `objectNameString = AdminControl.complete ObjectName('type=Server,*') objName = AdminControl. makeObjectName (objectNameString) attrs = ['cellName', 'nodeName'] AdminControl.getAttributes _jmx(objName, attrs)`
getCell	Returns the name of the connected cell.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getCell` Using Jython: `AdminControl.getCell()` Example output: `Mycell`
getConfigId	Creates a configuration ID from an ObjectName or an ObjectName fragment. Use this ID with the `$AdminConfig` command. Not all MBeans that run have configuration objects that correspond. If several MBeans correspond to an ObjectName fragment, a warning is created and a configuration ID builds for the first MBean it finds.	• Parameters: name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set serverConfigId [$AdminControl getConfigId node=SY1,type=Server,*]` Using Jython: `serverConfigId = AdminControl. getConfigId('node=SY1, type=Server,*')`

getDefault Domain	Returns the default domain name from the server.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getDefaultDomain` Using Jython: `AdminControl.getDefaultDomain()` Example output: `WebSphere`
getDomain Name	Returns the domain name from the server.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getDomainName` Using Jython: `AdminControl.getDomainName()` Example output: `WebSphere`
getHost	Returns the name of your host.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getHost` Using Jython: `AdminControl.getHost()` Example output: `myhost`
getMBean Count	Returns the number of MBeans that are registered in the server.	• Parameters: None • Returns: java.lang.Integer	Example usage: Using Jacl: `$AdminControl getMBeanCount` Using Jython: `AdminControl.getMBeanCount()` Example output: `114`

getMBeanInfo _jmx	Returns 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** command.	• Parameters: name-ObjectName • Returns: javax.management. MBeanInfo	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName type=Server,*] set objName [$AdminControl makeObjectName $objectNameString] $AdminControl getMBeanInfo _jmx $objName` Using Jython: `objectNameString = AdminControl.complete ObjectName('type=Server,*') objName = AdminControl. makeObjectName (objectNameString) AdminControl.getMBeanInfo _jmx(objName)` Example output: `javax.management.modelmbean. ModelMBeanInfoSupport@ 10dd5f35`
getNode	Returns the name of the connected node.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getNode` Using Jython: `AdminControl.getNode()` Example output: `Myhost`

getObjectInstance	Returns the object instance that matches the input object name.	• Parameters: Object name • Returns: The object instance that matches the input object name.	Example usage: 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()` Example output: `javax.management.modelmbean.RequiredModelMBean`
getPort	Returns the name of your port.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getPort` Using Jython: `AdminControl.getPort()` Example output: `8877`

getProperties ForData Source	Deprecated, no replacement. This command incorrectly assumes the availability of a configuration service when running in connected mode.	• Parameters: configId-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set ds [lindex [$AdminConfig list DataSource] 0] $AdminControl getProper tiesForDataSource $ds` Using Jython: `ds = AdminConfig.list ('DataSource')` `# get line separator import java.lang. System as sys lineSeparator = sys. getProperty('line. separator')` `dsArray = ds.split (lineSeparator) AdminControl.getProper tiesForDataSource (dsArray[0])` Example output: `WASX7389E: Operation not supported - get PropertiesForDataSource command is not supported.`
getType	Returns the connection type.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl getType` Using Jython: `AdminControl.getType()` Example output: `SOAP`

help	Returns general help text for the AdminControl object.	• Parameters: None • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl help` Using Jython: `AdminControl.help()` Example output: `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 agents that 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 a template name getAttribute_jmx 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_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

			Show help information invoke_jmx Given ObjectName, name of command, array of parameters and signature, 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 server setAttribute_jmx 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_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

help	Returns help text for the specific command of the AdminControl object. The command name is not case sensitive.	• Parameters: command-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl help getAttribute` Using Jython: `AdminControl.help('getAttribute')` Example output: `WASX7043I: command: getAttribute Arguments: object name, attribute Description: Returns value of "attribute" for the MBean described by "object name."`
invoke	Invokes the object operation without any parameter. Returns the result of the invocation.	• Parameters: name-java.lang.String; operationName-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*] $AdminControl invoke $objNameString stop` Using Jython: `objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*') AdminControl.invoke(objNameString, 'stop')`

invoke	Invokes the object operation using the parameter list that you supply. The signature generates automatically. The types of parameters are supplied by examining the MBeanInfo that the MBean supplies. Returns the string result of the invocation. The string that is returned is controlled by the Mbean method that you invoked. If the Mbean method is synchronous, then control is returned back to the wsadmin tool only when the operation is complete. If the Mbean method is asynchronous, control is returned back to the wsadmin tool immediately even though the invoked task might not be complete.	• Parameters: name-java.lang.String; operationName-java.lang.String; params-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]` `$AdminControl invoke $objNameString appendTraceString com.ibm.*=all=enabled` Using Jython: `objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')` `AdminControl.invoke(objNameString, 'appendTraceString', 'com.ibm.*=all=enabled')`

invoke	Invokes the object operation by conforming the parameter list to the signature. Returns the result of the invocation.	• Parameters: name-java.lang.String; operationName-java.lang.String; params-java.lang.String; sigs-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere:type=Server,*]` `$AdminControl invoke $objNameString appendTraceString com.ibm.*=all=enabled java.lang.String` Using Jython: `objNameString = AdminControl.completeObjectName('WebSphere:type=Server,*')` `AdminControl.invoke (objNameString, 'appendTrace String', 'com.ibm.*=all=enabled', 'java.lang.String')`
invoke_jmx	Invokes the object operation by conforming the parameter list to the signature. Returns the result of the invocation.	• Parameters: name-ObjectName; operationName-java.lang.String; params-java.lang.Object[]; signature-java.lang.String[] • Returns: java.lang.Object	Example usage: `set objNameString [$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 appendTraceString $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']` `AdminControl.invoke_jmx (objName, 'appendTraceString', parms, signature)`

isRegistered	If the ObjectName value is registered in the server, then the value is `true`.	• Parameters: name-java.lang.String • Returns: Boolean	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere: type=Server,*] $AdminControl isRegistered $objNameString` Using Jython: `objNameString = AdminControl. completeObjectName('WebSphere: type=Server,*') AdminControl.isRegistered (objNameString)`
isRegistered _jmx	If the ObjectName value is registered in the server, then the value is `true`.	• Parameters: name-ObjectName • Returns: Boolean	Example usage: Using Jacl: `set objectNameString [$AdminControl completeObjectName type=Server,*] set objName [$AdminControl makeObjectName $objNameString] $AdminControl isRegistered_jmx $objName` Using Jython: `objectNameString = AdminControl. completeObjectName('type=Server,*') objName = AdminControl. makeObjectName(objectNameString) AdminControl.isRegistered_jmx (objName)`

makeObject Name	A convenience command that creates 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.	• Parameters: name-java.lang.String • Returns: javax.management. ObjectName	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName type=Server, node=mynode,*] set objName [$AdminControl makeObjectName $objNameString]` Using Jython: `objectNameString = AdminControl.completeObjec tName('type=Server, node=mynode,*') objName = AdminControl. makeObjectName (objectNameString)`

queryMBeans	Returns a list of object instances that match the object name that you provide.	• Parameters: Object name • Returns: A list of object instances.	Example usage: 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]` 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()` Example output: `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`

queryMBeans	Returns a list of object instances that match the object name and query that you provide.	• Parameters: object name (type ObjectName), query (type QueryExp) • Returns: A list of object instances.	Example usage: Using Jacl: `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,*',None)` Use the following example to manipulate the return value of the queryMBeans command: `appArray = apps.toArray()` `app1 = appArray[0]` `print app1.getObjectName().toString()` Example output: `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`
queryNames	Returns a string that lists all the ObjectName objects based on the name template.	• Parameters: name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl queryNames WebSphere:type=Server,*` Using Jython: `AdminControl.queryNames ('WebSphere:type=Server,*')` Example output: `WebSphere:cell=Base ApplicationServerCell, name=server1,mbeanIdentifier= server1,type=Server,node= mynode,process=server1`

queryNames _jmx	Returns a set of ObjectName objects that are based on the ObjectName object and the QueryExp query that you provide.	• Parameters: name-javax.management. ObjectName;query-javax .management.QueryExp • Returns: java.util.Set	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName type=Server,*] set objName [$AdminControl makeObjectName $objNameString] set null [java::null] $AdminControl queryNames _jmx $objName $null` Using Jython: `objectNameString = AdminControl.completeObject Name('type=Server,*') objName = AdminControl.make ObjectName(objectNameString) AdminControl.queryNames_ jmx(objName, None)` Example output: `[WebSphere:cell=Base ApplicationServerCell, name=server1,mbeanIdentifier= server1,type=Server,node= mynode,process=server1]`
reconnect	Reconnects to the server, and clears information out of the local cache.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminControl reconnect` Using Jython: `AdminControl.reconnect()` Example output: `WASX7074I: Reconnect of SOAP connector to host myhost completed.`
setAttribute	Sets the attribute value for the name that you provide.	• Parameters: name-java.lang.String; attributeName-java.lang.String; attributeValue-java.lang.String • Returns: None	Example usage: 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,*') AdminControl.setAttribute (objNameString, 'trace Specification', 'com.ibm. *=all=disabled')`

setAttribute _jmx	Sets the attribute value for the name that you provide.	• Parameters: name-ObjectName; attribute-javax. management. Attribute • Returns: None	Example usage: Using Jacl: `set objectNameString [$AdminControl complete ObjectName WebSphere: type=TraceService,*] set objName [$AdminControl makeObjectName $objectNameString] set attr [java::new javax. management.Attribute traceSpecification com.ibm. *=all=disabled] $AdminControl setAttribute_ jmx $objName $attr` Using Jython: `objectNameString = AdminControl. completeObjectName('WebSphere: type=TraceService,*') import javax.management as mgmt objName = AdminControl. makeObjectName(objectNameString) attr = mgmt.Attribute ('traceSpecification', 'com.ibm.*=all=disabled') AdminControl.setAttribute_ jmx(objName, attr)`
setAttributes	Sets the attribute values for the names that you provide and returns a list of successfully set names.	• Parameters using Jacl: name-String; attributes-java.lang.String • Parameters using Jython: name-String; attributes-java.lang.String or name-String; attributes-java.lang.Object[] • Returns: java.lang.String	Example usage: Using Jacl: `set objNameString [$AdminControl completeObjectName WebSphere: type=TracesService,*] $AdminControl setAttributes $objNameString {{trace Specification com.ibm.ws. *=all=enabled}}` Using Jython with string attributes: `objNameString = AdminControl. completeObjectName('WebSphere: type=TracesService,*') AdminControl.setAttributes (objNameString, '[[trace Specification "com.ibm.ws. *=all=enabled"]]')` Using Jython with object attributes: `objNameString = AdminControl. completeObjectName('WebSphere: type=TracesService,*') 473 AdminControl.setAttributes (objNameString, [['trace Specification', 'com.ibm.ws. *=all=enabled']])`

setAttributes _jmx	Sets the attribute values for the names that you provide and returns a list of successfully set names.	• Parameters: name-ObjectName; attributes-javax.management. AttributeList • Returns: javax.management. AttributeList	Example usage: Using Jacl: `set objectNameString [$AdminControl completeObject Name WebSphere:type= TraceService,*] set objName [$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 $alist` Using Jython: `objectNameString = AdminControl.completeObject Name('WebSphere:type= TraceService,*') import javax.management as mgmt objName = AdminControl. makeObjectName(object NameString) attr = mgmt.Attribute ('traceSpecification', 'com.ibm.ws.*=all=enabled') alist = mgmt.AttributeList() alist.add(attr) AdminControl.setAttributes_ jmx(objName, alist)`
startServer	Starts the specified application server by locating it in the configuration. This command uses the default wait time. You can only use this command if the scripting client is connected to a node agent. This command returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1` Using Jython: `AdminControl.startServer ('server1')`

startServer	Starts the specified application server by locating it in the configuration. The start process waits the number of seconds specified by the wait time for the server to start. You can only use this command if the scripting client is connected to a node agent. This command returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String, wait time-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1 100` Using Jython: `AdminControl.startServer ('server1', 100)`
startServer	Starts the specified application server by locating it in the configuration. This command uses the default wait time. You can use this command when the scripting client is either connected to a node agent or to a deployment manager process. It returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1 myNode` Using Jython: `AdminControl.startServer ('server1', 'myNode')`

startServer	Starts the specified application server by locating it in the configuration. The start process waits the number of seconds specified by the wait time for the server to start. You can use this command when the scripting client is either connected to a node agent or to a deployment manager process. This command returns a message to indicate if the server starts successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String, wait time-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl startServer server1 myNode 100` Using Jython: `AdminControl.startServer ('server1', 'myNode', 100)`
stopServer	Stops the specified application server. The command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1` Using Jython: `AdminControl.stopServer ('server1')`
stopServer	Stops the specified application server. If you set the flag to `immediate`, the server stops immediately. Otherwise, a normal stop occurs. This command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String, immediate flag-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1 immediate` Using Jython: `AdminControl.stopServer ('server1', 'immediate')`

stopServer	Stops the specified application server. This command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1 myNode` Using Jython: `AdminControl.stopServer ('server1', 'my Node')`
stopServer	Stops the specified application server. If you set the flag to `immediate`, the server stops immediately. Otherwise, a normal stop occurs. This command returns a message to indicate if the server stops successfully.	• Parameters: server name-java.lang.String, node name-java.lang.String, immediate flag-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `$AdminControl stopServer server1 myNode immediate` Using Jython: `AdminControl.stopServer ('server1', 'my Node', 'immediate')`

test Connection	A convenience command communicates with the DataSource CfgHelper MBean to test a DataSource connection. This command works with the DataSource that resides in the configuration repository. If the DataSource to be tested is in the temporary workspace that holds the update to the repository, you have to save the update to the configuration repository before running this command. Use this command with the configuration ID that corresponds to the DataSource and the WAS40DataSource object types.	• Parameters: configId-java.lang.String • Returns: java.lang.String	Example usage: 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]` `AdminControl.testConnection(ds)` Example output: `WASX7217I: Connection to provided datasource was successful.`
	The return value is a message that contains the message indicating a successful connection or a connection with warning. If the connection fails, an exception is created from the server indicating the error.		

test Connection	Deprecated. This command can give false results and does not work when connected to a node agent. As of V5.0.2, the preferred way to test a data source connection is with the **test Connection** command that passes in the DataSource configId parameter as the only parameter.	• Parameters: configId-java.lang.String; props-java.lang.String • Returns: java.lang.String	Example usage: Using Jacl: `set ds [lindex [$AdminConfig list DataSource] 0]` `$AdminControl testConnection $ds {{prop1 val1}}` Using Jython: `# get line separator` `import java.lang.System as sys` `lineSeparator = sys.getProperty ('line.separator')` `ds = AdminConfig.list ('DataSource').split (lineSeparator)[0]` `AdminControl.testConnection(ds, '[[prop1 val1]]')` Example output: `WASX7390E: Operation not supported - testConnection command with config id and properties arguments is not supported. Use testConnection command with config id argument only.`
trace	Sets the trace specification for the scripting process to the value that you specify.	• Parameters: traceSpec-java.lang.String • Returns: None	Example usage: Using Jacl: `$AdminControl trace com.ibm.ws.scripting.*=all=enabled` Using Jython: `AdminControl.trace('com.ibm.ws.scripting.*=all=enabled')`

Commands for the AdminApp object

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. In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you will not be able to update the configuration because the configuration for these server processes are copies of the master configuration which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by

connecting a scripting client to a deployment manager. For this reason, to change a configuration, do not run a scripting client in local mode on a node machine. It is not a supported configuration.

The following commands are available for the AdminApp object:

Command name:	Description:	Parameters and return values:	Examples:
deleteUserAndGroupEntries	Deletes users or groups for all roles, and deletes user IDs and passwords for all of the RunAs roles that are defined in the application.	• Parameters: appname • Returns: None	Example usage: Using Jacl: `$AdminApp deleteUserAndGroupEntries myapp` Using Jython: `AdminApp.deleteUserAndGroupEntries('myapp')`
edit	Edits an application or module in non-interactive mode. The **edit** command changes the application deployment. Specify these changes in the options parameter. No options are required for the **edit** command.	• Parameters using Jacl: appname - string; options - string • Parameters using Jython: appname - string; options - string or appname - string; options - Jython list • Returns: string	Example usage: Using Jacl: `$AdminApp edit "JavaMail Sample" {-MapWebModToVH {{"JavaMail Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH}}}` Using Jython with string options: `AdminApp.edit("JavaMail Sample", '[-MapWebModToVH [["JavaMail 32 Sample WebApp" mtcomps.war,WEB-INF/web.xml newVH]]]')` Using Jython with list options: `option = [["JavaMail 32 Sample WebApp", "mtcomps.war,WEB-INF/web.xml", "newVH"]]` `mapVHOption = ["-MapWebModToVH", option]` `AdminApp.edit("JavaMail Sample", mapVHOption)`
editInteractive	Edits 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.	• Parameters using Jacl: appname - string; options - string • Parameters using Jython: appname - string; options - string or appname - string; options - Jython list • Returns: string	Example usage: Using Jacl: `$AdminApp editInteractive ivtApp` Using Jython: `AdminApp.editInteractive('ivtApp')`

export	Exports the application appname parameter to a file that you specify by file name.	• Parameters: appname, filename • Returns: None	Example usage: Using Jacl: `$AdminApp export "My App" /usr/me/myapp.ear` Using Jython: `AdminApp.export("My App", '/usr/me/myapp.ear')`
exportDDL	Extracts the data definition language (DDL) from the application appname parameter to the directoryname parameter that a directory specifies. The options parameter is optional.	• Parameters: appname, directoryname, options • Returns: None	Example usage: Using Jacl: `$AdminApp exportDDL "My App" /usr/me/DDL {-ddlprefix myApp}` Using Jython: `AdminApp.exportDDL("My App", '/usr/me/DDL', '[-ddlprefix myApp]')`

help	Displays general help for the AdminApp object.	• Parameters: None • Returns: None	Example usage: Using Jacl: `$AdminApp help` Using Jython: `print AdminApp.help()` Example output: `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`

help	Displays help for an AdminApp command or installation option.	• Parameters: operation name • Returns: none	Example usage: Using Jacl: `$AdminApp help uninstall` Using Jython: `print AdminApp.help('uninstall')` Example output: `WASX7102I: Method: uninstall` `Arguments: application name, options` `Description: Uninstalls application named by "application name" using the options supplied by String 2.` `Method: uninstall` `Arguments: application name` `Description: Uninstalls the application specified by "application name" using default options.`
install	Installs an application in non-interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.	• Parameters using Jacl: earfile- string; options- string • Parameters using Jython: earfile- string; options- string or earfile- string; options- Jython list • Returns: None	Example usage: Using Jacl: `$AdminApp install c:/apps/myapp.ear` Using Jython: `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: `$AdminApp options myApp.ear` Using Jython: `AdminApp.options('myApp.ear')` You can also obtain help for each object with the following command: Using Jacl: `$AdminApp help MapModulesToServers` Using Jython: `AdminApp.help('MapModulesToServers')`
installInteractive	Installs an application in interactive mode, given a fully qualified file name and a string of installation options. The options parameter is optional.	• Parameters using Jacl: earfile- string; options- string • Parameters using Jython: earfile- string; options- string or earfile- string; options- Jython list • Returns: None	Example usage: Using Jacl: `$AdminApp installInteractive c:/websphere/appserver/installableApps/jmsample.ear` Using Jython: `AdminApp.installInteractive('c:/websphere/appserver/installableApps/jmsample.ear')`

isAppReady	Tests to see 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.	• Parameters: application name • Returns: true or false	Example usage: Using Jacl: `$AdminApp isAppReady DefaultApplication` Using Jython: `AdminApp.isAppReady('DefaultApplication')` Example output: `ADMA5071I: Distribution status check started for a pplication DefaultApplication.` `WebSphere:cell=Node03Cell,node=myNode,distribution =true` `ADMA5011I: The cleanup of the temp directory for a pplication DefaultApplication is complete.` `ADMA5072I: Distribution status check completed for application DefaultApplication.` `true`
isAppReady	Tests to see if the specified application has been distributed and is ready to be run. Valid values for the ignoreUnknownState parameter include `true` and `false`. If you specify a value of `true`, nodes and servers with an unknown state will not be included in the final ready return. The command returns a value of `true` if the application is ready or a value of `false` if the application is not ready. This command is not supported when the wsadmin tool is not connected to a server.	• Parameters: application name, ignoreUnknownState • Returns: true or false	Example usage: Using Jacl: `$AdminApp isAppReady TEST true` Using Jython: `AdminApp.isAppReady('TEST', 'true')` Example output: `ADMA5071I: Distribution status check started for a pplication TEST.` `WebSphere:cell=myCell,node=myNode,distribution=unknown` `ADMA5011I: The cleanup of the temp directory for a pplication TEST is complete.` `ADMA5072I: Distribution status check completed for application TEST.` `false`

list	Lists the applications that are installed in the configuration.	• Parameters: None • Returns: application names	Example usage: Using Jacl: `$AdminApp list` Using Jython: `print AdminApp.list()` Example output: `adminconsole` `DefaultApplication` `ivtApp`
list	Lists the applications that are installed on a given target scope in the configuration.	• Parameters: target • Returns: Application names	Example usage: Using Jacl: `$AdminApp list WebSphere:cell=myCell,node=myNode, server=myServer` Using Jython: `print AdminApp.list("WebSphere:cell=myCell, node=myNode,server=myServer")` Example output: `DefaultApplication` `PlantsByWebSphere` `SamplesGallery` `ivtApp` `query`
listModules	Lists the modules in an application. The options parameter is optional. The valid option is -server. This option lists the application servers on which the modules are installed.	• Parameters: appname, options • Returns: modules in the application	Example usage: Using Jacl: `$AdminApp listModules ivtApp` Using Jython: `print AdminApp.listModules('ivtApp')` Example output: `ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml` `ivtApp#ivt_app.war+WEB-INF/web.xml` This example is formed by the concatenation of appname, #, module URI, +, and DD URI. You can pass this string to the **edit** and **editInteractive** AdminApp commands.

options	Displays a list of options for installing an Enterprise Archive (EAR) file.	• Parameters: earfile • Returns: Information about the valid installation options for an Enterprise Archive (EAR) file.	Example usage: Using Jacl: `$AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear` Using Jython: `AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear')` Example usage: `WASX7112I: The following options are valid for "c:/websphere/appserver/installableapps/ivtApp.ear"` `MapRolesToUsers` `BindJndiForEJBNonMessageBinding` `MapEJBRefToEJB` `MapWebModToVH` `MapModulesToServers` `EnsureMethodProtectionFor10EJB` `GetServerName` `preCompileJSPs` `nopreCompileJSPs` `distributeApp` `nodistributeApp` `useMetaDataFromBinary` `nouseMetaDataFromBinary` `deployejb` `nodeployejb` `createMBeansForResources` `nocreateMBeansForResources` `reloadEnabled` `noreloadEnabled` `deployws` `nodeployws` `usedefaultbindings` `defaultbinding.force` `allowPermInFilterPolicy` `noallowPermInFilterPolicy` `verbose` `update` `update.ignore.old` `update.ignore.new` `installed.ear.destination` `appname` `reloadInterval` `validateinstall` `deployejb.rmic` `deployejb.dbtype` `deployejb.dbschema` `deployejb.classpath` `deployws.classpath` `deployws.jardirs` `defaultbinding.datasource.jndi` `defaultbinding.datasource.username` `defaultbinding.datasource.password` `defaultbinding.cf.jndi` `defaultbinding.cf.resauth` `defaultbinding.ejbjndi.prefix` `defaultbinding.virtual.host` `defaultbinding.strategy.file` `server` `node` `cell` `cluster` `contextroot` `custom`

options	Displays a list of options for editing an existing application.	• Parameters: Application name • Returns: Information about the valid edit options for an application.	Example usage: Using Jacl: `$AdminApp options ivtApp` Using Jython: `AdminApp.options('ivtApp')` Example output: `WASX7112I: The following options are valid for "ivtApp"` `MapRolesToUsers` `BindJndiForEJBNonMessageBinding` `MapEJBRefToEJB` `MapWebModToVH` `MapModulesToServers` `distributeApp` `nodistributeApp` `useMetaDataFromBinary` `nouseMetaDataFromBinary` `createMBeansForResources` `nocreateMBeansForResources` `reloadEnabled` `noreloadEnabled` `verbose` `installed.ear.destination` `reloadInterval`
options	Displays a list of options for editing a module in an existing application.	• Parameters: application module name. This parameter requires the same module name format as the output that is returned by the **listModules** command. • Returns: Information about the valid edit options for a module.	Example usage: Using Jacl: `$AdminApp options ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml` Using Jython: `AdminApp.options('ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml')` Example output: `WASX7112I: The following options are valid for "ivtApp#ivtEJB.jar+META-INF/ejb-jar.xml"` `MapRolesToUsers` `BindJndiForEJBNonMessageBinding` `MapModulesToServers` `verbose`

options	Displays a list of options for installing or updating an application or application module file.	• Parameters: file, operation - The following list includes the valid values: – installapp - Installing the file that is specified – updateapp - Updating an existing application with the file that is specified – addmodule - Adding the module file that is specified to an existing application – updatemodule - Updating an existing module in an application with the module file that is specified • Returns: Information about the valid options that are available for the operation that is requested with the input file.	Example using the updateapp operation: Using Jacl: `$AdminApp options c:/websphere/appserver/installableApps/ivtApp.ear updateapp` Using Jython: `AdminApp.options('c:/websphere/appserver/installableApps/ivtApp.ear', 'updateapp')` Example using the addmodule operation: Using Jacl: `$AdminApp options myModule.jar addmodule` Using Jython: `AdminApp.options('DefaultWebApplication.war', 'addmodule')`

			Example output using the updateapp operation: WASX7112I: The following options are valid for "c:/websphere/appserver/installableApps/ivtApp.ear" MapRolesToUsers BindJndiForEJBNonMessageBinding MapEJBRefToEJB MapWebModToVH MapModulesToServers EnsureMethodProtectionFor10EJB GetServerName preCompileJSPs nopreCompileJSPs distributeApp nodistributeApp useMetaDataFromBinary nouseMetaDataFromBinary deployejb nodeployejb createMBeansForResources nocreateMBeansForResources reloadEnabled noreloadEnabled deployws nodeployws usedefaultbindings defaultbinding.force allowPermInFilterPolicy noallowPermInFilterPolicy verbose update update.ignore.old update.ignore.new installed.ear.destination reloadInterval deployejb.rmic deployejb.dbtype deployejb.dbschema deployejb.classpath deployws.classpath deployws.jardirs defaultbinding.datasource.jndi defaultbinding.datasource.username defaultbinding.datasource.password defaultbinding.cf.jndi defaultbinding.cf.resauth defaultbinding.ejbjndi.prefix defaultbinding.virtual.host defaultbinding.strategy.file appname contextroot custom contenturi contents operation

			Example output using the addmodule operation: `WASX7112I: The following options are valid for "DefaultWebApplication.war"` `MapRolesToUsers` `MapEJBRefToEJB` `MapWebModToVH` `MapModulesToServers` `GetServerName` `preCompileJSPs` `nopreCompileJSPs` `deployejb` `nodeployejb` `deployws` `nodeployws` `usedefaultbindings` `defaultbinding.force` `verbose` `defaultbinding.datasource.jndi` `defaultbinding.datasource.username` `defaultbinding.datasource.password` `defaultbinding.cf.jndi` `defaultbinding.cf.resauth` `defaultbinding.ejbjndi.prefix` `defaultbinding.virtual.host` `defaultbinding.strategy.file` `server` `node` `cell` `cluster` `contextroot` `custom` `contenturi` `contents` `operation`
publishWSDL	Publishes Web Services Description Language (WSDL) files for the application that is specified in the appname parameter to the file that is specified in the filename parameter.	• Parameters: appname, filename • Returns: None	Example usage: Using Jacl: `$AdminApp publishWSDL JAXRPCHandlerServer c:/temp/a.zip` Using Jython: `AdminApp.publishWSDL('JAXRPCHandlerServer', 'c:/temp/a.zip')`

publishWSDL	Publishes Web Services Description Language (WSDL) files for the application that is specified in the appname parameter to the file that is specified in the filename parameter using the SOAP address prefixes that are specified in the soapAddressPrefixes parameter.	• Parameters: appname, filename, soapAddressPrefixes • Returns: None	Example usage: Using Jacl: `$AdminApp publishWSDL JAXRPCHandlersServer c:/temp/a.zip {{JAXRPCHandlersServerApp.war {{http http://localhost:9080}}}}` Using Jython: `AdminApp.publishWSDL('JAXRPCHandlersServer', 'c:/temp/a.zip', '[[JAXRPCHandlersServerApp.war [[http http://localhost:9080]]]]')`
searchJNDI References	Lists applications that refer to the Java Naming and Directory Interface (JNDI) name on a specific node.	• Parameters: Node configuration ID, options • Returns: string	Example usage: The following example assumes that an installed application named `MyApp` has a JNDI name of `eis/J2CCF1`. Using Jacl: `$AdminApp searchJNDIReferences $node {-JNDIName eis/J2CCF1 -verbose}` Using Jython: `print AdminApp.searchJNDIReferences(node, '[-JNDIName eis/J2CCF1 -verbose]')` Example 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]`

taskInfo	Provides information about a particular task option for an application file. Many task names have changed between V5.x and V6.x for similar or the exact same operation. You may need to update existing scripts if you are migrating from V5.x to V6.x.	• Parameters: earfile, task name • Returns: None	Example usage: Using Jacl: `$AdminApp taskInfo c:/websphere/appserver/installableApps/jmsample.ear MapWebModToVH` Using Jython: `print AdminApp.taskInfo('c:/websphere/appserver/installableApps/jmsample.ear', 'MapWebModToVH')` Example output: `MapWebModToVH: Selecting virtual hosts for Web modules Specify the virtual host where you want to install the Web modules that are contained in your application. Web modules can be installed on the same virtual host or dispersed among several hosts. Each element of the MapWebModToVH task consists of the following three fields: "webModule," "uri," "virtualHost." Of these fields, the following fields might be assigned new values: "virtualHost"and the following are required: "virtualHost" The current contents of the task after running default bindings are: webModule: JavaMail Sample WebApp uri: mtcomps.war,WEB-INF/web.xml virtualHost: default_host`
uninstall	Uninstalls an existing application.	• Parameters: appname- string • Returns: None	Example usage: Using Jacl: `$AdminApp uninstall myApp` Using Jython: `AdminApp.uninstall('myApp')` Example 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.`

updateAccessIDs	Updates the access ID information for users and groups that are assigned to various roles that are defined in the application. The access IDs are read from the user registry and saved in the application bindings. This operation improves run-time performance of the application. Call 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 is set to `NONE`. You must be connected to a server to invoke this command. The bALL Boolean parameter retrieves and saves all access IDs for users and groups in the application bindings. Specify `false` if you want to retrieve access IDs for users or groups that do not have an access ID in the application bindings.	• Parameters: appname, bALL • Returns: None	Example usage: Using Jacl: `$AdminApp updateAccessIDs myapp true` Using Jython: `AdminApp.updateAccessIDs('myapp', 'true')`

view	View the task that is specified by the taskname option parameter for the application or by the module that is specified by the 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.	• Parameters: name, taskname option • Returns: string	Example usage: Using Jacl: `$AdminApp view adminconsole {-tasknames}` Using Jython: `AdminApp.view('adminconsole', ['-tasknames'])` Example output: `MapModulesToServers` `MapWebModToVH` `MapRolesToUsers` Using Jacl: `$AdminApp view adminconsole {-MapModulesToServers}` Using Jython: `AdminApp.view('adminconsole', ['-MapModulesToServers'])` Example output: `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` Example usage: Using Jacl: `$AdminApp view adminconsole#adminconsole.war+WEB-INF/web.xml {-MapRolesToUsers}`

			Using Jython: `AdminApp.view('adminconsole#adminconsole.war+WEB-INF/web.xml', ['-MapRolesToUsers'])` Example output: `MapRolesToUsers: Mapping Users to Roles` `Each role that is defined in the application or the module must be mapped to a user or a group from the user registry of the domain:` `Role: administrator` `Everyone?: No` `All Authenticated?: No` `Mapped Users:` `Mapped Groups:` `Role: operator` `Everyone?: No` `All Authenticated?: No` `Mapped Users:` `Mapped Groups:` `Role: configurator` `Everyone?: No` `All Authenticated?: No` `Mapped Users:` `Mapped Groups:` `Role: monitor` `Everyone?: No` `All Authenticated?: No` `Mapped Users:` `Mapped Groups:`

update	Updates an application in non-interactive mode. Provide the application name, content type, and update options.	• Parameters using Jacl: appname, content type, options – string format • Parameters using Jython: appname, content type, option- string or list format • Returns: String This command supports the addition, removal, and update of application subcomponents or the entire application. 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:	Example usage: Using Jacl: `$AdminApp update myApp file {-operation add -contents /apps/myApp/web.xml -contenturi META-INF/web.xml}` Using Jython with string options: `AdminApp.update('myApp', 'file', '[-operation add -contents /apps/myApp/web.xml -contenturi META-INF/web.xml]')` Using Jython with list options: `AdminApp.update('myApp', 'file', ['-operation', 'add', '-contents', '/apps/myApp/web.xml', '-contenturi', 'META-INF/web.xml'])` Example 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.`

		• – `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.	

		• – `partialapp` - Indicates that you want to update a partial application. Using a subset of application components provided in a zip file format you can update, add, and delete files and modules. The zip 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 zip 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 zip file. When a partial application is provided as an update input, binding information and application options cannot be specified and are ignored, if provided. For a list of the valid options for the **update** command, see "Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands" on page 769.	

updateInteractive	Updates an application in interactive mode. Provide the application name, content type, and update options.	• Parameters using Jacl: appname, content type, options - string format • Parameters using Jython: appname, content type, option - string or list format • Returns: String 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. Use the content type parameter to indicate if you want to update part of the application or the entire application. The following list contains the valid content type values for the **updateInteractive** command:	Example usage: Using Jacl: `$AdminApp updateInteractive myApp modulefile {-operation add -contents /apps/myApp/Increment.jar -contenturi Increment.jar -nodeployejb -BindJndiForEJBNonMessageBinding {{"Increment Enterprise JavaBeans" Increment Increment.jar,META-INF/ejb-jar.xml Inc}}}`

		• – `app` - Indicates that you want to update the entire application. This option is the same as indicating the update option with **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.	Using Jython string: `AdminApp.updateInteractive ('myApp', 'modulefile', '[-operation add -contents /apps/myApp/Increment.jar -contenturi Increment. jar -nodeployejb -BindJndiForEJBNonMessageBinding [["Increment Enterprise JavaBeans" Increment Increment. jar,META-INF/ejb-jar.xml Inc]]]')` Using Jython list: `bindJndiForEJBValue = [["Increment Enterprise Java Beans", "Increment", "Increment.jar,META-INF/ejb-jar.xml", "Inc"]]` `AdminApp.updateInteractive('myApp', 'modulefile', ['-operation', 'add', '-contents', '/apps/myApp/Increment.jar', '-contenturi', 'Increment.jar', '-nodeployejb', '-BindJndiForEJBNonMessageBinding', bindJndiForEJBValue])`

		• – `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, additional 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 the 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.	Example 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 Enterprise Java Bean` `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 Enterprise Java Bean` `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 Enterprise Java Bean` `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 Enterprise Java Bean` `URI:  Increment.jar,META-INF/ejb-jar.xml` `Server:` `[WebSphere:cell=myCell,node=myNode, server=server1]:`

		• – `partialapp` - Indicates that you want to update a partial application. Using subset of application components provided in a zip file format you can update, add, and delete files and modules. The zip file is not a valid Java 2 platform, Enterprise Edition (J2EE) archive. Instead, this file contains application artifacts in the same hierarchical structure as they are displayed in an EAR file.	

		For more information on how to construct the partial application zip 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 zip file. When a partial application is provided as an update input, the binding information and application options cannot be specified and are ignored, if provided. • For a list of the valid options for the **updateInteractive** command, see "Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands" on page 769.	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 Enterprise Java Bean 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 Enterprise Java Bean 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.

Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands

This article lists the available options for the **install**, **installInteractive**, **edit**, **editInteractive**, **update**, and **updateInteractive** commands of the AdminApp object.

The options listed in this article apply to all of these commands except where noted.

See Commands for the AdminApp object for more detailed information on how to use the commands. See Usage table for the options of the AdminApp object install, installInteractive, update, updateInteractive, edit, and editInteractive commands for a list of applicable commands for each option.

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. See “Pattern matching with the wsadmin tool” on page 833 for more information.

The following options are available for the **install**, **installInteractive**, **edit**, **editInteractive**, **update**, and **updateInteractive** commands:

Option name:	Description:	Examples:

ActSpecJNDI	Binds J2C activation specifications to destination JNDI names. You can bind J2C activation specifications in your application or module to a destination JNDI name. This option is optional. Each element of the ActSpecJNDI option consists of the following fields: RARModule, uri, j2cid, j2c.jndiName. j2c.jndiName field, can be assigned a value. The current 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 You can only use this option if the activation specification has the `Destination` property defined in the ra.xml file and the introspected type of the `Destination` property is the following: `javax.jms.Destination`	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']]])`

	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.	
allowDispatch Remote Include	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.	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 $attrs` Using Jython: `deployments = AdminConfig.getid ('/Deployment:RRDEnabledAppname/')` `deploymentObject = AdminConfig.showAttribute (deployments, 'deployedObject')` `rrdAttr = ['allowDispatchRemoteInclude', 'true']` `attrs = [rrdLocalAttr]` `AdminConfig.modify (deploymentObject, attrs)`
allowPermIn Filter Policy	Specifies to continue with the application deployment process even when the application contains policy permissions that are in the filter policy. This option does not require a value.	
allowService Remote Include	Enables an enterprise application to service an include request from an enterprise application with the option set to `true`.	Using Jacl: `set deployments [$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	Specifies the name of the application. The default is the display name of the application.	
BackendIdSelection	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.	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']]])`

BindJndiForEJB Message Binding	Binds enterprise beans to listener port names or Java Naming and Directory Interface (JNDI) names. Use this option to provide missing data or update a task. Ensure each message-driven enterprise bean in your application or module is bound to a listener port name. 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]:	Using Jacl: `$AdminApp install $ear {-BindJndiForEJBMessageBinding {{Ejb1 MessageBean ejb-jar-ic.jar,META-INF/ejb-jar.xml myListenerPort jndi1 jndiDest1 act$pecAuth1}}}` 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']]])`

	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 require an update.	
BindJndiForEJB NonMessage Binding	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. 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.	Using Jacl: `$AdminApp install myapp.ear {-BindJndiForEJBNonMessageBinding {{"Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-BindJndiForEJBNonMessageBinding {{.* .* Increment.jar,.* IncBean}}}` Using Jython: `AdminApp.install('myapp.ear', '[-BindJndiForEJBNonMessageBinding [["Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean]]]')` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-BindJndiForEJBNonMessageBinding', [['.*', '.*', 'Increment.jar,.*', 'IncBean']]])`

buildVersion	Displays the build version of an application EAR file. You cannot modify this option because it is read-only. This options returns the build version information for an application EAR if you have specified the build version in the `MANIFEST.MF` application EAR file.	
cell	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.	

cluster	Specifies the cluster name to install or update an entire application or to update an application in order to add a new module. This option only applies in a Network Deployment environment. 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. You cannot use the -cluster and -server options together. If you want to deploy an application and specify the HTTP server during the deployment so that the application will appear in the generated plugin-cfg.xml file, you must first install the application with a target of -cluster. After you install the application and before you save, use **edit** command of the AdminApp object to add the additional mapping to the Web server.	

contents	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	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	Specifies the context root that you use when installing a stand-alone Web archive (WAR) file.	

CorrectOracle Isolation Level	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 require an update.	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]]])`

CorrectUse System Identity	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 require an update.	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']]])`
	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	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>	Using Jacl: `$AdminApp install myapp.ear {-appname MyApp -CtxRootForWebMod {{"IVT Application" ivt_app.war,web.xml /mycontextroot}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-appname MyApp -CtxRootForWebMod {{.* .* /mycontextroot}}}` Using Jython: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-CtxRootForWebMod', [["IVT Application", 'ivt_app.war,web.xml', '/mycontextroot']]])`
custom	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.	

DataSourceFor 10CMPBeans	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.	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']]])`

	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 require an update.	

DataSourceFor20CMPBeans	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]:	Using Jacl: `$AdminApp install myapp.ear {-DataSourceFor20CMPBeans {{"Increment Enterprise Java Bean" Increment Increment.jar, META-INF/ejb-jar.xml jndi1 container}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-DataSourceFor20CMPBeans {{.* .* Increment.jar,.* jndi1 container}}}` Using Jython: `AdminApp.install('myapp.ear', ['-DataSourceFor20CMPBeans', [["Increment Enterprise Java Bean", 'Increment', 'Increment.jar, META-INF/ejb-jar.xml', 'jndi1', 'container']]])` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-DataSourceFor20CMPBeans', [['.*', '.*', 'Increment.jar,.*', 'jndi1', 'container']]])`

	• 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 require an update.	

DataSourceFor10 EJBModules	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 DataSourceFor 10EJBModules 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]:	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']]])`

	• 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 require an update.	

DataSourceFor20EJBModules	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, and auth.props. 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]:	Using Jacl: `$AdminApp install myapp.ear {-DataSourceFor20EJBModules {{"Increment Enterprise Java Bean" Increment.jar,META-INF/ejb-jar.xml jndi2 container}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-DataSourceFor20EJBModules {{.* Increment.jar,.* jndi2 container}}}` Using Jython: `AdminApp.install('myapp.ear', ['-DataSourceFor20EJBModules', [["Increment Enterprise Java Bean", 'Increment.jar,META-INF/ejb-jar.xml', 'jndi2', 'container']]])` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-DataSourceFor20EJBModules', [['.*', 'Increment.jar,.*', 'jndi2', 'container']]])`

	• 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. 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.	
`defaultbinding.cf.jndi`	Specifies the Java Naming and Directory Interface (JNDI) name for the default connection factory.	
`defaultbinding.cf.resauth`	Specifies the RESAUTH for the connection factory.	
`defaultbinding.datasource.jndi`	Specifies the Java Naming and Directory Interface (JNDI) name for the default data source.	

defaultbinding. datasource. password	Specifies the password for the default data source.	
	Specifies the user name for the default data source.	
defaultbinding. ejbjndi. prefix	Specifies the prefix for the enterprise bean Java Naming and Directory Interface (JNDI) name. To use the defaultbinding.ejbjndi.prefix option on the z/OS platform, you must also specify the usedefaultbindings option. If you do not specify the use usedefaultbindings option, then the default is the nousedefaultbindings option.	
defaultbinding.force	Specifies that the default bindings override the current bindings.	
defaultbinding. strategy. file	Specifies a custom default bindings strategy file.	
defaultbinding. virtual. host	Specifies the default name for a virtual host.	
depl.extension.reg	Deprecated. No replication option is available.	

deployejb	Specifies to run the EJBDeploy tool during installation. This option does not require a value. 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`.	
deployejb.classpath	Specifies an extra class path for the EJBDeploy tool.	
`hdeployejb.dbschema`	Specifies the database schema for the EJBDeploy tool.	
deployejb.dbtype	Specifies the database type for the EJBDeploy tool. Possible values include: `CLOUDSCAPE_V5` `DB2UDB_V72` `DB2UDBOS390_V6` `DB2UDBISERIES` `INFORMIX_V73` `INFORMIX_V93` `MSSQLSERVER_V7` `MSSQLSERVER_2000` `ORACLE_V8` `ORACLE_V9I` `SYBASE_V1200` For a list of current supported database vendor types, run `ejbdeploy -?`.	
deployejb.rmic	Specifies extra RMIC options to use for the EJBDeploy tool.	

deployws	Specifies to deploy Web services during installation. This option does not require a value. The default value is: `nodeployws`.	
deployws.classpath	Specifies the extra class path to use when you deploy Web services.	
deployws.jardirs	Specifies the extra extension directories to use when you deploy Web services.	
distributeApp	Specifies that the application management component distributes application binaries. This option does not require a value. This setting is the default.	

EmbeddedRar	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`	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']]])`

	Where j2cid is: `J2C connection factory : connection Factory Interface` `J2C admin object : adminObject Interface` `J2C activation specification : message listener type` If the ID is not unique in the ra.xml file, -<number> will be added. For example, javax.sql.DataSource-2.	
	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.	

EnsureMethod Protection For10EJB	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 require an update.	Using Jacl: `$AdminApp install myapp.ear {-EnsureMethodProtectionFor10EJB {{"Increment EJB Module" IncrementEJBBean.jar,META-INF/ejb-jar.xml ""} {"Timeout EJB Module" TimeoutEJBBean.jar,META-INF/ejb-jar.xml methodProtection.denyAllPermission}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-EnsureMethodProtectionFor10EJB {{.* IncrementEJBBean.jar,.* ""} {.* TimeoutEJBBean.jar,.* methodProtection.denyAllPermission}}}` Using Jython: `AdminApp.install('myapp.ear', '[-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.

EnsureMethod Protection For20EJB	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.	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	This 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`	Using Jacl: `$AdminApp install /ASV/o0528.02/WebSphere/AppServer/binaries/ DefaultApplication.ear {-appname MyApp -cell GooddogNode04Cell -node GooddogNode -server server1 -filepermission .*\\.jsp=777#.*\\.xml=755}` Using Jython: `AdminApp.install("/ASV/o0528.02/WebSphere/AppServer/binaries/ DefaultApplication.ear", ["-appname", "MyApp", "-cell", "GooddogNode04Cell", "-node", "GooddogNode", "-server", "server1", "-filepermission", ".*\\.jsp=777#.*\\.xml=755"])`
installdir	Deprecated. This option is replaced by the `installed.ear.destination` option.	
`installed.ear. destination`	Specifies the directory to place application binaries.	

JSPCompile Options	Assigns shared libraries to applications 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: • Web module: xxx • URI: xxx • JSP Class Path: <jsp class path> • Use Full Package Names: AppDeploymentOption.Yes | AppDeploymentOption.No • JDK Source Level: xx • disableJspRuntimeCompilation.column: AppDeploymentOption.Yes | AppDeploymentOption.No Specify the options for the JSP precompiler. This option is only valid if you use the preCompileJSPs option also.	Using Jacl: `$AdminApp install myapp.ear {-appname MyApp -preCompileJSPs -JSPCompileOptions {{"IVT Application" ivt_app.war,WEB-INF/web.xml jspcp AppDeploymentOption.Yes 15 AppDeploymentOption.No}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-appname MyApp -preCompileJSPs -JSPCompileOptions {{.* .* jspcp AppDeploymentOption.Yes 15 AppDeploymentOption.No}}}` Using Jython: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-preCompileJSPs', '-JSPCompileOptions', [["IVT Application", 'ivt_app.war,WEB-INF/web.xml', 'jspcp', 'AppDeloymentOption.Yes', 15, 'AppDeploymentOption.No']]])` Using Jython with pattern matching: `AdminApp.install('myapp.ear', '-appname', 'MyApp', '-preCompileJSPs', '-JSPCompileOptions', [['.*', '.*', 'jspcp', 'AppDeploymentOption.Yes', 15, 'AppDeploymentOption.No']]])`

JSPReloadFor WebMod	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: <AppDeploymentOption.Yes | AppDeploymentOption.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 you application.	Using Jacl: `$AdminApp install myapp.ear {-appname MyApp -JspReloadForWebMod {{"IVT Application" ivt_app.war,WEB-INF/ibm-web-ext.xmi AppDeploymentOption.Yes 5}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-appname MyApp -JspReloadForWebMod {{.* .* AppDeploymentOption.Yes 5}}}` Using Jython: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-JspReloadForWebMod', [["IVT Application", 'ivt_app.war,WEB-INF/ibm-web-ext.xmi', 'AppDeploymentOption.Yes', 5]]])` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-JspReloadForWebMod', [['.*', '.*', 'AppDeploymentOption.Yes', 5]]])`

MapEJBRefToEJB	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. 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.	Using Jacl: `$AdminApp install myapp.ear {-MapEJBRefToEJB {{"Examples Application" "" examples.war,WEB-INF/web.xml BeenThereBean com.ibm.websphere. beenthere.BeenThere IncBean}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-MapEJBRefToEJB {{.* .* .* .* .* IncBean}}}` Using Jython: `AdminApp.install('myapp.ear', '[-MapEJBRefToEJB [["Examples Application" "" examples.war,WEB-INF/web.xml BeenThereBean com.ibm.websphere. beenthere.BeenThere IncBean]]]')` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-MapEJBRefToEJB', [['.*', '.*', '.*', '.*', '.*', 'IncBean']]])`

MapEnvEntry ForWebMod	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: String • Description: null • Value: <env-entry value> Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for you application.	Using Jacl: `$AdminApp install myapp.ear {-appname MyApp -MapEnvEntryForWebMod {{"IVT Application" ivt_app.war,WEB-INF/web.xml ivt/ivtEJBObject String null newEnvEntry}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-appname MyApp -MapEnvEntryForWebMod {{.* .* .* .* newEnvEntry}}}` Using Jython: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForWebMod', [["IVT Application", 'ivt_app.war,WEB-INF/web.xml', 'ivt/ivtEJBObject', 'String', 'null', 'newEnvEntry']]])` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapEnvEntryForWebMod', [['.*', '.*', '.*', '.*', 'newEnvEntry']]])`

MapInitParam ForServlet	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> Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for you application.	Using Jacl: `$AdminApp install myapp.ear {-appname MyApp -MapInitParamForServlet {{"IVT Application" ivt_app.war,WEB-INF/web.xml ivtservlet pName1 null MyInitParamValue}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-appname MyApp -MapInitParamForServlet {{.* .* .* .* .* MyInitParamValue}}}` Using Jython: `AdminApp.install('myapp.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']]])`

MapMessage Destination RefToEJB	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: • EJB Module: ejb-jar-ic.jar • EJB: MessageBean • URI: ejb-jar-ic.jar,META-INF/ejb-jar.xml • JNDI Name: [eis/J2CACT1]: • Destination JNDI Name: [jms/TopicName]: • The default JNDI name will be picked up from the corresponding message destination reference.	Using Jacl: `$AdminApp install $earfile {-MapMessageDestinationRefToEJB {{ejb-jar-ic.jar Publisher ejb-jar-ic.jar,META-INF/ejb-jar.xml MyConnection jndi2} {ejb-jar-ic.jar Publisher ejb-jar-ic.jar,META-INF/ejb-jar.xml PhysicalTopic jndi3} {ejb-jar-ic.jar Publisher ejb-jar-ic.jar,META-INF/ejb-jar.xml jms/ABC jndi4}}}`  Using Jacl with pattern matching: `$AdminApp install $earfile {-MapMessageDestinationRefToEJB {{.* .* .* MyConnection jndi2} {.* .* .* PhysicalTopic jndi3} {.* .* .* jms/ABC jndi4}}}`  Using Jython: `AdminApp.install(ear1, ['-MapMessageDestinationRefToEJB', [['ejb-jar-ic.jar', 'Publisher', 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'MyConnection', 'jndi2'], ['ejb-jar-ic.jar', 'Publisher', 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'PhysicalTopic', 'jndi3'], ['ejb-jar-ic.jar', 'Publisher', 'ejb-jar-ic.jar,META-INF/ejb-jar.xml', 'jms/ABC', 'jndi4']]])`  Using Jython with pattern matching: `AdminApp.install(ear1, ['-MapMessageDestinationRefToEJB', [['.*', '.*', '.*', 'MyConnection', 'jndi2'], ['.*', '.*', '.*', 'PhysicalTopic', 'jndi3'], ['.*', '.*', '.*', 'jms/ABC', 'jndi4']]])`

	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.	

MapModulesTo Servers	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 require an update.	Using Jacl: `$AdminApp install /myapp.ear {-MapModulesToServers {{"Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml WebSphere:cell=mycell,node=mynode,server=server1} {"Default Application" default_app.war,WEB-INF/web.xml WebSphere:cell=mycell,node=mynode,server=server1} {"Examples Application" examples.war,WEB-INF/web.xml WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=yournode,server=server1}}}`  Using Jacl with pattern matching: `$AdminApp install myapp.ear {-MapModulesToServers {{.* .*.jar,.* WebSphere:cell=mycell,node=mynode,server=server2} {.* .*.war,.* WebSphere:cell=mycell,node=mynode,server=server1}}}`  The following example, adds *server2* and *server3* to application that is installed: `$AdminApp install /myapp.ear {-MapModulesToServers {{.* .* +WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=mynode,server=server3}} -appname myapp -update -update.ignore.old}`  The following example removes *server1* from application that is installed: `$AdminApp edit myapp {-MapModulesToServers {{.* .* -WebSphere:cell=mycell,node=mynode,server=server1}} -update -update.ignore.old}`  Using Jython: `AdminApp.install('/myapp.ear', '[-MapModulesToServers [["Increment Bean Jar" Increment.jar,META-INF/ejb-jar.xml WebSphere:cell=mycell,node=mynode,server=server1] ["Default Application" default_app.war,WEB-INF/web.xml WebSphere:cell=mycell,node=mynode,server=server1] ["Examples Application" examples.war,WEB-INF/web.xml WebSphere:cell=mycell,node=mynode,server=server2+WebSphere:cell=mycell,node=yournode,server=server1]]]')`  Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-MapModulesToServers', [['.*', '.*.jar,.*', 'WebSphere:cell=mycell,node=mynode,server=server2'] ['.*', '.*.war,.*', 'WebSphere:cell=mycell,node=mynode,server=server1']]])`  The following example, adds *server2* and *server3* to application that is installed: `AdminApp.install('/myapp.ear', ['-MapModulesToServers', [['.*', '.*', '+WebSphere:cell=mycell,node=mynode,server=server2 +WebSphere:cell=mycell,node=mynode,server=server3']], '-appname', 'myapp', '-update', '-update.ignore.old'])`  The following example removes *server1* from application that is installed: `AdminApp.edit('myapp', ['-MapModulesToServers', [['.*', '.*', '-WebSphere:cell=mycell,node=mynode,server=server1']]])`

MapResEnvRef ToRes	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 require an update.	Using Jacl: `$AdminApp install myapp.ear {-MapResEnvRefToRes {{AsyncSender AsyncSender asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-MapResEnvRefToRes {{.* .* .* .* .* jms/Resource2}}}` Using Jython: `AdminApp.install('myapp.ear', '[-MapResEnvRefToRes [[AsyncSender AsyncSender asyncSenderEjb.jar,META-INF/ejb-jar.xml jms/ASYNC_SENDER_QUEUE javax.jms.Queue jms/Resource2]]]')` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-MapResEnvRefToRes', [['.*', '.*', '.*', '.*', '.*', 'jms/Resource2']]])`

MapResRefToEJB	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. Each element of the MapResRefToEJB option consists of the following fields: module, EJB, uri, referenceBinding, resRef.type, JNDI, login.config.name, and auth.props. Of these fields, the following can be assigned values: JNDI, login.config.name, auth.props. The JNDI field is required. The current contents of the option after running default bindings 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]:	Using Jacl: `$AdminApp install myapp.ear {-MapResRefToEJB {{deplmtest.jar MailEJBObject deplmtest.jar,META-INF/ejb-jar.xml mail/MailSession9 javax.mail.Session jndi1 login1 authProps1} {"JavaMail Sample WebApp" "" mtcomps.war,WEB-INF/web.xml mail/MailSession9 javax.mail. Session jndi2 login2 authProps2}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-MapResRefToEJB {{deplmtest.jar .* .* .* .* jndi1 login1 authProps1} {"JavaMail Sample WebApp" .* .* .* .* jndi2 login2 authProps2}}}` 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'], ["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'], ["JavaMail Sample WebApp", '.*', '.*', '.*', '.*', 'jndi2', 'login2', 'authProps2']]])`

	• 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 that is needed for your application. You need to provide data for rows or entries that are either missing information, or require an update.	

MapRolesToUsers	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 require an update.	Using Jacl: `$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 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

MapRun AsRoles ToUsers	Maps RunAs Roles to users. The enterprise beans you that 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 require an update.	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]]]')`

MapShared LibForMod	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 you application.	Using Jacl: `$AdminApp install myapp.ear {-appname MyApp -MapSharedLibForMod {{EAR1 META/application.xml sharedlib1} {"IVT Application" ivt_ app.war,WEB-INF/web.xml sharedlib2} {"IVT EJB Module" ivtEJB.jar sharedlib3}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-appname MyApp -MapSharedLibForMod {{.* .* sharedlib1} {.* .* sharedlib2} {.* .* sharedlib3}}}` Using Jython: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapSharedLibFor Mod', [['EAR1', 'META/application.xml', 'sharedlib1'], ["IVT Applica tion", 'ivt_app.war,WEB-INF/web.xml', 'sharedlib2'], ["IVT EJB Module", 'ivtEJB.jar', 'sharedlib3']]])` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-appname', 'MyApp', '-MapSharedLibFor Mod', [['.*', '.*', 'sharedlib1'], ['.*', '.*', 'sharedlib2'], ['.*', '.*', 'sharedlib3']]])`

MapWebModToVH	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 require an update.	Using Jacl: `$AdminApp install myapp.ear {-MapWebModToVH {{"Default Application" default_app.war,WEB-INF/web.xml default_host} {"Examples Application" examples.war,WEB-INF/web.xml default_host}}}` Using Jacl with pattern matching: `$AdminApp install myapp.ear {-MapWebModToVH {{.* .* default_host}}}` Using Jython: `AdminApp.install('myapp.ear', '[-MapWebModToVH [["Default Application" default_app.war,WEB-INF/web.xml default_host] ["Examples Application" examples.war,WEB-INF/web.xml default_host]]]')` Using Jython with pattern matching: `AdminApp.install('myapp.ear', ['-MapWebModToVH', [['.*', '.*', 'default_host']]])`
noallowDispatch Remote Include	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.	

noallowPermIn FilterPolicy	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 it does not require a value.	
noallowService RemoteInclude	Disables the enterprise application that services an include request from an enterprise application that has the allowDispatchRemoteInclude option set to `true`.	
node	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.	

nocreateMBeans ForResources	Specifies that MBeans are not created for all resources such as, servlets, JSPs, 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	Specifies not to run the EJBDeploy tool during installation. This option is the default setting and it does not require a value.	
nodeployws	Specifies not to deploy Web services during installation. This option is the default setting and it does not require a value.	
nodistributeApp	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.	
noreloadEnabled	Disables class reloading. This option does not require a value. The default setting is the `reloadEnabled` option.	

nopreCompileJSPs	Specifies not to precompile JavaServer Pages files. This option is the default setting and it does not require a value.	
noprocessEmbeddedConfig	Use this option to ignore the embedded configuration data that is include in the application. This option does not required 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.	
nouseMetaDataFromBinary	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 it does not require a value. Use this option to indicate that the metadata that is used at run time comes from the enterprise archive file (EAR) file.	

nousedefault bindings	Specifies not to use default bindings for installation. This option is the default setting and it does not require a value.	
operation	Specifies the operation that you want to perform. This option only applies to the **update** command. 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 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]')` 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. 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	Use this option to process the embedded configuration data that is included in the application. This option does not required 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.	
preCompileJSPs	Specifies to precompile the JavaServer Pages files. This option does not require a value. The default is nopreCompileJSPs.	
reloadEnabled	Specifies that the file system of the application will be scanned for updated files so that changes reload dynamically. This option is the default setting and it does not require a value.	
reloadInterval	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.	

server	Specifies the server 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 application, this option only applies if the application contains a new module that does not exist in the installed application. You cannot use the -cluster and -server options together. If you want to deploy an application and specify the HTTP server during the deployment so that the application will appear in the generated plugin-cfg.xml file, you must first install the application with a target of -cluster. After you install the application and before you save, use **edit** command of the AdminApp object to add the additional mapping to the Web server.	

target	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 (+) or negative (-) sign to add or remove targets without having to specify the targets that are not changed.	Using Jacl: `$AdminApp install C:/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear {-appname MyApp -target WebSphere:cell=GooddogCell,node=GooddogNode,server=server2+WebSphere:cell=GooddogCell,node=BaddogNode,server=server3}` The following example removes *server3* from the application that is installed: `$AdminApp install C:/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear {-appname MyApp -target -WebSphere:cell=GooddogCell,node=BaddogNode,server=server3 -update -update.ignore.old}` The following example adds *server4* to the application that is installed: `$AdminApp upate app {-appname MyApp -target +WebSphere:cell=GooddogCell,node=GooddogNode,server=server4 -contents C:/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear -operation update -update.ignore.old}` Using Jython: `AdminApp.install("C:/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", ["-appname", "MyApp", "-target", "WebSphere:cell=GooddogCell,node=GooddogNode,server=server2+WebSphere:cell=GooddogCell,node=BaddogNode,server=server3"])` The following example removes *server3* from the application that is installed: `AdminApp.install("C:/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", ["-appname", "MyApp", "-target", "-WebSphere:cell=GooddogCell,node=BaddogNode,server=server3", "-update", "-update.ignore.old"])` The following example adds *server4* to the application that is installed: `AdminApp.upate("app", ["-appname", "MyApp", "-target", "+WebSphere:cell=GooddogCell,node=GooddogNode,server=server4", "-contents", "C:/ASV/o0528.02/WebSphere/AppServer/binaries/DefaultApplication.ear", "-operation", "update", "-update.ignore.old"])`

update	Updates the installed application with a new version of the enterprise archive file (EAR) file. This option does not require a value. The application that is being updated, 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 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	Specifies that during the update action, bindings from the new version of the application 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.	
update.ignore.old	Specifies that during the update action, the bindings from the installed version of the application 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.	

useMetaData FromBinary	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.	
usedefaultbindings	Specifies to use default bindings for installation. This option does not require a value. The default setting is nousedefaultbindings.	

validateinstall	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.	
verbose	Causes additional messages to display during installation. This option does not require a value.	

WebServicesClientBindDeployedWSDL	The immutable values for this option identify 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`.	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}}}` Using Jython: `AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindDeployedWSDL [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 META-INF/wsdl/DeployedWsdl1.wsdl]]]')` Using Jython with pattern matching: `AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindDeployedWSDL', ['.*', '.*', '.*', 'META-INF/wsdl/DeployedWsdl1.wsdl']]])`

	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`.	
WebServicesClientBindPortInfo	The immutable values identify 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.	Using Jacl: `$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPortInfo {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS 3000 newHTTP_ID newHTTP_pwd sslAliasConfig http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS}}}` Using Jacl with pattern matching: `$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]]]')` Using Jython with pattern matching: `AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindPortInfo', [['.*', '.*', '.*', '.*', '3000', 'newHTTP_ID', 'newHTTP_pwd', 'sslAliasConfig', 'http://yunus:9090/WSLoggerEJB/services/WSLoggerJMS']]])`

WebServicesClient BindPreferred Port	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.	Using Jacl: `$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort {{AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort}}}` Using Jacl with pattern matching: `$AdminApp install WebServicesSamples.ear {-WebServicesClientBindPreferredPort {{.* .* .* .* WSLoggerJMSPort}}}` Using Jython: `AdminApp.install('WebServicesSamples.ear', '[-WebServicesClientBindPreferredPort [[AddressBookW2JE.jar AddressBookW2JE service/WSLoggerService2 WSLoggerJMS WSLoggerJMSPort]]]')` Using Jython with pattern matching: `AdminApp.install('WebServicesSamples.ear', ['-WebServicesClientBindPreferredPort',[['.*', '.*', '.*', '.*', 'WSLoggerJMSPort']]])`

WebServicesServerBindPort	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`. .	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']]])`

WebServicesClient CustomProperty	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″}}	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	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/value pairs: {{″n1″ ″v1″}, {″n2″ ″v2″}, {″n3″ ″ v3″}}	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.response ContentEncoding', 'deflate']]])`

Usage table for the options of the AdminApp object install, installInteractive, update, updateInteractive, edit, and editInteractive commands:

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.

Option name	install and install Inter active command s - install an applicati on	update and update Inter active command s - Update an applicati on	update and update Inter active command s - Add a module	update and update Inter active command s - Update a module	edit and edit Inter active command s - Edit an applicati on	edit and edit Inter active command s - Edit a module
ActSpecJNDI	Yes	Yes	Yes	Yes	Yes	Yes
allowDispatch Remote Include	Yes	Yes			Yes	
allowPermInFilterPolicy	Yes	Yes				
allowService Remote Include	Yes	Yes			Yes	
appname	Yes	Yes				

BackendIdSelection	Yes	Yes	Yes	Yes		
BindJndiForEJBMess ageBinding	Yes	Yes	Yes	Yes	Yes	Yes
BindJndiForEJBNonMess ageBinding	Yes	Yes	Yes	Yes	Yes	Yes
cell	Yes	Yes	Yes			
cluster	Yes	Yes	Yes			
contents		Yes	Yes	Yes		
contenturi		Yes	Yes	Yes		
contextroot	Yes	Yes	Yes			
CorrectOracleIsolation Level	Yes	Yes	Yes	Yes	Yes	Yes
CorrectUseSystem Identity	Yes	Yes	Yes	Yes	Yes	Yes
createMBeansFor Resources	Yes	Yes			Yes	
CtxRootForWeb Mod	Yes	Yes	Yes	Yes	Yes	Yes
custom	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor10 CMPBeans	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor20 CMPBeans	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor10 EJBModules	Yes	Yes	Yes	Yes	Yes	Yes
DataSourceFor20 EJBModules	Yes	Yes	Yes	Yes	Yes	Yes
defaultbinding. datasource .jndi	Yes	Yes	Yes	Yes		
defaultbinding. cf.jndi	Yes	Yes	Yes	Yes		
defaultbinding. cf.resauth	Yes	Yes	Yes	Yes		
defaultbinding. datasource. password	Yes	Yes	Yes	Yes		
defaultbinding. datasource. username	Yes	Yes	Yes	Yes		
defaultbinding. ejbjndi. prefix	Yes	Yes	Yes	Yes		
defaultbinding. force	Yes	Yes	Yes	Yes		
defaultbinding. strategy.file	Yes	Yes	Yes	Yes		
defaultbinding. virtual.host	Yes	Yes	Yes	Yes		
depl.extension. reg (deprecated)						
deployejb	Yes	Yes	Yes	Yes		
deployejb.classpath	Yes	Yes	Yes	Yes		
deployejb.dbschema	Yes	Yes	Yes	Yes		
deployejb.dbtype	Yes	Yes	Yes	Yes		
deployejb.rmic	Yes	Yes	Yes	Yes		
deployws	Yes	Yes	Yes	Yes		
deployws.classpath	Yes	Yes	Yes	Yes		
deployws.jardirs	Yes	Yes	Yes	Yes		
distributeApp	Yes	Yes			Yes	
EmbeddedRar	Yes	Yes	Yes	Yes	Yes	Yes
EnsureMethod ProtectionFor 10EJB	Yes	Yes	Yes	Yes		

EnsureMethod ProtentionFor 20EJB	Yes	Yes	Yes	Yes		
filepermission	Yes	Yes			Yes	
JSPCompile Options	Yes	Yes	Yes	Yes		
JSPReloadFor WebMod	Yes	Yes	Yes	Yes	Yes	Yes
installdir (deprecated)						
installed.ear.destination	Yes	Yes			Yes	
MapInitParam ForServlet	Yes	Yes	Yes	Yes	Yes	Yes
MapMessage Destination RefToEJB	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			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			Yes	
noallowPermIn FilterPolicy	Yes	Yes				
noallowService Remote Include	Yes	Yes			Yes	
nocreateMBeans ForResources	Yes	Yes			Yes	
node	Yes	Yes	Yes			
nodeployejb	Yes	Yes	Yes	Yes		
nodeployws	Yes	Yes	Yes	Yes		
nodistributeApp	Yes	Yes			Yes	
nopreCompileJSPs	Yes	Yes	Yes	Yes		
noprocessEmbedded Config	Yes	Yes				
noreloadEnabled	Yes	Yes			Yes	
nousedefaultbindings	Yes	Yes	Yes	Yes		
nouseMetaData FromBinary	Yes	Yes			Yes	
operation		Yes	Yes	Yes		
preCompileJSPs	Yes	Yes	Yes	Yes		
processEmbedded Config	Yes	Yes				
reloadEnabled	Yes	Yes			Yes	
reloadInterval	Yes	Yes			Yes	
server	Yes	Yes	Yes			
target	Yes	Yes	Yes			
update	Yes	Yes				
update.ignore.old	Yes	Yes		Yes		
update.ignore.new	Yes	Yes		Yes		

useMetaData FromBinary	Yes	Yes			Yes	
usedefaultbindings	Yes	Yes	Yes	Yes		
validateinstall	Yes				Yes	
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

Pattern matching with the wsadmin tool

Before starting this task, the wsadmin tool must be running. See the “Starting the wsadmin scripting client” on page 135 article for more information.

Use pattern matching when you install, edit, or update 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. For more information about installing, editing, or updating an application, see the “Options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands” on page 769 article.

The following example uses pattern matching to install an application with the MapModulesToServers option:

- To install all WAR and JAR files to WebSphere Application Server:cell=myCell,node=myNode,server=myServer, use the following examples:
  - Using Jacl:

    ```
    $AdminApp install DefaultApplication.ear {-appname TEST
    -MapModulesToServers {{.* .* WebSphere:cell=myCell,node=myNode,server=myServer}}}
    ```

  - Using Jython:

    ```
    AdminApp.install('DefaultApplication.ear', ['-appname', 'TEST',
    '-MapModulesToServers', [['.*', '.*', 'WebSphere:cell=myCell,node=myNode,server=myServer']]])
    ```

- To install all WAR file to WebSphere Application Server :cell=myCell,node=myNode,server=myServer and all JAR files to WebSphere Application Server:cell=myCell,node=yourNode,server=yourServer, use the following examples:
  - 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}}}
    ```

  - 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']]])
    ```

Save the configuration changes. See the “Saving configuration changes with the wsadmin tool” on page 116 article for more information.

Example: Obtaining option information for AdminApp object commands

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.

- You can use the **options** command to see the requirements for an enterprise archive file (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.

  For example:

```
WASX7278I: Generated command line: install /websphere/appserver/installableapps/
jmsample.ear
{-BindJndiForEJBNonMessageBinding {{deplmtest.jar MailEJBObject deplmtest.jar,
META-INF/ejb-jar.xml ejb/JMSampEJB1 }}
-MapResRefToEJB {{deplmtest.jar MailEJBObject deplmtest.jar,META-INF/ejb-jar.xml
mail/MailSession9
javax.mail.Session mail/DefaultMailSessionX } {"JavaMail Sample WebApp"
 mtcomps.war,WEB-INF/web.xml
mail/MailSession9 javax.mail.Session mail/DefaultMailSessionY }}
-MapWebModToVH {{"JavaMail Sample WebApp"
mtcomps.war,WEB-INF/web.xml newhost }} -nopreCompileJSPs
-novalidateApp -installed.ear.destination
/mylocation -distributeApp -nouseMetaDataFromBinary}
```

Commands for the AdminTask object

Use the AdminTask object to run an administrative command.

Administrative commands are discovered 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 WebSphere Application Server that you have.

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.

In a deployment manager environment, configuration updates are available only if a scripting client is connected to a deployment manager. When connected to a node agent or a managed application server, you cannot update the configuration because the configuration for these server processes are copies of the master configuration, which resides in the deployment manager. The copies are created on a node machine when a configuration synchronization occurs between the deployment manager and the node agent. Make configuration changes to the server processes by connecting a scripting client to a deployment manager. To change a configuration, do not run a scripting client in local mode on a node machine because this is not supported.

The following AdminTask commands are available but do not belong to a group:

Some of the information in the table is split on multiple lines for printing purposes.

Command name:	Description:	Target object:	Parameters and return values:	Examples:
configureTAM				**Interactive mode example usage:** • Using Jacl: `$AdminTask configureTAM {-interactive}` • Using Jython string: `AdminTask.configureTAM ('[-interactive]')` • Using Jython list: `AdminTask.configureTAM (['-interactive'])`
createDefaultCGAP				**Interactive mode example usage:** • Using Jacl: `$AdminTask createDefaultCGAP {-interactive}` • Using Jython string: `AdminTask.createDefaultCGAP ('[-interactive]')` • Using Jython list: `AdminTask.createDefaultCGAP (['-interactive'])`
createGenericServerTemplate				**Interactive mode example usage:** • Using Jacl: `$AdminTask createGenericServerTemplate {-interactive}` • Using Jython string: `AdminTask.createGenericServerTemplate ('[-interactive]')` • Using Jython list: `AdminTask.createGenericServerTemplate (['-interactive'])`

create Server Type	Use the **createServerType** command to define a server type.	None	• Parameters: **-version** (String, required) **-serverType** (String, required) **-create Template Command** (String, required) **-createCommand** (String, required) **-defaultTemplateName** The default value is: default. (String, optional) **-defaultOSTemplateName** The default value is: default_zOS. (String, optional) **-configValidator** (String, optional) • Returns: The identification of the server type that you created, `javax.management.ObjectName`.	**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 string: `AdminTask.createServerType ('[-interactive]')` • Using Jython list: `AdminTask.createServerType (['-interactive'])`

createTCP EndPoint	The **createTCP EndPoint** command creates a new named end point that you can associate with a TCP inbound channel.	Parent instance of the Transport Channel Service that contains the TCPInbound Channel. (ObjectName, required)	• Parameters: **-name** Name for the new NamedEndPoint. (String, required) **- host** Host for the new NamedEndPoint. (String, required) **- port** Port for the new NamedEndPoint. (String, required) • Returns: Object name of the created NamedEndPoint.	**Batch mode example usage:** • Using Jacl: `$AdminTask createTCPEndPoint (cells/rohitbuildCell01/nodes/ rohitbuildCellManager01/servers/dmgr\|server.xml #TransportChannelService_1) {-name Sample_End_Pt_Name -host rohitbuild.raleigh.ibm.com -port 8978}` • Using Jython string: `AdminTask.createTCPEndPoint( 'cells/rohitbuildCell01/nodes /rohitbuildCellManager01/servers/dmgr\|server.xml #TransportChannelService_1', '[-name Sample_End_Pt_Name -host rohitbuild.raleigh.ibm.com -port 8978]')` • Using Jython list: `AdminTask.createTCPEndPoint ('cells/rohitbuildCell01/nodes /rohitbuildCellManager01/servers/dmgr\|server.xml #TransportChannelService_1', ['-name', 'Sample_End_Pt_Name', '-host', 'rohitbuild.raleigh.ibm.com', '-port', '8978'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask createTCPEndPoint {-interactive}` • Using Jython string: `AdminTask.createTCPEndPoint ('[-interactive]')` • Using Jython list: `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.	TCPInboundChannel, or containing chain, instance that is associated with a NamedEndPoint. (ObjectName, required)	• Parameters: None • Returns: Object name of an existing named end point that is associated with the TCP inbound channel instance or a channel chain.	**Batch mode example usage:** • Using Jacl: `$AdminTask getTCPEndPoint TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01/servers /dmgr\|server.xml#TCPInboundChannel_1)` `$AdminTask getTCPEndPoint DCS(cells /rohitbuildCell01/nodes/ rohitbuildCellManager01/servers/dmgr|server.xml #Chain_3)` • Using Jython string: `AdminTask.getTCPEndPoint('TCP_1(cells /rohitbuildCell01/ nodes/rohitbuildCellManager01/servers /dmgr|server.xml#TCPInboundChannel_1)')` `AdminTask.getTCPEndPoint('DCS(cells /rohitbuildCell01/nodes/ rohitbuildCellManager01/servers /dmgr|server.xml#Chain_3)')` • Using Jython list: `AdminTask.getTCPEndPoint('TCP_1(cells /rohitbuildCell01/ nodes/rohitbuildCellManager01/servers /dmgr|server.xml#TCPInboundChannel_1)')` `AdminTask.getTCPEndPoint('DCS(cells /rohitbuildCell01/nodes/ rohitbuildCellManager01/servers /dmgr|server.xml#Chain_3)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask getTCPEndPoint {-interactive}` • Using Jython string: `AdminTask.getTCPEndPoint ('[-interactive]')` • Using Jython list: `AdminTask.getTCPEndPoint (['-interactive'])`
help	The **help** command provides a summary of the help commands and ways to invoke an administrative command.	None	• Parameters: None • Returns: A general help description	• Using Jacl: `$AdminTask help` • Using Jython string: `print AdminTask.help()` • Using Jython list: `print AdminTask.help()`

help	The **help** command provides a list of available administrative commands if you use the `-commands` parameter or it provides a list of administrative command groups if you use the `-commandGroups` parameter.	None	• Parameters: **- commands** (String, optional) **- commandGroups** (String, optional) • Returns: A summary of all available administrative commands.	• Using Jacl: `$AdminTask help -commands` • Using Jython string: `AdminTask.help('-commands')` • Using Jython list: `AdminTask.help('-commands')`
help	If you provide the step name, this command provides help information for a given step of an administrative command. Otherwise, it provides help information for a given admin command or administrative command group. The stepName parameter is optional.	None	• Parameters: **- commandName** (String, optional) **- stepName** (String, optional) • Returns: A summary of the specified command group, administrative command, or step.	• Using Jacl: `$AdminTask help createJ2CConnectionFactory` • Using Jython string: `AdminTask.help('createJ2CConnectionFactory')` • Using Jython list: `AdminTask.help('createJ2CConnectionFactory')`

	The **listSSLRepertoires** command lists all of the Secure Sockets Layer (SSL) configuration instances that you can associate with an SSL inbound channel. If you create a new SSL alias using the administrative console, the alias name is automatically created in the *node_name*/*alias_name* format. However, if you create a new SSL alias using the wsadmin tool, you must create the SSL alias and specify both the node name and alias name in the *node_name*/*alias_name* format.	SSLInbound Channel instance for which the SSLConfig candidates are listed.	• Parameters: None • Returns: A list of eligible SSL configuration object names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listSSLRepertoires SSL_3(`*`cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2\|server.xml#SSLInboundChannel_1093445762330`*`)` • Using Jython string: `AdminTask.listSSLRepertoires('SSL_3(`*`cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2\|server.xml#SSLInboundChannel_1093445762330`*`)')` • Using Jython list: `AdminTask.listSSLRepertoires('SSL_3(`*`cells/rohitbuildCell01/nodes/rohitbuildNode01/servers/server2\|server.xml#SSLInboundChannel_1093445762330`*`)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listSSLRepertoires {-interactive}` • Using Jython string: `AdminTask.listSSLRepertoires ('[-interactive]')` • Using Jython list: `AdminTask.listSSLRepertoires (['-interactive'])`
listTAMSettings				**Interactive mode example usage:** • Using Jacl: `$AdminTask listTAMSettings {-interactive}` • Using Jython string: `AdminTask.listTAMSettings ('[-interactive]')` • Using Jython list: `AdminTask.listTAMSettings (['-interactive'])`

listTCPEndPoints	The **listTCPEndPoints** command lists all the named end points that can be associated with a TCP inbound channel.	TCP Inbound Channel instance for which named end points candidates are listed. (ObjectName, required)	• Parameters: **-excludeDistinguished** Shows only non-distinguished named end points. This parameter does not require a value. (Boolean, optional) **- unusedOnly** Shows the named end points not in use by other TCP inbound channel instances. This parameter does not require a value. (Boolean, optional) • Returns: A list of object names for the eligible named end points.	**Batch mode example usage:** • Using Jacl: `$AdminTask listTCPEndPoints TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr\|server.xml# TCPInboundChannel_1)` `$AdminTask listTCPEndPoints TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr|server.xml# TCPInboundChannel_1) {-excludeDistinguished}` `$AdminTask listTCPEndPoints TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr|server.xml# TCPInboundChannel_1) {-excludeDistinguished -unusedOnly}` • Using Jython string: `AdminTask.listTCPEndPoints( 'TCP_1(cells/rohitbuildCell01 /nodes/rohitbuildCellManager01 /servers/dmgr|server.xml #TCPInboundChannel_1)', '[-excludeDistinguished]')` `AdminTask.listTCPEndPoints( 'TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01/ servers/dmgr|server.xml# TCPInboundChannel_1)', '[-excludeDistinguished]')` `AdminTask.listTCPEndPoints( 'TCP_1(cells/rohitbuildCell01 /nodes/rohitbuildCellManager01 /servers/dmgr|server.xml #TCPInboundChannel_1)', '[-excludeDistinguished -unusedOnly]')` • Using Jython list: `AdminTask.listTCPEndPoints( 'TCP_1(cells/rohitbuildCell01 /nodes/rohitbuildCellManager01 /servers/dmgr|server.xml #TCPInboundChannel_1)', ['-excludeDistinguished'])` `AdminTask.listTCPEndPoints( 'TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr|server.xml# TCPInboundChannel_1)', ['-excludeDistinguished'])` `AdminTask.listTCPEndPoints( 'TCP_1(cells/rohitbuildCell01 /nodes/rohitbuildCellManager01 /servers/dmgr|server.xml #TCPInboundChannel_1)', ['-excludeDistinguished', '-unusedOnly'])`

				Interactive mode example usage: • Using Jacl: `$AdminTask listTCPEndPoints {-interactive}` • Using Jython string: `AdminTask.listTCPEndPoints ('[-interactive]')` • Using Jython list: `AdminTask.listTCPEndPoints (['-interactive'])`
listTCP Thread Pools	The **list TCP Thread Pools** command lists all of the thread pools that can be associated with a TCP inbound channel or TCP outbound channel.	TCPInboundChannel or TCPOutboundChannel instance for which ThreadPool candidates are listed. (ObjectName, required)	• Parameters: None • Returns: A list of eligible thread pool object names.	**Batch mode example usage:** • Using Jacl: `$AdminTask listTCPThreadPools TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr\|server.xml# TCPInboundChannel_1)` • Using Jython string: `AdminTask.listTCPThreadPools( 'TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr\|server.xml# TCPInboundChannel_1)')` • Using Jython list: `AdminTask.listTCPThreadPools( 'TCP_1(cells/rohitbuildCell01/ nodes/rohitbuildCellManager01 /servers/dmgr\|server.xml# TCPInboundChannel_1)')` **Interactive mode example usage:** • Using Jacl: `$AdminTask listTCPThreadPools {-interactive}` • Using Jython string: `AdminTask.listTCPThreadPools ('[-interactive]')` • Using Jython list: `AdminTask.listTCPThreadPools (['-interactive'])`
reconfigureTAM				**Interactive mode example usage:** • Using Jacl: `$AdminTask reconfigureTAM {-interactive}` • Using Jython string: `AdminTask.reconfigureTAM ('[-interactive]')` • Using Jython list: `AdminTask.reconfigureTAM (['-interactive'])`

set Resource Property	Use the **set Resource Property** command to set the value of a specified property defined on a resource provider such as JDBCProvider or a connection factory such as DataSource or JMSConnectionFactory. If the property with specified key is defined already, then this command overrides the value. If no property with a specified key is defined, this command will add the property with specified key and value.	The configuration object ID of a resource provider or a connection factory.	• Parameters: **- propertyName** The name of the property. (String, required) **- propertyValue** The value of a property. (String, required) **- propertyType** The type of the property. The default value is java.lang.String. (String, optional) **- propertyDescription** The description of the defined property. (String, optional) • Returns: None	**Batch mode example usage:** • Using Jacl: `$AdminTask setResourceProperty {-propertyName test.property -propertyValue testValue}` • Using Jython string: `AdminTask.setResourceProperty ('[-propertyName test.property -propertyValue testValue]')` • Using Jython list: `AdminTask.setResourceProperty (['-propertyName', 'test.property', '-propertyValue', 'testValue'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask setResourceProperty {-interactive}` • Using Jython string: `AdminTask.setResourceProperty ('[-interactive]')` • Using Jython list: `AdminTask.setResourceProperty (['-interactive'])`

show Resource Properties	Use the **show Resource Properties** command to list all of the property values that are defined on a resource provider such as JDBC provider or a connection factory such as data source or JMS connection factory.	The configuration object ID of a resource provider or a connection factory.	• Parameters: **- propertyName** The name of the property. If you specify the property name, the value of the specified property name is returned. If you do not specify the property name, all property values will be listed. Each element in the list is a property name value pair. (String, optional) • Returns: The property values that are defined on the resource provider or the connection factory that you specified.	**Batch mode example usage:** • Using Jacl: `$AdminTask showResourceProperties {-propertyName test.property}` • Using Jython string: `AdminTask.showResourceProperties ('[-propertyName test.property]')` • Using Jython list: `AdminTask.showResourceProperties (['-propertyName', 'test.property'])` **Interactive mode example usage:** • Using Jacl: `$AdminTask showResourceProperties {-interactive}` • Using Jython string: `AdminTask.showResourceProperties ('[-interactive]')` • Using Jython list: `AdminTask.showResourceProperties (['-interactive'])`
unconfigureTAM				**Interactive mode example usage:** • Using Jacl: `$AdminTask unconfigureTAM {-interactive}` • Using Jython string: `AdminTask.unconfigureTAM ('[-interactive]')` • Using Jython list: `AdminTask.unconfigureTAM (['-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.	None	• Parameters: **-ApplicationNames** The names of the applications that are updated. **-timeout** The timeout value in seconds for each node synchronization. The default is 300 seconds. • Returns: None	**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 string: `AdminTask.updateAppOnCluster ('[-interactive]')` • Using Jython list: `AdminTask.updateAppOnCluster (['-interactive'])`

	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, and in the `sas.client.props` file, when a RMI connector is used. This command is not supported in local mode.			

Service integration commands

The service integration functionality within the product provides a highly-flexible messaging system that supports a service-oriented architecture with a wide spectrum of quality of service options, supported protocols, and messaging patterns. It supports both message-oriented and service-oriented applications.

A variety of administrative commands are available for service integration, such as bus administrative commands, messaging engine administrative commands, mediation administrative commands, and so on. For information on the service integration commands, click on the administering applications section of the information center navigation, then click on Administering WebSphere applications, and then finally click on Service integration.

Administrative command invocation syntax

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:

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.

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.

Properties used by scripted administration

This article explains the Java properties that are used by scripted administration. Three levels of default property files load before any property file that is specified on the command line. The first level represents an installation default, located in the properties directory for each WebSphere 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. These 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.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`, `RMI`, or `NONE`. The `wsadmin.properties` file specifies `SOAP` as the connector.

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.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.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.traceString
Turns on tracing for the scripting process. The default has tracing turned off.

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 `logs` directory of each WebSphere 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.validationOutput
Determines where the validation reports are directed. The default file is `wsadmin.valout` which is located in the `logs` directory of each WebSphere 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.

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.tempdir
Determines the directory to use for temporary files when installing applications.

The Java virtual machine 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.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.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.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.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.

Chapter 4. Using administrative programs (JMX)

This topic describes how to use Java application programming interfaces (APIs) to administer WebSphere Application Server and to manage your applications.

You can administer WebSphere Application Server and your applications through tools that come with the product or through programming with the Java APIs.

The wsadmin scripting tool, the administrative console, and the administrative command-line tools come with the product. These administrative tools provide most of the functions that you need to manage the product and the applications that run in WebSphere Application Server. You can use the command-line tools from automation scripts to control the servers. Scripts that are written for the wsadmin scripting tool offer a wide range of possible custom solutions that you can develop quickly.

Investigate these tools with the Java APIs to determine the best ways to administer WebSphere Application Server and your applications. For information on the Java APIs, view Java Management Extensions (JMX) API documentation.

WebSphere Application Server supports access to the administrative functions through a set of Java classes and methods. You can write a Java program that performs any of the administrative features of the WebSphere Application Server administrative tools. You can also extend the basic WebSphere Application Server administrative system to include your own managed resources.

You can prepare, install, uninstall, edit, and update applications through programming. Preparing an application for installation involves collecting various types of WebSphere Application Server-specific binding information to resolve references that are defined in the application deployment descriptors. This information can also be modified after installation by editing a deployed application. Updating consists of adding, removing or replacing a single file or a single module in an installed application, or supplying a partial application that manipulates an arbitrary set of files and modules in the deployed application. Updating the entire application uninstalls the old application and installs the new one. Uninstalling an application removes it entirely from the WebSphere Application Server configuration.

Perform any or all of the following tasks to manage WebSphere Application Server and your Java 2 Platform, Enterprise Edition (J2EE) applications through programming.

- Create a JMX remote client program by using the JMX remote API (JSR 160)..

  This topic describes how to develop a JMX remote program that uses the JMX remote API (JSR 160) to access the WebSphere Application Server administrative system.
- Create a custom Java administrative client program using the Java administrative APIs.

  This topic describes how to develop a Java program that uses the WebSphere Application Server administrative APIs to access the administrative system of WebSphere Application Server.
- Extend the WebSphere Application Server administrative system with custom MBeans.

  This topic describes how to extend the WebSphere Application Server administration system by supplying and registering new JMX MBeans in one of the Application Server processes. In this case, you can use the administrative classes and methods to add newly managed objects to the administrative system.
- Deploy and manage a custom Java administrative client program for use with multiple Java 2 Platform, Enterprise Edition application servers.

  This topic describes how to connect to a J2EE server, and how to manage multiple vendor servers.
- Manage applications through programming

  This topic describes how, through Java MBean programming, to install, update, and delete a J2EE application on WebSphere Application Server.
- Extend application management operations through programming

This topic describes how, through programming, to use the common deployment framework to add additional logic to application management operations.

Depending on which tasks you complete, you have created your own administrative program, extended the WebSphere Application Server administrative console, connected and managed vendor servers, or managed your applications through programming.

You can continue to administer WebSphere Application Server and your applications through programming or in combination with the tools that come with the WebSphere Application Server.

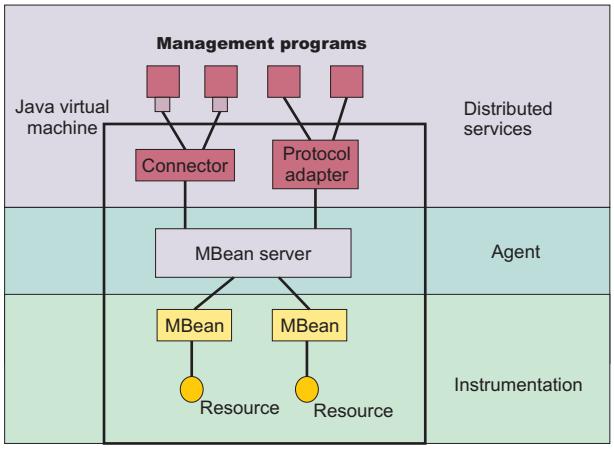
Java Management Extensions

This topic gives an overview of Java Management Extensions (JMX) in general and how this standard applies to WebSphere Application Server.

Java Management Extensions overview

Java Management Extensions (JMX) is the Java standard for managing application resources. The management architecture that is defined by JMX is divided into three levels:

- The bottom level is management instrumentation. Each manageable resource is described by an interface that specifies the attributes it has, the operations it supports, and the notifications it sends. This resource is a managed bean (MBean).
- The middle level is the management agent. Each managed process contains a JMX agent that includes an MBean server, which provides a registry and access point for MBeans. Management clients must use the MBean server to access the registered MBeans.
- The top level of the architecture is identified, but undefined in the current level of the JMX specification. It is the distributed services level, and its role is to facilitate remote access to JMX agents. This task is accomplished through connectors, which provide a protocol-independent, location-transparent, client-side interface to the MBean server (for example, a Remote Method Invocation (RMI) connector), or protocol adapters, which provide protocol-specific, server-side access to the MBean server (for example, an HTTP adapter).



Java Management Extensions in WebSphere Application Server

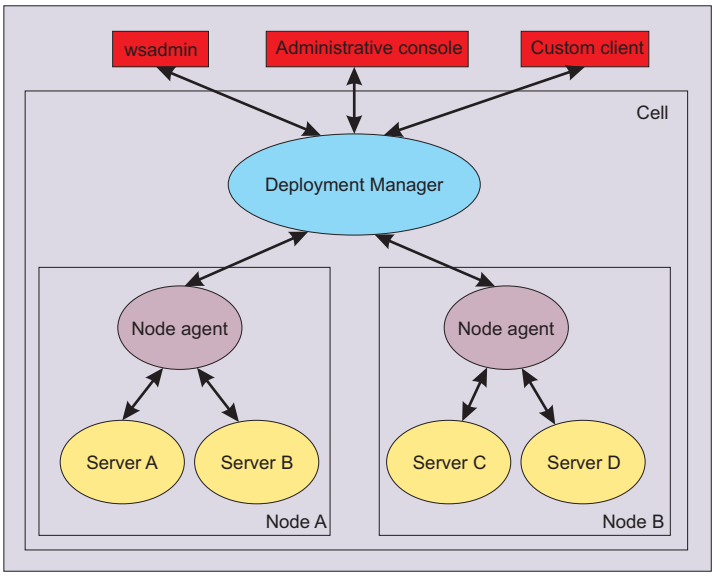
Java Management Extensions (JMX) is at the core of Application Server administration capabilities. The application server contains a JMX agent. All of the system components are defined as MBeans. The JMX agent in Application Server supports three types of connectors, Remote Method Invocation/Internet Inter-ORB Protocol (RMI/IIOP), Simple Object Access Protocol/Hypertext Transfer Protocol (SOAP/HTTP),

and Simple Object Access Protocol/Hypertext Transfer Protocol Secure (SOAP/HTTPS), which provides remote access to the server resources. All of the administration tools included with Application Server use these JMX facilities to accomplish their functions.

In a stand-alone Application Server installation, servers exist and are administered individually. An administrative client connects directly to the Application Server in this environment.

In a Network Deployment installation, a hierarchical topology groups application servers within nodes and groups nodes within a cell. Administrative servers exist at the node level (node agents) and at the cell level (the deployment manager), and act as aggregation points for the administrative services in the subordinate servers.

MBeans in all servers on a node are visible through that node agent, and MBeans in all nodes are visible through the deployment manager. Therefore, by connecting to the deployment manager, you can invoke operations, can get and set attributes, and can receive notifications for any MBean in the cell. Application Server provides an AdminService class that reflects the standard JMX MBeanServer interface, and wraps the MBeanServer interface so that it takes part in implementing this distributed management functionality.



Creating a custom Java administrative client program using WebSphere Application Server administrative Java APIs

This section describes how to develop a Java program for accessing the WebSphere Application Server administrative system by using the WebSphere Application Server administrative application programming interfaces (APIs).

This task assumes a basic familiarity with Java Management Extensions (JMX) API programming. See the JMX API documentation for information.

When you develop and run administrative clients that use various JMX connectors and that have security enabled, use the following guidelines. When you follow these guidelines, you guarantee the behavior among different implementations of JMX connectors. Any programming model that strays from these guidelines is unsupported.

1. Create and use a single administrative client before you create and use another administrative client.
2. Create and use an administrative client on the same thread.
3. Use one of the following ways to specify a user ID and password to create a new administrative client:

- Specify a default user ID and password in the property file.
- Specify a user ID and password other than the default. Once you create an administrative client with a nondefault user ID and password, specify the nondefault user ID and password when you create subsequent administrative clients.

1. Develop an administrative client program.
2. If your administrative client uses Simple Object Access Protocol (SOAP) as its Connector, you must configure Java Secure Socket Extension (JSSE) as your transport layer. WebSphere Application Server uses supports Secure Sockets Layer (SSL) and Transport Layer Security (TLS) with the Java Secure Sockets Extension (JSSE) and System SSL packages. Create digital certificates for the user ID used to run your administrative client. If you want to use System Authorization Facility (SAF) to create digital certificates and store them in a SAF keyring, refer to Defining SSL Security for Client Outbound Requests. (You can save the name of the keyring you create for use in the next step.)
3. Update the `soap.client.props` file in the *profile_root*`/properties` directory being used by your administrative client with the name of the SAF keyring. Refer to the second step in Using System Authorization Facility keyrings with Java Secure Sockets Extension for directions on updating the `soap.client.props` file.
4. Build the administrative client program.

   Compile it with the javac command and provide the location of the necessary JAR files in the classpath argument.

   For example, if your installation directory is `/DeploymentManager` a typical command would look like the following example:

   ```
   javac -extdirs "$JAVA_HOME/lib/ext;
   /DeploymentManager/classes;/DeploymentManager/lib;
   /DeploymentManager/lib/ext"  MyAdminClient.java
   ```

   (The previous command is split on multiple lines for publication.)
5. Run the administrative client program.

   Run the administrative client program by setting up the run-time environment so that the program can find all of the prerequisites. Many of the batch or script files in the `bin` directory under the installation root perform a similar function. The following is an example of a batch file that runs an administrative client program named `MyAdminClient` follows:

   ```
   @echo off

   binDir=`dirname "$0"`
   . "$binDir/setupCmdLine.sh"


   "$JAVA_HOME/bin/java" "$CLIENTSOAP" "-Dwas.install.root=$WAS_HOME"
   "-Dwas.repository.root=$CONFIG_ROOT"
   -Dcom.ibm.CORBA.BootstrapHost=$COMPUTERNAME
    "-Djava.ext.dirs=$JAVA_HOME/lib/ext;$WAS_HOME/classes;
   $WAS_HOME/lib;$WAS_HOME/lib/ext" MyAdminClient $@
   ```

   (The contents of the previous batch file is split on multiple lines for publication.)

Developing an administrative client program

This topic describes how to develop an administrative client program that utilizes WebSphere Application Server administrative application programming interfaces (APIs) and Java Management Extensions (JMX).

WebSphere Application Server administrative APIs provide control of the operational aspects of your distributed system as well as the ability to update your configuration. This topic also demonstrates examples of MBean operations. For information, view the JMX API documentation or the MBean API documentation.

1. Create an AdminClient instance.

An administrative client program needs to invoke methods on the AdminService object that is running in the deployment manager or the application server in the base installation. The AdminClient class provides a proxy to the remote AdminService object through one of the supported Java Management Extensions (JMX) connectors. The following example shows how to create an AdminClient instance:

```
Properties connectProps = new Properties();
connectProps.setProperty(
AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);

connectProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
connectProps.setProperty(AdminClient.CONNECTOR_PORT, "8879");
AdminClient adminClient = null;
try
{
       adminClient = AdminClientFactory.createAdminClient(connectProps);
}
catch (ConnectorException e)
{
       System.out.println("Exception creating admin client: " + e);
}
```

a. Set up a Properties object.

   The example sets up a Properties object with the properties that are required to get to your server. In this case, you use the Simple Object Access Protocol (SOAP) connector to reach the server; for the connector type, use the value: AdminClient.CONNECTOR_TYPE_SOAP.

b. Set the port number on which the server SOAP connector is listening.

   In a single server installation, the default port number for the application server SOAP connector is 8880. In a Network Deployment installation, the default port number for the deployment manager SOAP connector is 8879.

c. After the connection properties are set, use the AdminClientFactory class and the Properties object to create an AdminClient object that is connected to your chosen server.

   Depending on factors such as your desired protocol and security environment, you might need to set other properties. For more detailed information about the AdminClient interface and additional creation examples, refer to the AdminClient interface in the JMX API documentation.

2. Find an MBean. When you obtain an AdminClient instance, you can use it to access managed resources in the administration servers and application servers. Each managed resource registers an MBean with the AdminService through which you can access the resource. The MBean is represented by an ObjectName instance that identifies the MBean. An ObjectName instance consists of a domain name followed by an unordered set of one or more key properties. The syntax for the domain name follows:

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

   For WebSphere Application Server, the domain name is `WebSphere` and the key properties defined for administration are as follows:

type	The type of MBean. For example: Server, TraceService, Java virtual machine (JVM).
name	The name identifier for the individual instance of the MBean.
cell	The name of the cell that the MBean is running.
node	The name of the node that the MBean is running.
process	The name of the process that the MBean is running.

   Some MBeans in WebSphere Application Server use additional key properties. An MBean without key properties can be registered with the MBean server in a WebSphere Application Server process. However, such an MBean cannot participate in the distributed enhancements that WebSphere Application Server adds, for example, request routing, distributed event notification, and so on.

If you know the complete set of key properties for an ObjectName instance, you can use it to find the MBean it identifies. However, finding MBeans without having to know all of their key properties is usually more practical and convenient. Use the wildcard character asterisk (*) for any key properties that you do not need to match. The following table provides some examples of object names with wildcard key properties that match single or multiple MBeans.

:type=Server,	All MBeans of type Server
:node=Node1,type=Server,	All MBeans of type Server on Node1
:type=JVM,process=server1,node=Node1,	The JVM MBean in the server named server1 node Node1
:process=server1,	All MBeans in all servers named server1
:process=server1,node=Node1,	All MBeans in the server named server1 on Node1

You can locate an MBean by querying for it with object names that match key properties. The following example shows how to find the MBean for the node agent of node, MyNode:

```
String nodeName = "MyNode";
String query = "WebSphere:type=NodeAgent,node=" + nodeName + ",*";
ObjectName queryName = new ObjectName(query);
ObjectName nodeAgent = null;
Set s = adminClient.queryNames(queryName, null);
if (!s.isEmpty())
    nodeAgent = (ObjectName)s.iterator().next();
else
    System.out.println("Node agent MBean was not found");
```

a. Build an ObjectName instance with a query string that specifies the key properties of type and node.

   By using a wildcard for the remaining key properties, this pattern matches the object names for all MBeans of the type NodeAgent on the node MyNode. Because only one node agent per node exists, this information is sufficient to identify the MBean that you want.

b. Give this ObjectName instance to the queryNames method of the AdminClient interface.

   The AdminClient interface performs the remote call to the AdminService interface to obtain the set of MBean object names that match the query. The null second parameter to this method is a query expression (QueryExp) object that you can use as an additional query over the MBeans that match the ObjectName pattern in the first parameter.

c. Use the set iterator to get the first and, in this case, only element.

   The element is the MBean ObjectName instance of the node agent.

3. Use the MBean. What a particular MBean can do depends on the management interface of that MBean. An MBean can declare:
   - Attributes that you can obtain or set
   - Operations that you can invoke
   - Notifications for which you can register listeners
   -

   For the MBeans provided by WebSphere Application Server, you can find information about the interfaces they support in the MBean API documentation. The following example invokes one of the operations available on the NodeAgent MBean that you located previously. The following example starts the *MyServer* application server:

```
String opName = "launchProcess";
String signature[] = { "java.lang.String" };
String params[] = { "MyServer" };
try
{
     adminClient.invoke(nodeAgent, opName, params, signature);
}
```

```
catch (Exception e)
{
     System.out.println("Exception invoking launchProcess: " + e);
}
```

The AdminClient.invoke method is a generic means of invoking any operation on any MBean. The parameters are:

- The object name of the target MBean, `nodeAgent`
- The name of the operation, `opName`
- An object array that contains the operation parameters, `params`
- A string array that contains the operation signature, `signature`

The launchProcess operation in the example has a single parameter which is a string that identifies the server to start.

The invoke method returns an object instance, which the calling code can use to cast to the correct return type for the invoked operation. The launchProcess operation is declared void so that you can ignore the return value in this example.

4. Register for events. In addition to managing resources, the JMX API also supports application monitoring for specific administrative events. Certain events produce notifications, for example, when a server starts. Administrative applications can register as listeners for these notifications. The WebSphere Application Server provides a full implementation of the JMX notification model, and provides additional function so you can receive notifications in a distributed environment. For a complete list of the notifications emitted from WebSphere Application Server MBeans, refer to the com.ibm.websphere.management.NotificationConstants class in the MBean API documentation The following example shows how an object can register for event notifications that are emitted from an MBean using the ObjectName node agent:

   ```
   adminClient.addNotificationListener(nodeAgent, this, null, null);
   ```

   In this example, the first parameter is the ObjectName for the node agent MBean. The second parameter identifies the listener object, which must implement the NotificationListener interface. In this case, the calling object is the listener. The third parameter is a filter that you can use to indicate which notifications you want to receive. When you leave this value as `null`, you receive all notifications from this MBean. The final parameter is a handback object that you can use to set the JMX API to return to you when it emits a notification.

   If your MBean is located on another server in the cell, you can receive its notifications even though your administrative client program might be connected to the deployment manager server. All notifications flow to the upstream server. For example, a notification from an application server first flows to the local node agent and then to the deployment manager.

   Another enhanced feature that Application Server provides is the ability to register as a notification listener of multiple MBeans with one call. This registration is done through the addNotificationListenerExtended method of the AdminClient interface, an extension of the standard JMX addNotificationListener method. This extension method even lets you register for MBeans that are not currently active. This registration is important in situations where you want to monitor events from resources that can be stopped and restarted during the lifetime of your administrative client program.

5. Handle the events. Objects receive JMX event notifications through the handleNotification method, which is defined by the NotificationListener interface and which any event receiver must implement. The following example is an implementation of the handleNotification method that reports the notifications that it receives:

   ```
   public void handleNotification(Notification n, Object handback)
   {
        System.out.println("************************************************");
        System.out.println("* Notification received at " + new Date().toString());
        System.out.println("* type      = " + ntfyObj.getType());
        System.out.println("* message   = " + ntfyObj.getMessage());
        System.out.println("* source    = " + ntfyObj.getSource());
        System.out.println(
        "* seqNum    = " + Long.toString(ntfyObj.getSequenceNumber()));
   ```

```
        System.out.println("* timeStamp = " + new Date(ntfyObj.getTimeStamp()));
        System.out.println("* userData  = " + ntfyObj.getUserData());
        System.out.println("************************************************");
    }
```

Administrative client program example

This example is a complete administrative client program.

Copy the contents to a file named `MyAdminClient.java`. After changing the node name and server name to the appropriate values for your configuration, you can compile and run it using the instructions from Creating a custom Java administrative client program using WebSphere Application Server administrative Java APIs

```
import java.util.Date;
import java.util.Properties;
import java.util.Set;

import javax.management.InstanceNotFoundException;
import javax.management.MalformedObjectNameException;
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.ObjectName;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.exception.ConnectorException;

public class AdminClientExample implements NotificationListener
{

    private AdminClient adminClient;
    private ObjectName nodeAgent;
    private long ntfyCount = 0;

    public static void main(String[] args)
    {
       AdminClientExample ace = new AdminClientExample();

       // Create an AdminClient
       ace.createAdminClient();

       // Find a NodeAgent MBean
       ace.getNodeAgentMBean("ellington");

       // Invoke launchProcess
       ace.invokeLaunchProcess("server1");

       // Register for NodeAgent events
       ace.registerNotificationListener();

       // Run until interrupted
       ace.countNotifications();
    }

    private void createAdminClient()
    {
        // Set up a Properties object for the JMX connector attributes
        Properties connectProps = new Properties();
        connectProps.setProperty(
        AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
        connectProps.setProperty(AdminClient.CONNECTOR_HOST, "localhost");
        connectProps.setProperty(AdminClient.CONNECTOR_PORT, "8879");

        // Get an AdminClient based on the connector properties
        try
        {
```

```
            adminClient = AdminClientFactory.createAdminClient(connectProps);
        }
        catch (ConnectorException e)
        {
            System.out.println("Exception creating admin client: " + e);
            System.exit(-1);
        }

        System.out.println("Connected to DeploymentManager");
    }


    private void getNodeAgentMBean(String nodeName)
    {
        // Query for the ObjectName of the NodeAgent MBean on the given node
        try
        {
            String query = "WebSphere:type=NodeAgent,node=" + nodeName + ",*";
            ObjectName queryName = new ObjectName(query);
            Set s = adminClient.queryNames(queryName, null);
            if (!s.isEmpty())
                nodeAgent = (ObjectName)s.iterator().next();
            else
            {
                System.out.println("Node agent MBean was not found");
                System.exit(-1);
            }
        }
        catch (MalformedObjectNameException e)
        {
            System.out.println(e);
            System.exit(-1);
        }
        catch (ConnectorException e)
        {
            System.out.println(e);
            System.exit(-1);
        }

        System.out.println("Found NodeAgent MBean for node " + nodeName);
    }

    private void invokeLaunchProcess(String serverName)
    {
        // Use the launchProcess operation on the NodeAgent MBean to start
        // the given server
        String opName = "launchProcess";
        String signature[] = { "java.lang.String" };
        String params[] = { serverName };
        boolean launched = false;
        try
        {
            Boolean b = (Boolean)adminClient.invoke(

nodeAgent, opName, params, signature);
            launched = b.booleanValue();
            if (launched)
                System.out.println(serverName + " was launched");
            else
                System.out.println(serverName + " was not launched");

        }
        catch (Exception e)
        {
            System.out.println("Exception invoking launchProcess: " + e);
        }
    }
```

```
    private void registerNotificationListener()
    {
        // Register this object as a listener for notifications from the
        // NodeAgent MBean.  Don't use a filter and don't use a handback
        // object.
        try
        {
            adminClient.addNotificationListener(nodeAgent, this, null, null);
            System.out.println("Registered for event notifications");
        }
        catch (InstanceNotFoundException e)
        {
            System.out.println(e);
        }
        catch (ConnectorException e)
        {
            System.out.println(e);
        }
    }

    public void handleNotification(Notification ntfyObj, Object handback)
    {
        // Each notification that the NodeAgent MBean generates will result in
        // this method being called
        ntfyCount++;
        System.out.println("*************************************************");
        System.out.println("* Notification received at " + new Date().toString());
        System.out.println("* type      = " + ntfyObj.getType());
        System.out.println("* message   = " + ntfyObj.getMessage());
        System.out.println("* source    = " + ntfyObj.getSource());
        System.out.println(
        "* seqNum    = " + Long.toString(ntfyObj.getSequenceNumber()));
        System.out.println("* timeStamp = " + new Date(ntfyObj.getTimeStamp()));
        System.out.println("* userData  = " + ntfyObj.getUserData());
        System.out.println("*************************************************");

    }

    private void countNotifications()
    {
        // Run until killed
        try
        {
            while (true)
            {
                Thread.currentThread().sleep(60000);
                System.out.println(ntfyCount + " notification have been received");
            }
        }
        catch (InterruptedException e)
        {
        }
    }

}
```

Creating a Java Management Extensions client program using the Java Management Extensions Remote application programming interface

This topic describes how to develop and build a Java Management Extensions (JMX) client program that is compliant with JMX Remote application programming interface (JSR 160). After you have a working JMX client program, you can use it to manage WebSphere Application Server or non-WebSphere Application Server systems.

This task assumes a basic familiarity with JSR 160 and JMX application programming interface (API) programming. For information on JSR 160, see http://www.jcp.org/en/jsr/detail?id=160. For information on the JMX APIs, see the JMX API documentation.

When you develop and run JMX clients that use various JMX connectors and that have security enabled, use the following guidelines. When you follow these guidelines, you guarantee the behavior among different implementations of JMX connectors. Any programming model that strays from these guidelines is unsupported.

1. Create and use a single JMX client before you create and use another JMX client.
2. Create and use a JMX client on the same thread.
3. Use one of the following ways to specify a user ID and password to create a new JMX client:
   - Specify a default user ID and password in the property file.
   - Specify a user ID and password other than the default. After you create a JMX client with a nondefault user ID and password, specify the nondefault user ID and password when you create subsequent JMX clients.

1. Develop a JMX client program.
2. Build the JMX remote client program.

   Compile the program with the javac and provide the location of the `ibm.admin.thinclient.jar` file in the classpath argument.

   For example, if your `ibm.admin.thinclient.jar` file is in the `/opt/resources/ibm.admin.thinclient.jar` path, and you want to compile the `JMXRemoteClientApp.java` file in the current directory, use the following settings and commands:

   ```
   CLASSPATH=/opt/resources/ibm.ws.admin.thinclient.jar:${CLASSPATH}
   export CLASSPATH
   ${JAVA_HOME}/bin/javac JMXRemoteClientApp.java
   ```

3. Run the JMX client program.

   Run the JMX client program by setting up the runtime environment so that the program can find all of the prerequisites. Many of the batch or script files in the `bin` directory under the installation root perform a similar function. The following example is a batch file that runs the JMXRemoteClientApp JMX client program:

   ```
   #!/bin/sh
   CONNECTORPROPS=-Dcom.ibm.CORBA.configURL=<location of sas.client.props>
   JAVA_HOME=<location of Java>
   TAC_CLASSPATH=<location of ibm.admin.thinclient.jar>


   "${JAVA_HOME}/bin/java"  \
   -Djava.ext.dirs="${JAVA_HOME}/jre/lib/ext"  \
   -classpath "${TAC_CLASSPATH}"  ${CONNECTORPROPS}  JMXRemoteClientApp $@
   ```

   (The contents of the previous batch file are split on multiple lines for publication.)

You have developed, built, and run a JMX client program that is JSR 160 compliant.

Developing a Java Management Extensions client program using Java Management Extensions Remote application programming interface

This topic describes how to develop a Java Management Extensions (JMX) connector specification and JMX Remote application programming interface (API) (JSR 160). The program can communicate by Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP)

This topic assumes a basic understanding of JSR 160, JMX APIs, and managed beans (MBeans). For more information on JSR 160, see http://www.jcp.org/en/jsr/detail?id=160. For more information on the JMX APIs, see the JMX API documentation. For more information on MBeans, see the MBean API documentation.

You can administer your WebSphere Application Server environment through the administrative console, the wsadmin utility, or Java Management Extensions (JMX) programming. This topic discusses how to develop a JMX remote client program using the JMX remote API so that you can administer your WebSphere Application Server environment through JMX programming.

1. Specify the JMX connector address for the server through the JMXServiceURL class.

   The form of the JMXServiceURL class is service:jmx:rmi://*host name*:*port*/jndi/JMXConnector
2. Specify the user ID and password for the server, if security is enabled.
3. Establish the JMX connection.
4. Get the MBean server connection instance.

You have established a connection to WebSphere Application Server through an RMI connection and started the WebSphere Application Server through the node agent.

```
import java.util.Date;
import java.util.Set;
import java.util.Hashtable;

import javax.management.InstanceNotFoundException;
import javax.management.MalformedObjectNameException;
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.ObjectName;
import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;



public class JMXRemoteClientApp implements NotificationListener {

   private MBeanServerConnection mbsc = null;
   private ObjectName nodeAgent;
   private long ntfyCount = 0;

   public static void main(String[] args)
   {
      try {

         JMXRemoteClientApp client = new JMXRemoteClientApp();

         String hostname=args[0];
         String port=args[1];
         String nodeName =args[2];

         client.connect(hostname,port);

         // Find a node agent MBean
         client.getNodeAgentMBean(nodeName);
```

```
      // Invoke the launch process.
      client.invokeLaunchProcess("server1");

      // Register for node agent events
      client.registerNotificationListener();

      // Run until interrupted.
      client.countNotifications();

   } catch (Exception e) {
      e.printStackTrace();
   }
}

private void connect(String hostname,String port) throws Exception
{
   String jndiPath="/jndi/JMXConnector";

   JMXServiceURL url = new JMXServiceURL("service:jmx:iiop://"+hostname+":"+port+jndiPath);

   Hashtable h  = new Hashtable();

   //Specify the user ID and password for the server if security is enabled on server.

   // String[] credentials = new String[] {username ,password };
   //h.put("jmx.remote.credentials", credentials);


   //Establish the JMX connection.

   JMXConnector jmxc = JMXConnectorFactory.connect(url, h);

   //Get the MBean server connection instance.

   mbsc = jmxc.getMBeanServerConnection();

   System.out.println("Connected to DeploymentManager");
}


private void getNodeAgentMBean(String nodeName)
{
   // Query for the object name of the node agent MBean on the given node
   try {
      String query = "WebSphere:type=NodeAgent,node=" + nodeName + ",*";
      ObjectName queryName = new ObjectName(query);
      Set s = mbsc.queryNames(queryName, null);
      if (!s.isEmpty())
         nodeAgent = (ObjectName)s.iterator().next();
      else {
         System.out.println("Node agent MBean was not found");
         System.exit(-1);
      }
   } catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
   }

   System.out.println("Found NodeAgent MBean for node " + nodeName);
}

private void invokeLaunchProcess(String serverName)
{
   // Use the launch process on the node agent MBean to start
   // the given server.
   String opName = "launchProcess";
   String signature[] = { "java.lang.String"};
```

```
        String params[] = { serverName};
        boolean launched = false;
        try {
           Boolean b = (Boolean)mbsc.invoke(nodeAgent, opName, params, signature);
           launched = b.booleanValue();
           if (launched)
              System.out.println(serverName + " was launched");
           else
              System.out.println(serverName + " was not launched");

        } catch (Exception e) {
           System.out.println("Exception invoking launchProcess: " + e);
        }
     }

     private void registerNotificationListener()
     {
        // Register this object as a listener for notifications from the
        // node agent MBean.  Do not use a filter and do not use a handback
        // object.
        try {
           mbsc.addNotificationListener(nodeAgent, this, null, null);
           System.out.println("Registered for event notifications");
        } catch (Exception e) {
           System.out.println(e);
        }
     }

     public void handleNotification(Notification ntfyObj, Object handback)
     {
        // Each notification that the node agent MBean generates results in
        // a call to this method.
        ntfyCount++;
        System.out.println("*************************************************");
        System.out.println("* Notification received at " + new Date().toString());
        System.out.println("* type      = " + ntfyObj.getType());
        System.out.println("* message   = " + ntfyObj.getMessage());
        System.out.println("* source    = " + ntfyObj.getSource());
        System.out.println(
                          "* seqNum    = " + Long.toString(ntfyObj.getSequenceNumber()));
        System.out.println("* timeStamp = " + new Date(ntfyObj.getTimeStamp()));
        System.out.println("* userData  = " + ntfyObj.getUserData());
        System.out.println("*************************************************");

     }

     private void countNotifications()
     {
        // Run until stopped.
        try {
           while (true) {
              Thread.currentThread().sleep(60000);
              System.out.println(ntfyCount + " notification have been received");
           }
        } catch (InterruptedException e) {
        }
     }

}
```

Extending the WebSphere Application Server administrative system with custom MBeans

You can extend the WebSphere Application Server administration system by supplying and registering new Java Management Extensions (JMX) MBeans (see JMX 1.x Specification for details) in one of the WebSphere processes.

JMX MBeans represent the management interface for a particular piece of logic. All of the managed resources within the standard WebSphere infrastructure are represented as JMX MBeans. There are a variety of ways in which you can create your own MBeans and register them with the JMX MBeanServer running in any WebSphere process. For more information, view the MBean API documentation.

1. Create custom JMX MBeans.

   You have some alternatives to select from, when creating MBeans to extend the WebSphere administrative system. You can use any existing JMX MBean from another application. You can register any MBean that you tested in a JMX MBean server outside of the WebSphere Application Server environment in a WebSphere Application Server process, including standard MBeans, dynamic MBeans, open MBeans, and model MBeans.

   In addition to any existing JMX MBeans, and ones that were written and tested outside of the WebSphere Application Server environment, you can use the special distributed extensions provided by WebSphere and create a WebSphere ExtensionMBean provider. This alternative provides better integration with all of the distributed functions of the WebSphere administrative system. An ExtensionMBean provider implies that you supply an XML file that contains an MBean Descriptor based on the DTD shipped with the WebSphere Application Server. This descriptor tells the WebSphere system all of the attributes, operations, and notifications that your MBean supports. With this information, the WebSphere system can route remote requests to your MBean and register remote Listeners to receive your MBean event notifications.

   All of the internal WebSphere MBeans follow the Model MBean pattern. Pure Java classes supply the real logic for management functions, and the WebSphere MBeanFactory class reads the description of these functions from the XML MBean Descriptor and creates an instance of a ModelMBean that matches the descriptor. This ModelMBean instance is bound to your Java classes and registered with the MBeanServer running in the same process as your classes. Your Java code now becomes callable from any WebSphere Application Server administrative client through the ModelMBean created and registered to represent it.

   User MBeans that run on both the WebSphere Application Server distributed platforms and the WebSphere Application Server for z/OS platform may require special coding to function properly in the z/OS multiprocess model. On WebSphere Application Server distributed platforms where each application server runs in a single Java™ Virtual Machine (JVM), there is only one MBean server. The MBean server controls all MBeans that are registered within that application server. On the WebSphere Application Server for z/OS platform, there is a control process and a federation of servant processes, each with their own MBean server. The control process has its own MBean proxy server to distribute requests among the servant processes. See the detailed discussion of the JMX MBean multiprocess model request flow.

2. Optionally define an explicit MBean security policy.

   If you do not define an MBean security policy, WebSphere Application Server uses the default security policy.

3. Register the new MBeans. There are various ways to register your MBean.

   You can register your MBean with the WebSphere Application Server administrative service.

   You can register your MBean with the MBeanServer in a WebSphere Application Server process. The following list describes the available options in order of preference:

   - Go through the MBeanFactory class. If you want the greatest possible integration with the WebSphere Application Server system, then use the MBeanFactory class to manage the life cycle of your MBean through the activateMBean and deactivateMBean methods of the MBeanFactory class. Use these methods, by supplying a subclass of the RuntimeCollaborator abstract superclass and an

XML MBean descriptor file. Using this approach, you supply a pure Java class that implements the management interface defined in the MBean descriptor. The MBeanFactory class creates the actual ModelMBean and registers it with the WebSphere Application Server administrative system on your behalf.

This option is recommended for registering model MBeans.

- Use the JMXManageable and CustomService interface. You can make the process of integrating with WebSphere administration even easier by implementing a CustomService interface that also implements the JMXManageable interface. Using this approach, you can avoid supplying the RuntimeCollaborator. When your CustomService interface is initialized, the WebSphere MBeanFactory class reads your XML MBean descriptor file and creates, binds, and registers an MBean to your CustomService interface automatically. After the shutdown method of your CustomService is called, the WebSphere Application Server system automatically deactivates your MBean.
- Go through the AdminService interface. You can call the registerMBean() method on the AdminService interface and the invocation is delegated to the underlying MBeanServer for the process, after appropriate security checks. You can obtain a reference to the AdminService using the getAdminService() method of the AdminServiceFactory class.

  This option is recommended for registering standard, dynamic, and open MBeans. Implement the UserCollaborator class to use the MBeans and to provide a consistent level of support for them across distributed and z/OS platforms.

  For the z/OS platform, an MBean registered through the registerMBean() method on the AdminService interface is not visible from outside the server and can only be invoked from within the servant process in which it was registered.
- Get MBeanServer instances directly. You can get a direct reference to the JMX MBeanServer instance running in any WebSphere Application Server process, by calling the getMBeanServer() method of the MBeanFactory class. You get a reference to the MBeanFactory class by calling the getMBeanFactory() method of the AdminService interface.

  When a custom MBean is registered directly with the MBean server, the MBean object name is enhanced with the cell, node and process name keys by default. This registration allows the MBean to participate in the distributed features of the WebSphere Application Server administrative system. You can turn off the default behavior by setting the com.ibm.websphere.mbeans.disableRouting custom property.

  See the *Installing your application serving environment* PDF for more information on the com.ibm.websphere.mbeans.disableRouting custom property.

Regardless of the approach used to create and register your MBean, you must set up proper Java 2 security permissions for your new MBean code. The WebSphere AdminService and MBeanServer are tightly protected using Java 2 security permissions and if you do not explicitly grant your code base permissions, security exceptions are thrown when you attempt to invoke methods of these classes. If you are supplying your MBean as part of your application, you can set the permissions in the `was.policy` file that you supply as part of your application metadata. If you are using a CustomService interface or other code that is not delivered as an application, you can edit the `library.policy` file in the node configuration, or even the `server.policy` file in the `properties` directory for a specific installation.

Best practices for standard, dynamic, and open MBeans

This article discusses recommended guidelines for standard, dynamic, and open MBeans.

The underlying interface for the WebSphere Application Server administrative service is AdminService. Remote access occurs through the AdminControl scripting object.

For WebSphere Application Server Version 5, the MBean registration and capabilities are as follows:

MBean type	Registered with:	Capabilities

Model	WebSphere Application Server administrative service	Local access is through the WebSphere Application Server administrative service or the MBean server. Remote access is through the WebSphere Application Server administrative service, and WebSphere Application Server security.Remote access is also through z/OS system extensions.
Standard, dynamic, or open	MBean server	Local access is through the WebSphere Application Server administrative service or the MBean server on the distributed platform. Local access is only through the MBean server.

best-practices: For V6, you can optionally register standard, dynamic, and open custom MBeans with the WebSphere Application Server administrative service to take advantage of the capabilities that in V5 are available only to model MBeans.

V6 introduces a special run-time collaborator that you use with standard, dynamic or open custom MBeans to register the custom MBeans with the WebSphere Application Server administrative service. The standard, dynamic, and open MBeans display in the administrative service as model MBeans. The administrative service uses the capabilities available to MBeans that are registered with the administrative service.

For WebSphere Application Server Version 6, the MBean registration and capabilities are as follows:

MBean type	Registered with:	Capabilities
Model, and optionally standard, dynamic, or open	WebSphere Application Server administrative service	Local access is through the WebSphere Application Server administrative service or the MBean server. Remote access is through the WebSphere Application Server administrative service, and WebSphere Application Server security. Remote access is also through z/OS system extensions.
Standard, dynamic, or open	MBean server	Local access is through the WebSphere Application Server administrative service or the MBean server on the distributed platform. Local access is only through the MBean server.Remote access is through the WebSphere Application Server administrative service, the Java Management Extensions (JMX) Remote application programming interface (API) (JSR 160) client code, and WebSphere Application Server security.

Creating and registering standard, dynamic, and open custom MBeans

You can create standard, dynamic, and open custom MBeans and register them with the WebSphere Application Server administrative service.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Perform the following tasks to create and register a standard, dynamic, or open custom MBean.

1. Create your particular MBean class or classes.
2. Write an MBean descriptor in the XML language for your MBean.
3. Register your MBean by inserting code that uses the WebSphere Application Server run-time com.ibm.websphere.management.UserMBeanCollaborator collaborator class into your application code.
4. Package the class files for your MBean interface and implementation, the descriptor XML file, and your application Java archive (JAR) file.

After you successfully complete the steps, you have a standard, dynamic, or open custom MBean that is registered and activated with the WebSphere Application Server administrative service.

The following example shows how to create and register a standard MBean with the WebSphere Application Server administrative service:

SnoopMBean.java:

```
/**
 * Use the SnoopMBean MBean, which has a standard mbean interface.
 */
public interface SnoopMBean {
    public String getIdentification();
    public void snoopy(String parm1);
}
```

SnoopMBeanImpl.java:

```
/**
 * SnoopMBeanImpl - SnoopMBean implementation
 */
public class SnoopMBeanImpl implements SnoopMBean {
    public String getIdentification() {
        System.out.println(">>> getIdentification() called...");
        return "snoopy!";
    }

    public void snoopy(String parm1) {
        System.out.println(">>> snoopy(" + parm1 + ") called...");
    }
}
```

Define the following MBean descriptor for your MBean in an .xml file. The getIdentification method is set to run with the unicall option and the snoopy method is set to use the multicall option. These options are used only for z/OS platform applications. The WebSphere Application Server for z/OS options are not applicable to the distributed platforms, but they do not need to be removed. The options are ignored on the distributed platforms. . Some statements are split on multiple lines for printing purposes.

SnoopMBean.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SnoopMBean"
 version="5.0"
 platform="dynamicproxy"
 description="Sample SnoopMBean to be initialized inside an EJB.">


 <attribute name="identification" getMethod="getIdentification"
type="java.lang.String" proxyInvokeType="unicall"/>
```

```
 <operation name="snoopy" role="operation"  type="void" targetObjectType="objectReference"
   impact="ACTION" proxyInvokeType="multicall">
  <signature>
   <parameter name="parm1" description="test parameter" type="java.lang.String"/>
  </signature>
 </operation>
</MBean>
```

Assume that your MBean is used in an enterprise bean. Register your MBean in the enterprise bean ejbCreate method and unregister it in the ejbRemove method.

```
//The method MBeanFactory.activateMBean() requires four parameters:
//String type: The type value that you put in this MBean's descriptor. For this example
//the string type is SnoopMBean.
//RuntimeCollaborator co: The UserMBeanCollaborator user MBean collaborator instance
//that you create
//String id: Unique name that you pick
//String desciptor: The MBean descriptor file name


import com.ibm.websphere.management.UserMBeanCollaborator;
//Import other classes here.
.
.
.
static private ObjectName snoopyON = null;
static private Object lockObj = "this is a lock";
.
.
.
/**
 * ejbCreate method: Register your Mbean.
 */
public void ejbCreate() throws javax.ejb.CreateException {
    synchronized (lockObj) {
        System.out.println(">>> SnoopMBean activating for --|" + this + "|--");
        if (snoopyON != null) {
            return;
        }
        try {
            System.out.println(">>> SnoopMBean activating...");
            MBeanFactory mbfactory = AdminServiceFactory.getMBeanFactory();
            RuntimeCollaborator snoop = new UserMBeanCollaborator(new SnoopMBeanImpl());
            snoopyON = mbfactory.activateMBean("SnoopMBean", snoop, "snoopMBeanId",
"SnoopMBean.xml");
            System.out.println(">>> SnoopMBean activation COMPLETED! --|" + snoopyON + "|--");
        } catch (Exception e) {
            System.out.println(">>> SnoopMBean activation FAILED:");
            e.printStackTrace();
        }
    }
}
.
.
.
/**
 * ejbRemove method: Unregister your MBean.
 */
public void ejbRemove() {
    synchronized (lockObj) {
        System.out.println(">>> SnoopMBean Deactivating for --|" + this + "|--");
        if (snoopyON == null) {
            return;
        }
        try {
            System.out.println(">>> SnoopMBean Deactivating ==|" + snoopyON + "|== for --|"
+ this + "|--");
```

```
            MBeanFactory mbfactory = AdminServiceFactory.getMBeanFactory();
            mbfactory.deactivateMBean(snoopyON);
            System.out.println(">>> SnoopMBean Deactivation COMPLETED!");
        } catch (Exception e) {
            System.out.println(">>> SnoopMBean Deactivation FAILED:");
            e.printStackTrace();
        }
    }
}
```

Compile the MBean Java files and package the resulting class files with the descriptor .xml file, into the enterprise bean JAR file.

Setting Java 2 security permissions

You must configure Java 2 security permissions to use Java Management Extension and WebSphere Application Server administrative methods.

When you enable Java 2 security, you must grant Java 2 security permissions to application-specific code for Java Management Extensions (JMX) and WebSphere Application Server administrative privileges. With these permissions, your application code can call WebSphere Application Server administrative methods and JMX methods.

If you are using Java 2 security then you need to verify that your extensions and application server can access the required resources. The following steps show how to configure access for JMX and the application server administrative methods.

- Use the following permission to invoke all the JMX class methods and interface methods:

  ```
  permission javax.management.MBeanPermission "*", "*";
  ```

  Consult the Java Management Extensions (JMX) API documentation for specific actions under the MBeanPermission class.
- Use the following permission for WebSphere Application Server administrative application programming interfaces (APIs):

  ```
  permission com.ibm.websphere.security.WebSphereRuntimePermission "AdminPermission";
  ```

Java Management Extensions MBean multiprocess model request flow for WebSphere Application Server for z/OS

Using the Java Management Extensions (JMX) dynamic proxy capability, applications that depend on JMX operations can exhibit consistent behavior whether the server architecture uses a single process model or a multiprocess model.

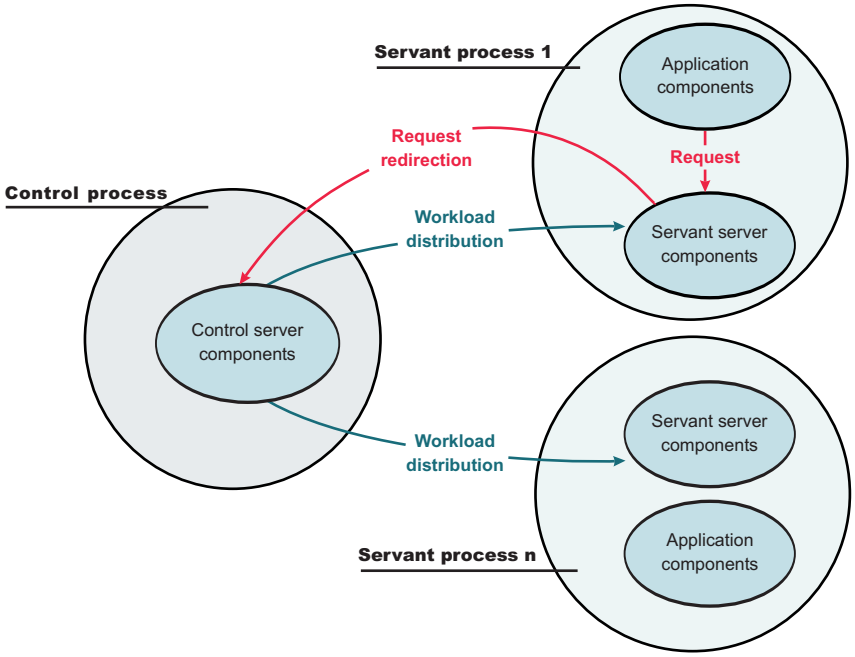
All the MBeans that the WebSphere Application Server runtime provides are capable of running under the single process model employed by WebSphere Application Server on distributed platforms, or the multiprocess model employed by WebSphere Application Server for z/OS. User MBean providers might need to modify their MBeans so that they work on both the WebSphere Application Server distributed platforms and the WebSphere Application Server for z/OS. For more information, view the Application Server application programing interface (API) documentation.

The simplified dynamic proxy model presented here discusses the two general request flows that exist in the multiprocess model. Operation requests on an MBean can be initiated from one of two places:

- Within an application component running under the same servant process as the MBean.
- Outside the server through one of the JMX connectors (SOAP, Remote Method Invocation (RMI), HTTP, and so on).

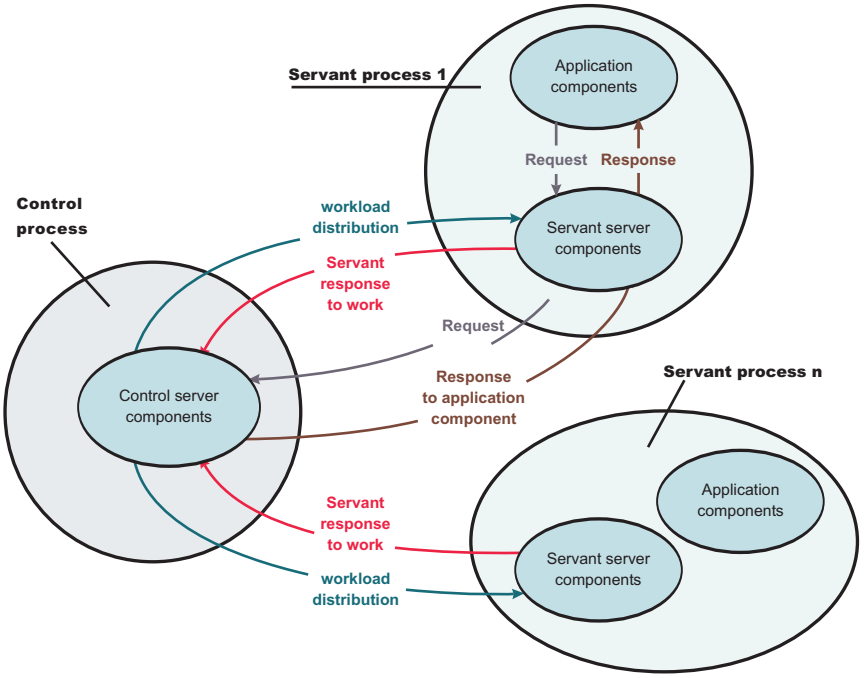
Requests that come from an application component generally follow this flow. The application component sends its request to the servant components. The servant components redirect the request to the control

process where the dynamic proxy for the MBean runs. WebSphere Application Server automatically generates the MBean dynamic proxy during runtime. If the request calls a method that the MBean provider defined with a unicall option, the dynamic proxy in the control process randomly dispatches the work, with MVS workload management (WLM), to one servant process. If the request calls a method that the MBean provider defined with a multicall option, the dynamic proxy in the control process, in conjunction with WLM, distributes the work to all the servant processes.

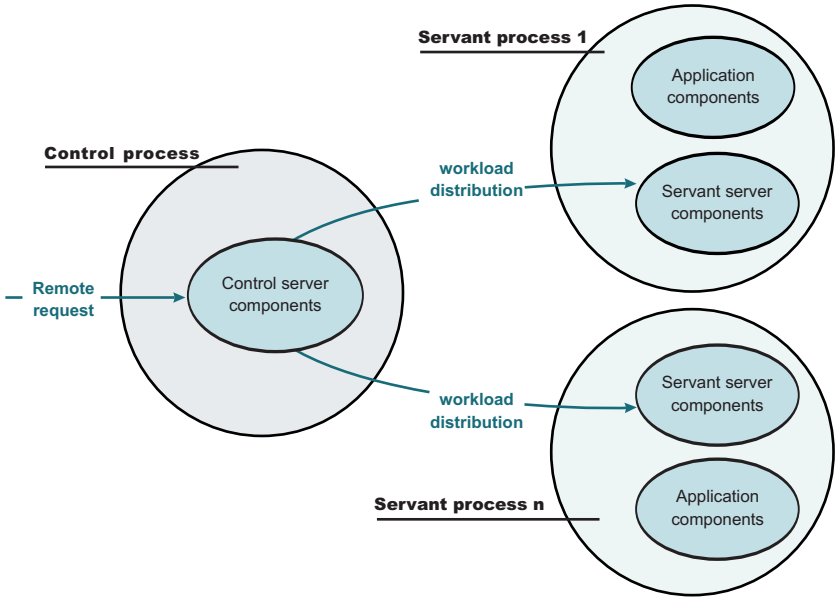


Each servant process that completes the work optionally sends a response back to the control process. If the MBean was defined with a unicall option, and the return type is anything but void, the control process returns the response to the servant that made the request. The servant server components then return the response to the application component. If the MBean was defined with a multicall option, the MBean inside each servant process runs separately and finishes processing the request at different times. After all the requests are processed, you might need a result aggregation and an event aggregation to properly return

a result to the application component.

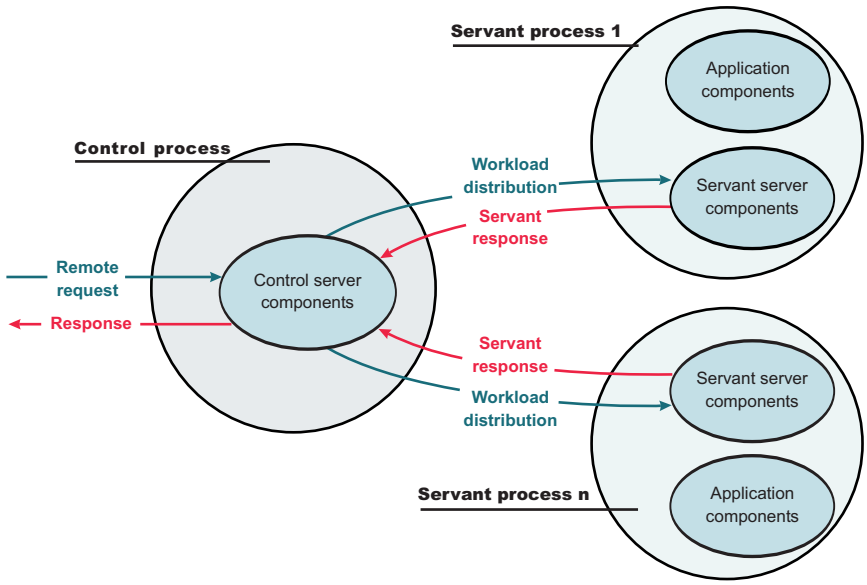


Requests that come from a remote location outside the server generally follow this flow. The remote location sends its request to the control process where the dynamic proxy for the MBean runs. WebSphere Application Server automatically generates the MBean dynamic proxy during run time. The next part of the request process behaves the same as a request that originates from an application component. If the request calls a method that the MBean provider defined with a unicall option the dynamic proxy in the control process randomly dispatches the work, with MVS workload management (WLM), to one servant process. If the request calls a method that the MBean provider defined with a multicall option, the dynamic proxy in the control process, in conjunction with WLM, distributes the work to all the servant process.



The flow of the response back to the remote location is similar to the response back to the application component. Each servant process that completes the work optionally sends a response back to the control

process. If the MBean was defined with a unicall option, and the return type is anything but void, the control process returns the response to the remote location. If the MBean was defined with a multicall option, the MBean that runs inside each servant process runs separately and finishes processing the request at different times. After all the requests nare processed, you might need a result aggregation and an event aggregation to properly return a result to the remote location.



Java Management Extensions dynamic proxy concepts

A Java Management Extensions (JMX) dynamic proxy coordinates MBean requests among multiprocess servers. This section discusses the main terms associated with a JMX dynamic proxy.

Control process

Receives requests and distributes them to servant processes so that the application server can do work for the requests.

Servant process

Receives work from the control process and carries out the work.

Unicall option versus the multicall option

Use the unicall option on the proxyInvocationType method when a request invokes an arbitrary servant process or servant processes. Use the multicall option on the proxyInvocationType method when a request goes to multiple servant processes and the servant processes return different results.

The following example shows an MBean descriptor that was developed for a single process model (before) and modified for a multiprocess model (after).

Before

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SampleStateMBean"
  version="6.0"
  description="Sample State MBean for the documentation example.">

  <attribute description="The name of the MBean."
    getMethod="getMBeanName" name="mbeanName" type="java.lang.String"/>
  <attribute description="The state of the MBean."name="state"
    getMethod="getState" setMethod="setState" type="java.lang.String"/>
  <operation
    description="Initialize the State MBean."
    impact="ACTION" name="initializeState" role="operation"
    targetObjectType="objectReference" type="void">
    <signature>
```

```
      <parameter description="The name of the MBean."
        name="mbeanName" type="java.lang.String"/>
      <parameter description="The initial state of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
  </operation>
</MBean>
```

After

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SampleStateMBean"
  version="6.0"
  platform="dynamicproxy"
  description="Sample State MBean for the documentation example.">

  <attribute description="The name of the MBean."
    getMethod="getMBeanName" name="mbeanName" type="java.lang.String"/>
  <attribute description="The state of the MBean."name="state"
    getMethod="getState" setMethod="setState" type="java.lang.String"/>
    proxyInvokeType="unicall" proxySetterInvokeType="multicall"/>
  <operation
    description="Initialize the State MBean."
    impact="ACTION" name="initializeState" role="operation"
    targetObjectType="objectReference" type="void" proxyInvokeType="multicall">
    <signature>
      <parameter description="The name of the MBean."
        name="mbeanName" type="java.lang.String"/>
      <parameter description="The initial state of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
  </operation>
</MBean>
```

Make the user MBean run in dynamic proxy mode by specifying `dynamicproxy` on the platform attribute. If no platform attribute exists on the MBean descriptor, the user MBean deployed on WebSphere Application Server for z/OS automatically uses the dynamic proxy mode.

Update the attribute XML tag or the operation XML tag, as shown in the After example, to specify the unicall behavior or the multicall behavior in the multiprocess environment. If no proxyInvokeType option or proxySetterInvokeType option exists, the behavior defaults to unicall. In the After example, the getMBeanName method and the getState method run with unicall behavior. The setState method and the initializeState method run with multicall behavior.

Single process model

The single process application server has one server run time. The MBean generally acts on one instance of each major run time component: one Enterprise JavaBeans (EJB) container, one Web container, one Java 2 Platform, Enterprise Edition (J2EE) connection manager, and so on. This model assumes that each MBean invocation on the server runs in the same process and the same Java Virtual Machine (JVM).

Multiprocess model

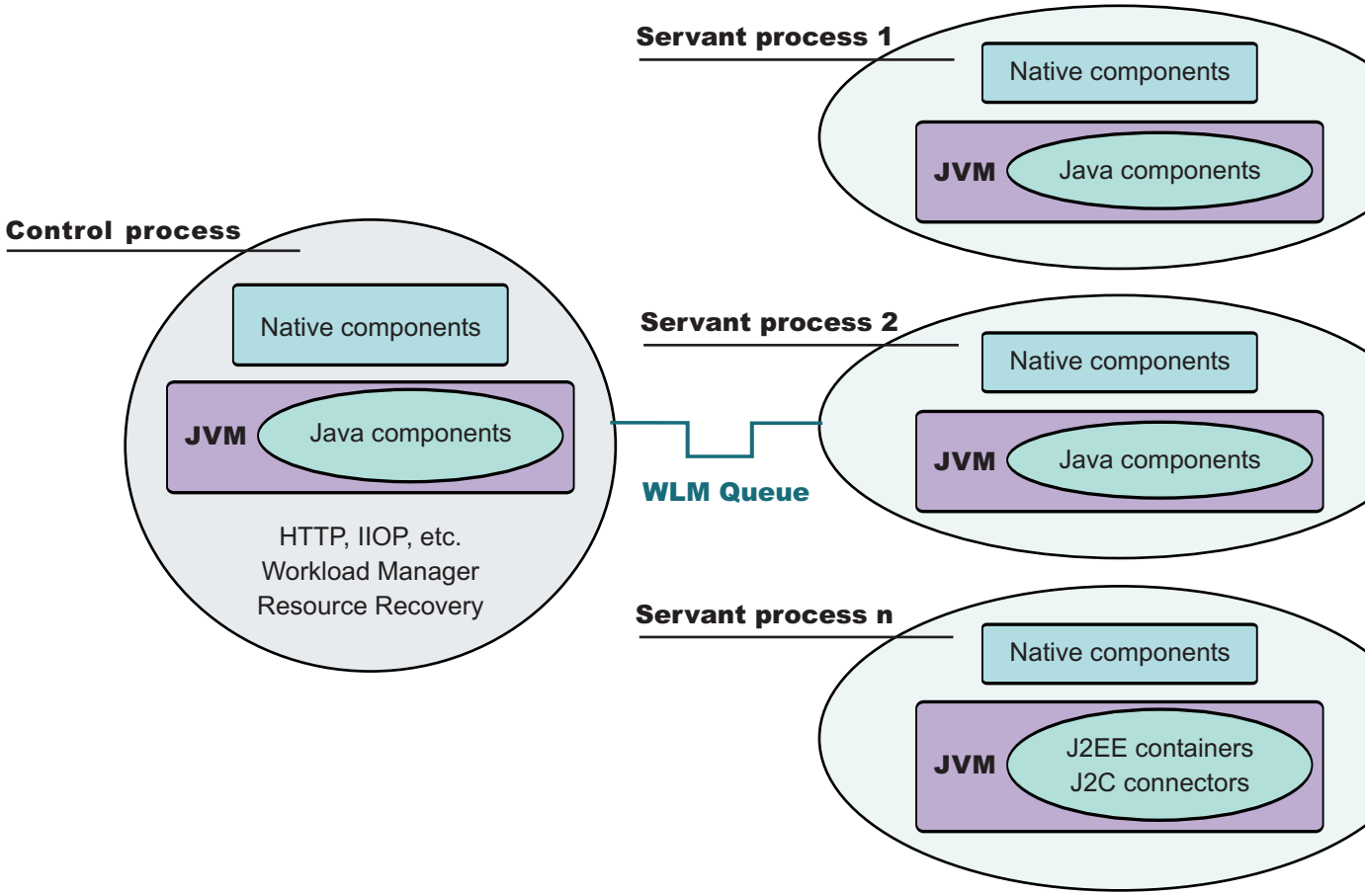
The multiprocess model asserts that a single server instance is a federation of Java virtual machines (JVMs), that run in a separate operating process. The control process is responsible for such server functions as communication endpoints, authorization, resource recovery, and workload management. All other JVMs are worker JVMs, in which application requests run. These JVMs take direction from, and interact only with the control process.

All inbound and outbound requests go through the control process. Client requests arrive at the control process. The control process, with assistance from the MVS workload manager (WLM), dispatches the work to the servant processes.

The number of servant processes is managed by WLM and varies based on demand. The demand is measured against installation-specific performance goals, expressed as WLM policy. Each servant process is identical and hosts the necessary application server components to enable the J2EE application programming model. The servant processes rely on the control process for numerous services, such as communication, security, and transaction control.

The multiprocess model imposes additional demands on the Java Management Extension (JMX) infrastructure over the single process model. Administrative requests to a multiprocess server often require coordination among the processes that comprise the application server. The JMX infrastructure includes additional facilities to enable this coordination.

WebSphere Application Server for z/OS



State object support for dynamic proxy MBean

com.ibm.websphere.management.dynamicproxy.StateObject class: The MBean provider extends the StateObject abstract class. Specify the subclass of the StateObject class so that the JMX run time can instantiate it before the dynamic proxy MBean completes its initialization. The JMX run time attaches StateObject class to the dynamic proxy Invocation Handler interface to keep track of the current state of the dynamic proxy before and after the MBean method runs. The JMX run time also attaches the StateObject class to the Result Aggregation interface class as well as the Event Handler interface class to support appropriate aggregation application.

Result aggregation handler support interface

com.ibm.websphere.management.dynamicproxy.AggregationHandler class: The result aggregation handler support interface defines the method that an MBean provider uses to handle result aggregation in a dynamic proxy-enabled WebSphere Application Server for z/OS MBean. Specify the aggregationHandlerClass attribute on the MBeanDescriptor MBean XML tag. Implement the interface for MBean methods that use the multicall proxyInvokeType option and that return a value.

The interface determines the method for which this aggregation is processed. It then properly aggregates all servant MBean results that the servant processes pass back to the control process, and then compiles a single result to return to the caller.

Event aggregation handler support interface

com.ibm.websphere.management.dynamicproxy.EventHandler interface class: The event aggregation handler support interface defines the method that an MBean provider uses to handle event aggregation in a dynamic proxy-enabled WebSphere Application Server for z/OS MBean. Specify the eventHandlerClass attribute on the MBeanDescriptor MBean XML tag. The interface handles all incoming servant MBean events and aggregates them to filter out duplicate events from multiple servant MBeans. It sends one event back to the listener of the dynamic proxy MBean. The interface adjusts the current dynamic proxy MBean state according to the MBean provider requirements.

Invocation handler support interface

com.ibm.websphere.management.dynamicproxy.InvocationHandler class: The invocation handler support interface defines the preInvoke and postInvoke methods that a WebSphere Application Server for z/OS dynamic proxy MBean implements when it requires state management information. The MBean uses the information to coordinate with the servant MBeans in cases where the multicall invocation type is required. Specify the invocationHandlerClass attribute on the MBeanDescriptor MBean XML tag. Use the interface for dynamic proxy MBeans that require state management before and after invoking a method that changes its state.

User MBean

The user MBean resides in the servant process and handles requests through its dynamically created proxy MBean, which runs inside the control process. An MBean provider can package handlers with the user MBean so that the provider hooks in his own specialized processing for the following situations:

- Result aggregation
- Event aggregation
- Invocation handling
- State management of objects

Example: The SampleStateMBean MBean

Use this example to guide you in developing user MBeans that work for the WebSphere Application Server on both the distributed platforms and the z/OS platform. The example uses all the special handlers to show its dynamic proxy MBean responsibilities and capabilities. The SampleStateMBean example keeps track of its state and generates state change events when it invokes setter methods.

MBeanDescriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="SampleStateMBean"
  aggregationHandlerClass="com.ibm.ws390.sample.SampleStateAggregationHandler"
  eventHandlerClass="com.ibm.ws390.sample.SampleStateEventHandler"
  invocationHandlerClass="com.ibm.ws390.sample.SampleStateInvocationHandler"
  stateObjectClass="com.ibm.ws390.sample.SampleState"
  version="6.0"
  platform="dynamicproxy"
  description="Sample State MBean for the documentation example.">

  <attribute description="The name of the MBean."
    getMethod="getMBeanName" name="mbeanName" type="java.lang.String"
    proxyInvokeType="unicall"/>
  <attribute description="The state of the MBean." name="state"
    getMethod="getState" setMethod="setState" type="java.lang.String"
    proxyInvokeType="multicall" proxySetterInvokeType="multicall"/>
  <operation
    description="Initialize the State MBean."
    impact="ACTION" name="initializeState" role="operation"
```

```
    targetObjectType="objectReference" type="void" proxyInvokeType="multicall">
    <signature>
      <parameter description="The name of the MBean."
        name="mbeanName" type="java.lang.String"/>
      <parameter description="The initial state of the MBean."
        name="mbeanName" type="java.lang.String"/>
    </signature>
  </operation>

  <notification name="j2ee.state.starting" severity="6" log="false"
                description="This sample state MBean is in starting state.">
      <notificationType>j2ee.state.starting</notificationType>
  </notification>
  <notification name="j2ee.state.running" severity="6" log="false"
                description="This sample state MBean is in running state.">
      <notificationType>j2ee.state.running</notificationType>
  </notification>
  <notification name="j2ee.state.stopping" severity="6" log="false"
                description="This sample state MBean is in stopping state.">
      <notificationType>j2ee.state.stopping</notificationType>
  </notification>
  <notification name="j2ee.state.stopped" severity="6" log="false"
                description="This sample state MBean is in stopped state.">
      <notificationType>j2ee.state.stopped</notificationType>
  </notification>
</MBean>
```

SampleState implementation

```
package com.ibm.ws390.sample;

import com.ibm.ejs.ras.Tr;
import com.ibm.ejs.ras.TraceComponent;
import java.io.Serializable;
import com.ibm.websphere.management.dynamicproxy.StateObject;

public class SampleState extends StateObject {

    private static TraceComponent tc =
    Tr.register(SampleState.class,"SampleState",null);

    // Package protected STATE constants.
    static final String STATE_STARTING = "j2ee.state.starting";
    static final String STATE_RUNNING  = "j2ee.state.running";
    static final String STATE_STOPPING = "j2ee.state.stopping";
    static final String STATE_STOPPED  = "j2ee.state.stopped";

    // Dynamicproxy State is initialized with STOPPED state.
    private String state = STATE_STOPPED;

    public SampleState() {
        if (tc.isEntryEnabled()) Tr.entry(tc,"<init>");

        // State is initialized during "state" initialization above,
        // but can also be initialized here in the constructor as well.
        /*
        state = "WebSphere Application Server for z/OS ready for e-business";
        */

        if (tc.isEntryEnabled()) Tr.exit(tc,"<init>");
    }

    public synchronized String getState() {
        if (tc.isEntryEnabled()) Tr.entry(tc,"getState");
        if (tc.isEntryEnabled()) Tr.exit(tc,"getState",state);
        return state;
    }
```

```
    public synchronized void setState(String state) {
        if (tc.isEntryEnabled()) Tr.entry(tc,"setState",state);
        this.state = state;
        if (tc.isEntryEnabled()) Tr.exit(tc,"setState");
    }

    public synchronized String getStateObjectInfo() {
        return state;
    }
}
```

SampleStateAggregationHandler implementation

```
package com.ibm.ws390.sample;

import com.ibm.websphere.management.dynamicproxy.AggregationHandler;
import com.ibm.websphere.management.dynamicproxy.StateObject;

import com.ibm.ejs.ras.Tr;
import com.ibm.ejs.ras.TraceComponent;

public class SampleStateAggregationHandler implements AggregationHandler {

    private static TraceComponent tc =
    Tr.register(SampleStateAggregationHandler.class,"SampleState",null);

    /**
     * Return an aggregated result from a multicall Mbean operation which
     * compiles through all servant MBeans' results and returns a respective
     * single return value for an invoked method.
     *
     * @param  methodName           MBean method name
     * @param  params               MBean method parameters
     * @param  signatures           MBean method signatures
     * @param  servantMBeanResults  Result of each servant MBean instances
     *                              invoked by the dynamicproxy multicast
     *                              invocation.
     *                              Note: this value can be "null" OR can be
     *                              an array of "null"s in case return value
     *                              of the method is "void." Implementation
     *                              of this method MUST handle this case to
     *                              avoid a <code>NullPointerException</code>.
     * @param stateObject
     *        MBean provider provided <code>StateObject</code> used by
     *        dynamicproxy MBean in CR to manage its state. Note: this object
     *        MAY BE null if "stateObjectClass" was not specified OR internal
     *        error occurred during initialization of this dynamicproxy MBean.
     *        Implmentation MUST properly handle "null" input.
     *
     * @return aggregated result as defined by MBean xml for specified
     *         MBean operation.
     */
    public Object aggregateResults(String methodName,
                                   Object[] params,
                                   String[] signatures,
                                   Object[] servantMBeanResults,
                                   StateObject stateObject) {

        if (tc.isEntryEnabled()) Tr.entry(tc,"aggregateResults",methodName);

        // As you can see from the MBeanDescriptor of SampleStateMBean,
        // it declares the following four methods:
        // 1. String getMBeanName()         [proxyInvokeType == unicall]
        // 2. String getState()             [proxyInvokeType == multicall]
        // 3. void setState(String)         [proxyInvokeType == multicall]
        // 4. void initializeState()        [proxyInvokeType == multicall]
```

```
        //
        // Looking at the above methods, only method that requires aggregation
        // is #2 getState method which is a multicall MBean operation AND
        // it returns a value that can be aggregated.
        //
        // In this example, we simply take each servants' getState MBean
        // request result and concatenate them into one long String that
        // displays each servants' state.
        if (methodName.equals("getState")) {
            StringBuffer stateBuf = new StringBuffer();

            for (int i=0; i<servantMBeanResults.length; i++) {
                stateBuf.append("SERVANT #" + i + " state ==|" +
                                servantMBeanResults[i] + "|== ");
            }
            return stateBuf.toString();
        }
        // If we also had an example method which returns say an int,
        // getNumberOfMBeans(), it can take the similar approach
        // and to add each servants' getNumberOfMBeans() result together here.
        /* example added for non-existent method: int getNumberOfMBeans()
        else if (methodName.equals("getNumberOfMBeans")) {
            int aggregatedResult = 0;

            for (int i=0; i<servantMBeanResults.length; i++) {
                aggregatedResult += (int) servantMBeanResults[i];
            }
            return aggregatedResult;
        }
        */

        return methodName + " is NOT handled by " + getClass().getName() + "!";
    }

}
```

Administrative security

Access to the Java Management Extension (JMX) administrative subsystem requires role-based access control when administrative security is enabled.

Administrative security is also referred to as *administrative security*. A client, which can be a user or an administrative client program, can access an MBean method only if at least one of the required roles is granted to the client. WebSphere Application Server uses the declarative security approach to specify the security policy on the JMX MBean. This approach has the advantage of not requiring MBean developers to add security code. Moreover, WebSphere Application Server provides a default security policy for an MBean so in most case MBean developers do not need to specify a security policy at all. With WebSphere Application Server, you can define explicit security policy for your MBeans if the default security policy does not meet your specific security requirements.

Default MBean security policy

This topic discusses the default managed bean (MBean) security policy. In most cases, MBean developers do not need to specify a security policy.

Three types of MBeans exist for the default MBean security policy:

- A configuration type MBean
- A runtime type MBean
- A deployer type MBean

An optional attribute in the MBean descriptor XML file defines the type of MBean.

The ConfigRepository MBean is an example of one of a few configuration types. In the `configRepository.xml` descriptor file, the configureMBean = ″true″ attribute indicates that the MBean is a configuration type.

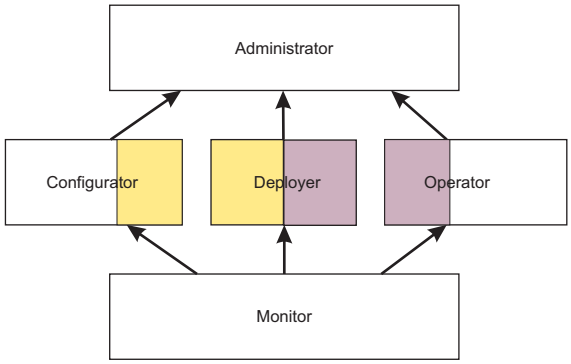
```
<MBean type="ConfigRepository"
  version="5.0"
  platform="common"
  description="Management interface for the configuration repository."
  configureMBean="true">
```

The EJBModule MBean is an example of deployer type MBeans. In the `EJBModule.xml` descriptor file, the deployerMBean=″true″ attribute indicates that the MBean is a deployer type.

```
<MBean type="EJBModule" j2eeType="EJBModule"
  version="5.0"
  platform="dynamicproxy"
  resourceIdentifierKey="Application"
  resourceType="Application"
  deployerMBean="true"
  description="Management interface for the EJBModule component.">
```

WebSphere Application Server extended role-based access control supports role inheritance. Five administrative roles of administrator, configurator, operator, deployer, and monitor exist.The monitor role is the least privileged administrative role. Users that are granted the monitor role can view the WebSphere Application Server configuration and the run-time status, but cannot make any changes. The other administrative roles each have their own unique set of privileges as well as the same privileges as the monitor role.

The configurator role has permission to modify WebSphere Application Server configuration data. The operator role has permission to change the runtime state, such as the start and stop of administrative resources. A configurator role cannot change the runtime status and conversely an operator role cannot change the WebSphere Application Server configuration. The administrator role includes configurator and operator role, but has more permissions than the union of configurator role and operator role. The administrator role can additionally change the administrative security configuration. A simple picture shows the administrative role inheritance relationship. The deployer role is a combination of the configurator and operator roles for application management. The deployer role has both configurator and operator permission for applications. A diagram shows the administrative role inheritance relationship.



Each MBean method or operation is assigned an impact attribute with a value of either `INFO` or `ACTION`. Here are some examples:

- A get method has an impact value of `INFO` and a write method has an impact value of `ACTION`.
- In the ConfigRepository MBean, the extract method does not change the configuration data and has an impact value of `INFO`, while the modify method has an impact value of `ACTION`.
- In the NodeAgent MBean, which is an operator type of MBean, the terminate method has an impact value of `ACTION`.

A configuration MBean method that has an impact value of INFO requires the monitor role. A configuration MBean method that has an impact value of ACTION requires the configurator role. A deployer MBean method that has an impact value of INFO requires the monitor role. A deployer MBean method that has an impact value of ACTION requires the deployer role. Because all administrative roles are monitor roles, any administrative role can access configuration MBean methods and deployer MBean methods that have an impact value of INFO. The administrator role is a configurator role and has access to the configuration MBean methods that have an impact value of ACTION.

The default security policy for the configuration MBean is summarized in the following table:

Method impact	Monitor role	Operator role	Configurator role	Deployer role	Administrator role
INFO	X	X	X	X	X
ACTION			X		X

The default security policy for the operation MBean is summarized in the following table:

Method impact	Monitor role	Operator role	Configurator role	Deployer role	Administrator role
INFO	X	X	X	X	X
ACTION		X			X

The default security policy for the deployer MBean is summarized in the following table:

Method impact	Monitor role	Operator role	Configurator role	Deployer role	Administrator role
INFO	X	X	X	X	X
ACTION		X		X	X

If an MBean has both the configureMBean attribute and the deployerMBean attribute set to true, the required role for all actions is either configurator or monitor. No such MBean is presently defined in the system. Users with the deployer role cannot log in to the administrative console unless that user is also granted the administrator, configurator, operator, or monitor role for the entire cell.

Defining an explicit MBean security policy

You can explicitly define an MBean security policy for a particular MBean. Use this example to define an MBean security policy.

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

Perform the following tasks to define an explicit security policy.

1. Assume that you have an MBean defined by the MBean sample.xml descriptor file.
2. Specify the explicit security policy for that MBean in the sampleSecurity.xml file. The naming convention is that you must append ″Security″ to the MBean descriptor file name as the name of the MBean security descriptor file.
3. Place the security policy descriptor file at the same directory where the MBean security descriptor file is so that the MBean loader can find it. This directory is the typical location for the security policy descriptor file. If no MBean security descriptor file is present, the default MBean security policy is used.
4. Specify the MBean name of sample in the resource element resource-name field of the sampleSecurity.xml file so that the MBean policy loader can associate the MBean security policy with

the MBean. The MBean security descriptor definition is very similar to the security policy that is defined by the Java 2 Platform, Enterprise Edition (J2EE) deployment descriptor.

You now have an explicitly defined MBean security policy that you can run with an MBean.

The following example describes the MBean security descriptor file format for the `sampleSecurity.xml` file.

Line 2 specifies that an MBean security descriptor schema is defined by the `RolePermissionDescriptor.dtd` file, which is a document type definition (DTD) in WebSphere Application Server.

As shown on line 3, each MBean descriptor file contains a single role-permission element. The administrative security role hierarchy is defined in the security-role elements between line 9 and line 37. The administrative security role has an inheritance relationship.

As defined on line 14 through 21, the operator security role implies the monitor security role, which means that a user with the operator role has all the permissions of the monitor role. As defined between line 30 and line 38, an administrator security role implies both the configurator and operator security role. Every MBean security descriptor file typically has the same role relationship definition so that you can cut and paste this section to your MBean security descriptor file.

One or more method-permission elements are defined after the security-role element. Each method-permission element defines the required roles for one or more methods. Specify method parameters to avoid method name collision in case multiple methods have the same name.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE role-permission SYSTEM "RolePermissionDescriptor.dtd" >
3. <role-permission>
4.    <resource>
5.       <resource-name>sample</resource-name>
6.       <class-name>com.ibm.ws.security.descriptor.sample</class-name>
7.       <description>This is a sample for testing role permission descriptor.</description>
8.    </resource>
9.    <security-role>
10.      <role>
11.         <role-name>monitor</role-name>
12.      </role>
13.   </security-role>
14.   <security-role>
15.      <role>
16.         <role-name>operator</role-name>
17.         <imply>
18.            <role-name>monitor</role-name>
19.         </imply>
20.      </role>
21.   </security-role>
22.   <security-role>
23.      <role>
24.         <role-name>configurator</role-name>
25.         <imply>
26.            <role-name>monitor</role-name>
27.         </imply>
28.      /role>
29.   </security-role>
30.   <security-role>
31.      <role>
32.         <role-name>administrator</role-name>
33.         <imply>
34.            <role-name>operator</role-name>
35.            <role-name>configurator</role-name>
36.         </imply>
37.      </role>
38.   </security-role>
```

```
39.   <method-permission>
40.      <description>Sample method permission table</description>
41.      <role-name>operator</role-name>
42.      <method>
43.         <description>Sample operation</description>
44.         <resource-name>sample</resource-name>
45.         <method-name>stop</method-name>
46.      </method>
47.   </method-permission>
48.   <method-permission>
49.      <description>Sample method permission table</description>
50.      <role-name>operator</role-name>
51.      <method>
52.         <description>Sample operation</description>
53.         <resource-name>sample</resource-name>
54.         <method-name>start</method-name>
55.         <method-params>
56.            <method-param>java.lang.String</method-param>
57.            <method-param>java.lang.String</method-param>
58.         </method-params>
59.      </method>
60.   </method-permission>
61.   <method-permission>
62.      <description>Sample method permission table</description>
63.      <role-name>operator</role-name>
64.      <method>
65.         <description>Sample operation</description>
66.         <resource-name>sample</resource-name>
67.         <method-name>monitor</method-name>
68.         <method-params>
69.         </method-params>
70.      </method>
71.   </method-permission>
72.   <method-permission>
73.      <description>Sample method permission table</description>
74.      <role-name>configurator</role-name>
75.      <method>
76.         <description>Sample operation</description>
77.         <resource-name>sample</resource-name>
78.         <method-name>setValue</method-name>
79.         <method-params>
80.            <method-param>java.lang.Boolean</method-param>
81.         </method-params>
82.      </method>
83.   </method-permission>
84.   <method-permission>
85.      <description>Sample method permission table</description>
86.      <role-name>monitor</role-name>
87.      <method>
88.         <description>Sample operation</description>
89.         <resource-name>sample</resource-name>
90.         <method-name>getValue</method-name>
91.      </method>
92.   </method-permission>
93. </role-permission>
```

Specifying fine-grained MBean security in the MBean descriptor

To implement fine-grained administrative security, your code must identify the resource instance that the managed bean (MBean) represents and assign the user the required role for that instance of the resource. This topic discusses what to do to identify the resource and assign the required role. This topic also discusses how to make an MBean method run under a different user identity so that the method can access other resource instances. Lastly, this topic discusses how to check if an MBean method has access to a resource instance by using programmatic interfaces.

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

Perform the following task to ensure that an MBean or MBean method is protected. Identify the resource instance that the MBean or MBean method represents and assign required roles to access the MBean. Perform this task during the development of the MBean.

1. Determine the resource instance that the MBean represents and the required roles to invoke the MBean methods.

   Every MBean method has a default MBean security policy. When the MBean method uses the default security policy, the resource instance that the MBean represents is assumed to be the server in which the MBean runs. If an MBean or MBean method represents a resource instance other than the server on which it runs, perform the following steps:

   a. Identify the resource instance that the MBean represents.

      - If an MBean, such as the Server MBean, accesses and modifies the server in which the MBean runs, do not specify a security policy to verify that the user invoking the MBean is granted access to the server because the default security policy is in force. In most cases, you use an MBean to access and modify the server.
      - If an MBean that runs inside a server can access and modify resources that do not directly belong to the server, check if the user invoking the MBean is granted access to the instance of the resource before allowing the MBean method to run.

        In most cases, identify the resource instance by identifying the key-value pair in the object name of the MBean that represents the resource instance. The resourceIndentifierKey attribute defines the key.

        For example, you can use the EJBModule MBean to access an Enterprise JavaBeans (EJB) module within an application that runs inside the server. In this case, the object name of the EJBModule MBean contains a key-value pair. The key is `Application`. The value represents the resource instance that the EJBModule MBean tries to access. The user that invokes this MBean method is verified to make sure that access is granted to this instance of the application before allowing the MBean method to run.

        The following example shows how to describe the fine-grained administrative security for the EJBModule type of MBean in the MBean descriptor:

        ```
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
        <MBean type="EJBModule" j2eeType="EJBModule"
          version="5.0"
          platform="dynamicproxy"
          resourceIdentifierKey="Application"
          resourceType="Application"
          deployerMBean="true"
          description="Management interface for the EJBModule component.">
        ```

      - If you can determine the resource that the MBean accesses before the MBean is invoked, but you cannot use the MBean object name to determine the resource instance that the MBean accesses, use parameters that are passed to the MBean instead.

        Identify the MBean method parameter name with a parameter value that represents the resource instance. Mark the corresponding parameter metadata in the MBean descriptor as the resource identifier. To mark a parameter as the resource identifier, add the resourceType attribute. The attribute specifies the type resource that the parameter value contains. When the resourceType attribute is present for any MBean method parameter, the parameter value determines the resource instance that the MBean method represents.

        For example, one instance of the ApplicationManager MBean runs in each server. The same MBean can be used to start and stop all the applications in the server. The start and stop methods of this MBean each take the application name as a parameter. They use the parameter to determine the instance of the application that this MBean method tries to access.

The following example shows how to describe the fine-grained administrative security for this type of MBean in the MBean descriptor:

```
<operation
    description="Start Application"
    impact="ACTION" name="startApplication" role="operation"
    targetObjectType="objectReference" type="void" proxyInvokeType="spray">
    <signature>
     <parameter description="Application Name" resourceType="Application"
        name="applicationName" type="java.lang.String"/>
    </signature>
</operation>
```

- If the resource that an MBean accesses cannot be determined until the MBean is invoked, check if the user invoking the MBean is granted access to the instance of the resource by using application programming interfaces (APIs).

  Mark the MBean or MBean method as excluded from access checking in the MBean descriptor by using the excludeAccessCheck attribute. When an MBean is marked as excluded from access checking, all its methods are also excluded from access checking.

  For example, the ConfigService MBean that runs in the deployment manager is used to configure all the resources within a cell. Exclude this MBean from access checking before invoking the MBean methods. Check that the ConfigService MBean is granted access to the configuration resource when the MBean attempts to access the resource.

  The following example shows how to describe the fine-grained administrative security for the ConfigServices type of MBean in the MBean descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean
  version="5.0"
  platform="proxy"
    collaboratorClass="com.ibm.ws390.management.proxy.ConfigServiceManager"
  description="Config Service component provides service of
configuration related tasks on top of configuration repository service."
  type="ConfigService"
  excludeAccessCheck="true"
  configureMBean="true">
```

  Some statements are split on multiple lines for printing purposes.

  The following example shows how to invoke the MBean method logic to perform authorization checking programmatically:

```
// Get administration authorizer.
AdminAuthorizer aa = AdminAuthorizerFactory.getAdminAuthorizer();
// Set the role that is required for this operation.
String role = com.ibm.ws.security.util.Constants.CONFIG_ROLE;
// Set the resource name.
// cells/cellName is optional.
String resource = "/nodes/"+ nodeName + /servers/" + serverName;
// Check access
if ( aa != null && !aa.checkAccess(resource, role) )
   // Disallow access.
   else
      // Allow access.
```

b. Assign required roles for the MBean and MBean methods.

   The required roles are automatically assigned, based on the type of MBean and the impact of the MBean method, as described in the topic on the default MBean security policy.

2. Specify delegation mode.

   In some cases, after performing the initial access check, the MBean method might need to run under a different user identity so that it can access other resource instances. For example the syncNode operation in the CellSync MBean grants the user the operator role to the instance of the node being synchronized. The syncNode operation tries to access resources under the cell scope. The user might

not have access to open files under the cell directory. The MBean must run as `System` after the initial access check so that the operation completes without any access denied problems.

Set the runAs attribute to `System` to specify delegation mode for an MBean or MBean method. When you set the runAs attribute for an MBean, the value applies to all MBean methods for that MBean.

The following example shows how to describe fine-grained administrative security for the CellSync type of MBean in the MBean descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MBean SYSTEM "MbeanDescriptor.dtd">
<MBean type="CellSync"
  version="5.0.1"
  platform="common"
  runAs="System"
  description="Management interface for the configuration synchronization logic
performed at the central deployment manager for the cell.">
<operation
    description="Initiate a synchronization request for a given node" impact="ACTION"
    name="syncNode" role="operation" targetObjectType="objectReference" type="ja
va.lang.Boolean">
<signature>
<parameter resourceType="Node"
 description="The name of the node"
 name="nodeName" type="java.lang.String"/>
</signature>
</operation>
```

You have determined the type of resource a given MBean method is accessing and performed the necessary access check so that WebSphere Application Server can allow access to the resource.

Administrative programs for multiple Java 2 Platform, Enterprise Edition application servers

You can develop an administrative client to manage multiple vendor application servers through existing MBean support in the WebSphere Application Server.

Existence of MBeans for stopped components

best-practices: The WebSphere Application Server completely implements the Java 2 Platform, Enterprise Edition (J2EE) Management specification. However, some differences in details between the J2EE specification and the WebSphere Application Server implementation are important for you to understand when you access WebSphere Application Server components. These differences are important to you when you access application MBeans because you can use either the WebSphere Application Server programming model or the J2EE programming model.

In the WebSphere Application Server programming model, if an MBean exists, you can assume that it is running. If an MBean does not exist, you can assume that it is stopped. Transient states between the started state and the stopped state are the same as the stopped state, which means that no MBean exists.

In the J2EE programming model, the MBean always exists regardless of the state of the component.

You can determine the state of a component by querying the state attribute. However, the state attribute only exists for MBeans that are state manageable, meaning that they implement the StateManageable interface. State manageable MBeans have start(), startRecursive(), and stop() operations whether these MBeans are J2EE MBeans or WebSphere Application Server MBeans. Additionally, the WebSphere Application Server defines the stateful interface. The stateful interface means that the component has a

state and emits the j2ee.state.notifications method, but that the component cannot directly manage the state. For example, a Web module cannot stop itself. However, the application that contains the Web module can stop it.

Not all MBeans that have a state are state-manageable. Servlets, J2EE modules and enterprise beans, for example, are all stateful, but are not state manageable. The J2EE server is not state-manageable because no start() operation is available on a server.

The J2EEApplication MBean is an example of a state manageable MBean. When the WebSphere Application Server starts, each application activates a J2EEApplication MBean for itself. A J2EEApplication MBean has a J2EE type of J2EEApplication (for example, `ObjectName *:*,j2eeType=J2EEApplication`). If the application starts, it also activates an Application MBean with a type of Application (for example, `*:*,type=Application`). When the application changes state, the Application MBean is activated or deactivated. However, the J2EEApplication MBean is always activated. You can retrieve the application state changes by getting the state attribute.

The modules attribute on the J2EEApplication component returns an array of object names, one for every module in the application. The Application Server activates an MBean for each of these modules only after the Application Server starts the application. The managed enterprise bean isRegistered(ObjectName) method returns `false` if the application, and therefore the module, is not running.

All of the attributes that are defined in the J2EE management specification return valid values when the managed object stops. Other attributes and operations, for example those that are specifically defined for the Application Server, use the com.ibm.websphere.management.exception.ObjectNotRunningException exception if they are accessed when the object is stopped.

If you install the application while the server runs, the application installs the J2EEApplication MBean when the installation completes. Conversely, when the application uninstalls the J2EEApplication MBean, the application deactivates the MBean.

Mapping key properties

The following table lists the mapping from the J2EE management-defined j2eeType key property to the WebSphere Application Server type key property. You can use either key property to access MBeans. However, only use the j2eeType key property if you want to connect to application servers other than WebSphere Application Server.

j2eeType key property	**type key property**
J2EEDomain	J2EEDomain [new]
J2EEServer	Server
JVM	JVM
J2EEApplication	Application [separate MBean from j2eeType
WebModule	WebModule
ResourceAdapterModule	ResourceAdapterModule
EJBModule	EJBModule
EJB and subtypes / Servlet / ResourceAdapter	EJB and subtypes / Servlet / ResourceAdapter
JavaMailResource	MailProvider
JNDIResource	NameServer
JMSResource	JMSProvider
JTAResource	TransactionService
RMI_IIOPResource	ORB

URLResource	URLProvider
JDBCResource	JDBCProvider
JDBCDataSource	DataSource
JDBCDriver	JDBCDriver [new]
JCAResource	J2CResourceAdapter
JCAConnectionFactory	J2CConnectionFactory
JCAManagedConnectionFactory	J2CManagedConnectionFactory [new]

Optional WebSphere Application Server interfaces

The following table shows the optional J2EE management interfaces that WebSphere Application Server provides. Some j2eeType key properties are split on multiples lines for printing purproses.

j2eeType key property	EventProvider interface	StateManageable interface	StatisticsProvider interface
J2EEDomain	No	No	No
J2EEServer	Yes	Stateful	No
JVM	No	No	Yes
J2EEApplication	Yes	Yes	No
WebModule	Yes	Stateful	No
ResourceAdapterModule	Yes	Stateful	No
EJBModule	Yes	Stateful	No
AppClientModule	Yes	Stateful	No
EJB and subtypes / Servlet / ResourceAdapter	No	No	Yes
JavaMailResource	No	No	No
JNDIResource	No	No	No
JMSResource	No	No	No
JTAResource	Yes	No	Yes
RMI_IIOPResource	No	No	Yes
URLResource	No	No	No
JDBCResource	No	No	Yes
JDBCDataSource	No	No	No
JDBCDriver	No	No	No
JCAResource	Yes	No	Yes
JCAConnectionFactory	No	No	No
`JCAManagedConnection Factory`	No	No	No

Deploying and managing a custom Java administrative client program with multiple Java 2 Platform, Enterprise Edition application servers

This section describes how to connect to a Java 2 Platform, Enterprise Edition (J2EE) server, and how to manage multiple vendor servers.

The WebSphere Application Server completely implements the J2EE Management specification, also known as JSR-77 (Java Specification Requests 77). However, some differences in details between the J2EE specification and the WebSphere Application Server implementation are important for you to understand when you develop a Java administrative client program to manage multiple vendor servers. For information, see the Java 2 Platform, Enterprise Edition (J2EE) Management Specification and the MBean Java application programming interface (API) documentation.

When your administrative client program accesses WebSphere Application Servers exclusively, you can use the Java APIs and WebSphere Application Server-defined MBeans to manage them. If your program needs to access both WebSphere Application Servers and other J2EE servers, use the API defined in the J2EE Management specification.

1. Connect to a J2EE server.

   Connect to a server by looking up the Management enterprise bean from the Java Naming and Directory Interface (JNDI). The Management enterprise bean supplies a remote interface to the MBean server that runs in the application server. The Management enterprise bean works almost exactly like the WebSphere Application Server administrative client, except that it does not provide WebSphere Application Server specific functionality. The following example shows how to look up the Management enterprise bean.

   ```
   import javax.management.j2ee.ManagementHome;
   import javax.management.j2ee.Management;

   Properties props = new Properties();

   props.setProperty(Context.PROVIDER_URL, "iiop://myhost:2809");
   Context ic = new InitialContext(props);
   Object obj = ic.lookup("ejb/mgmt/MEJB");
   ManagementHome mejbHome = (ManagementHome)
           PortableRemoteObject.narrow(obj, ManagementHome.class);
   Management mejb = mejbHome.create();
   ```

   The example gets an initial context to an application server by passing the host and port of the Remote Method Invocation (RMI) connector. You must explicitly code the RMI port, in this case 2809. The lookup method looks up the `ejb/mgmt/MEJB` path, which is the location of the Management enterprise bean home. The example then creates the mejb stateless session bean, which you use in the next step.

2. Manage multiple vendor application servers.

   After you create the mejb stateless session bean, you can use it to manage your application servers. Components from the application servers appear as MBeans, which the specification defines. These MBeans all have the j2eeType key property. This key property is one of a set of types that the specification defines. All of these types have a set of exposed attributes.

   Use the following example to guide you in managing multiple vendor application servers. The example uses the Java virtual machine (JVM) MBean to determine what the current heap size is for the application server.

   ```
   ObjectName jvmQuery = new ObjectName("*:j2eeType=JVM,*");
   Set s = mejb.queryNames(jvmQuery, null);
   ObjectName jvmMBean = (ObjectName) s.iterator().next();
   boolean hasStats = ((Boolean) mejb.getAttribute(jvmMBean,
           "statisticsProvider")).booleanValue();
   if (hasStats) {
       JVMStats stats = (JVMStats) mejb.getAttribute(jvmMBean,
                                                     "stats");
       String[] statisticNames = stats.getStatisticNames();
       if (Arrays.asList(statisticNames).contains("heapSize")) {
           System.out.println("Heap size: " + stats.getHeapSize());
       }
   }
   ```

The queryNames() method first queries the JVM MBean. The getAttribute method gets the statisticsProvider attribute and determine if this MBean provides statistics. If the MBean does, the example accesses the stats attribute, and then invokes the getHeapSize() method to get the heap size.

The strength of this example is that the example can run on any vendor application server. It demonstrates that an MBean can optionally implement defined interfaces, in this case the StatisticsProvider interface. If an MBean implements the StatisticsProvider interface, you can see if an application server supports a particular statistic, in this case the heap size. The specification defines the heap size, although this value is optional. If the application server supports the heap size, you can display the heap size for the JVM.

Java Management Extensions V1.0 to Java Management Extensions V1.2 migration

You might need to migrate custom MBeans that are supplied by products other than the Application Server from Version 5 to Version 6.x for full compatibility.

Each Java virtual machine (JVM) in WebSphere Application Server includes an embedded implementation of Java Management Extensions (JMX). In Application Server, Version 5, the JVMs contain an implementation of the JMX 1.0 specification. In Application Server, Version 6, the JVMs contain an implementation of the JMX 1.2 specification. The JMX 1.0 implementation used in Version 5 is the TMX4J package that IBM Tivoli products supply. The JMX 1.2 specification used in Version 6 is the open source mx4j package. The JMX implementation change across the releases does not affect the behavior of the JMX MBeans in the Application Server. No Application Server administrative application programming interfaces (APIs) are altered due to the change from the JMX V1.0 specification to the JMX V1.2 specification.

The JMX V1.2 specification is compatible with the earlier JMX V1.0 specification. However, you might need to migrate custom MBeans that are supplied by products other than the Application Server from Version 5 to Version 6. The primary concern for these custom MBeans is related to the values that are used in key properties of the JMX ObjectName class for the MBean. The open source mx4j implementation more stringently enforces property validation according to the JMX 1.2 specification. Test the custom MBeans that you deployed in Version 5 in Version 6.x, to ensure compatibility. Full details of the JMX V1.2 specification changes from the JMX V1.0 specification are available in the JMX 1.2 specification.

Java Management Extensions interoperability

Starting with Version 6, WebSphere Application Server implements Java Management Extensions (JMX) Version 1.2, while WebSphere Application Server Version 5 implements JMX Version 1.0.

Differences between Version 5 and Version 6

Due to the evolution of the JMX specification, the serialization format for JMX objects, such as the javax.management.ObjectName object, differs between the V5 implementation and the V6 and later implementation. The V6 and later JMX run time is enhanced to be aware of the version of the client with which it is communicating. The V6 and later run time makes appropriate transformations on these incompatible serialized formats to support communication between the different version run times.

A V5 wsadmin script or a V5 administrative client can call a V6 and later deployment manager, node, or server. A V6 and later wsadmin script or a V6 and later administrative client can call a V5 node or server.

When a V5 wsadmin script or a V5 administrative client calls a V6 and later MBean, the instances of classes that are new in V6 and later cannot be passed back to V5 because these classes are not present

in the V5 environment. The problem occurs infrequently. However, it usually occurs when an exception embeds a nested exception that is new starting with V6. The symptom is usually a serialization exception or a NoClassDefFoundException exception.

Due to changes in the JMX implementation from V5 to V6, different exceptions are created when a method on an MBean is invoked for V5 than when a method on an MBean is invoked for V6 and later . For example, when a method gets or sets an unknown attribute for V5, the MBeanRuntimeException exception is created. When a method gets or sets an unknown attribute for V6 and later, the MBeanException exception that wraps a ServiceNotFoundException exception is created.

An instance of a user-defined class that implements the Serializable interface that is passed as a parameter or return value during MBean invocation, or sent as part of a notification, cannot contain a non-transient instance variable that is in the javax.management.package package. If the instance does, it cannot be properly deserialized when passed between V5 and V6 or later run times.

Due to changes in the supported format for the ObjectName class from V5 to V6, the configuration ID in starting with V6 contains a vertical bar (|), whereas in V5, the ID contains a colon (:). This change is reflected in the output for wsadmin clients. For example, for a V5 client, the output is:

```
wsadmin>  $AdminConfig list Cell
     DefaultCellNetwork(cells/DefaultCellNetwork:cell.xml#Cell_1)
```

whereas for a V6 and later client, the output is:

```
wsadmin>  $AdminConfig list Cell
     DefaultCellNetwork(cells/DefaultCellNetwork|cell.xml#Cell_1)
```

The change to the configuration ID generally is not a problem because configuration IDs are generated dynamically. When a V5 client passes a configuration ID that contains a colon, the JMX run time, for upward compatibility, automatically transforms the configuration ID that contains a colon into a configuration ID that contains a vertical bar. Similarly, a reverse transformation is performed for backward compatibility.

Do not save the configuration ID and then try to use it later. Only query the ID and use it.

Differences between Version 6.1 and Version 6.0.x

A serialization format mismatch exists between the JMX implementation in V6.1 and V6.0.x releases. When a V6.0.x wsadmin script or a V6.0.x administrative client tries to retrieve the ModelMBeanInfo interface of a V6.1 MBean, the expected field names are not found in the deserialized object because of the case difference between the versions. For example, the following wsadmin function does not work when a V6.0.x wsadmin script connects to a V6.1 server:

```
$Help attributes MBeanObjectName
$Help operations MBeanObjectName
$Help all MBeanObjectName
```

where MBeanObjectName is a string representation of a V6.1 MBean Object.

To avoid this problem, set the jmx.serial.form Java virtual machine (JVM) custom property on the JVM custom properties page in the administrative console. Create the custom property by specifying the name value-pair on the V6.1 Application Server that you are connecting to from a V6.0.x client. The field names are forced to lower case to be compatible with what the V6.0.x client expects. The lower case field names contradict the JMX specification and compromise interoperability with future versions. Therefore, the recommendation is that you set this property only when it is absolutely needed in a mixed version environment.

Property name	jmx.serial.form
Data type	string

Value 1.2.0 or 1.2.1

To access the JVM custom properties page, click:

Servers > Application Servers > *server1* **> Java and Process Management > Process Definition > Control > Java Virtual Machine > Custom Properties**

Managed object metadata

Information about a node, such as operating system platform and product features, is maintained in the configuration repository in the form of properties. As product features are installed on a node, new property settings are added.

WebSphere Application Server system management uses the managed object metadata properties as follows:

- To display the node version in the administrative console
- To ensure that new configuration types or attributes are not created or set on older release nodes
- To ensure that new resource types are not created on old release nodes
- To ensure that new applications are not installed on old release nodes because the old run time cannot support the new applications

Base properties

The following base property keys are defined for WebSphere Application Server:

com.ibm.websphere.baseProductVersion: The version of WebSphere Application Server that is installed.

com.ibm.websphere.nodeOperatingSystem: The operating system platform on which the node runs.

com.ibm.websphere.nodeSysplexName: The sysplex name on a z/OS operating system. This property applies to the z/OS operating system only.

com.ibm.websphere.deployed.features: A list of features that extends a profile. An example of a feature is an administrative console plug-in.

Here are examples of metadata property values. The last item is split on multiple lines for printing purposes.

```
com.ibm.websphere.baseProductVersion=6.0.0.0
com.ibm.websphere.nodeOperatingSystem=os390
com.ibm.websphere.nodeSysplexName=PLEX1
com.ibm.websphere.deployed.features=
 com.ibm.ws.base_6.0.0.0,com.ibm.ws.j2ee_6.0.0.0,
 com.ibm.ws.uddi_6.0.0.0,com.ibm.ws.wsgateway_6.0.0.0
```

For detailed information on metadata properties, view the ManagedObjectMetadataHelper class in the Application Server API documentation.

Accessing managed object metadata properties

An administrator can query managed object metadata through the wsadmin tool or Application Server APIs. They can additionally be viewed on the Node Installation properties administrative console panel. This article provides details on the Application Server API method.

An accessor class is used to obtain the managed object metadata properties. An accessor instance is created through its factory. A helper class, which uses the accessor instance, makes it easy to query the

base metadata properties. These classes are all part of the com.ibm.websphere.management.metadata package in the Application Server API documentation. The specific names of these classes are:

- com.ibm.websphere.management.metadata.ManagedObjectMetadataHelper
- com.ibm.websphere.management.metadata.ManagedObjectMetadataAccessor
- com.ibm.websphere.management.metadata.ManagedObjectMetadataAccessorFactory

Managing applications through programming

Through Java MBean programming, you can install, update, and delete a Java 2 Platform, Enterprise Edition (J2EE) application on WebSphere Application Server.

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

For information on the restarting of updated applications, refer to Fine-grained recycle behavior in *IBM WebSphere Developer Technical Journal: System management for WebSphere Application Server V6 -- Part 5 Flexible options for updating deployed applications*.

Before you can install or change an application on WebSphere Application Server, you must first create or update your application and assemble it using an assembly tool.

Besides installing, uninstalling, and updating applications through programming, you can additionally install, uninstall, and update J2EE applications through the administrative console or the wsadmin tool. All three ways provide identical updating capabilities.

1. Perform any or all of the following tasks to manage your J2EE applications through programming.
   - Access the application management function.

     This topic provides examples to access the application management functionality:
     - From WebSphere Application Server code
     - From outside WebSphere Application Server
     - When WebSphere Application Server is not running
   - Install an application.

     This topic provides an example for initially installing an application on WebSphere Application Server.
   - Uninstall an application.

     This topic provides an example for uninstalling an application that resides on WebSphere Application Server.
   - Manipulate additional attributes for a deployed application.

     This topic provides an example for manipulating attributes that are not exposed through the AppDeploymentTask object.
   - Share sessions for application management.

     This topic provides an example for saving application-specific updates for a deployed application to a session, and then to the configuration repository.
   - Update an application.

     This topic provides an example for updating the installed application on WebSphere Application Server with a new application. When you completely update an application, the deployed application is uninstalled and the new enterprise archive (EAR) file is installed.
   - Add to, update, or delete part of an application.

     This topic provides an example that you can use to add, update, or delete part of an application on WebSphere Application Server.
   - Edit an application.

This topic provides an example that you can use to edit an application on WebSphere Application Server.

- Add a module.

  This topic provides an example for adding a module to an application that resides on WebSphere Application Server.
- Update a module.

  This topic provides an example for updating a module that resides on WebSphere Application Server. When you update a module, the deployed module is uninstalled and the updated module is installed.
- Delete a module.

  This topic provides an example for deleting a module that resides on WebSphere Application Server. When you delete a module, the deployed module is uninstalled.
- Add a file.

  This topic provides an example for adding a file to an application that resides on WebSphere Application Server.
- Update a file.

  This topic provides an example for updating a file on WebSphere Application Server. When you update a file, the deployed file is uninstalled and the updated file is installed.
- Delete a file.

  This topic provides an example for deleting a file on WebSphere Application Server. When you delete a file, the deployed file is uninstalled.

2. Save your changes to the master configuration repository.
3. Synchronize changes to the master configuration across the nodes for the changes to take effect.

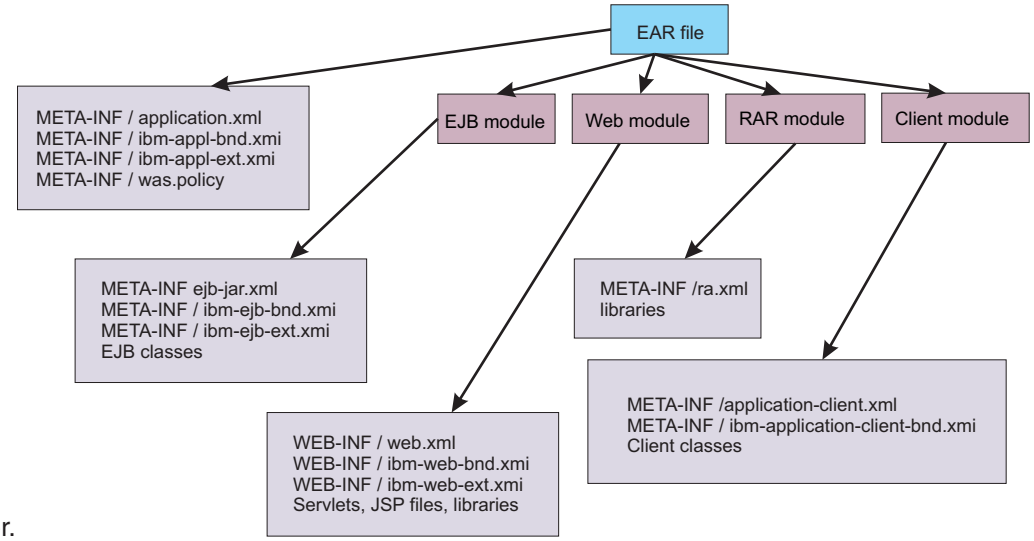
If you have further application updates, you can do the updates through programming, the administrative console, or the wsadmin tool.

Application management

Java 2 Platform, Enterprise Edition (J2EE) applications and modules include an Extensible Markup Language (XML)-based deployment descriptor that specifies various J2EE artifacts that pertain to applications or modules. The J2EE artifacts include Enterprise JavaBeans (EJB) definitions, security role definitions, EJB references, resource references, and so on. These artifacts define various unresolved references that the application logic uses. The J2EE specification requires that these artifacts map to J2EE platform-specific information, such as that found in WebSphere Application Server, during deployment of J2EE applications.

The application assembly tools that WebSphere Application Server supports, as well as the application management support that is provided with the product, facilitate collection of certain WebSphere Application Server information. The collected information is used to resolve references that are defined in various deployment descriptors in a J2EE application. This information is stored in the application EAR file in conjunction with the deployment descriptors. The following diagram shows the structure of an Enterprise

Archive (EAR) file that is populated with deployment information that is specific to WebSphere Application Server.



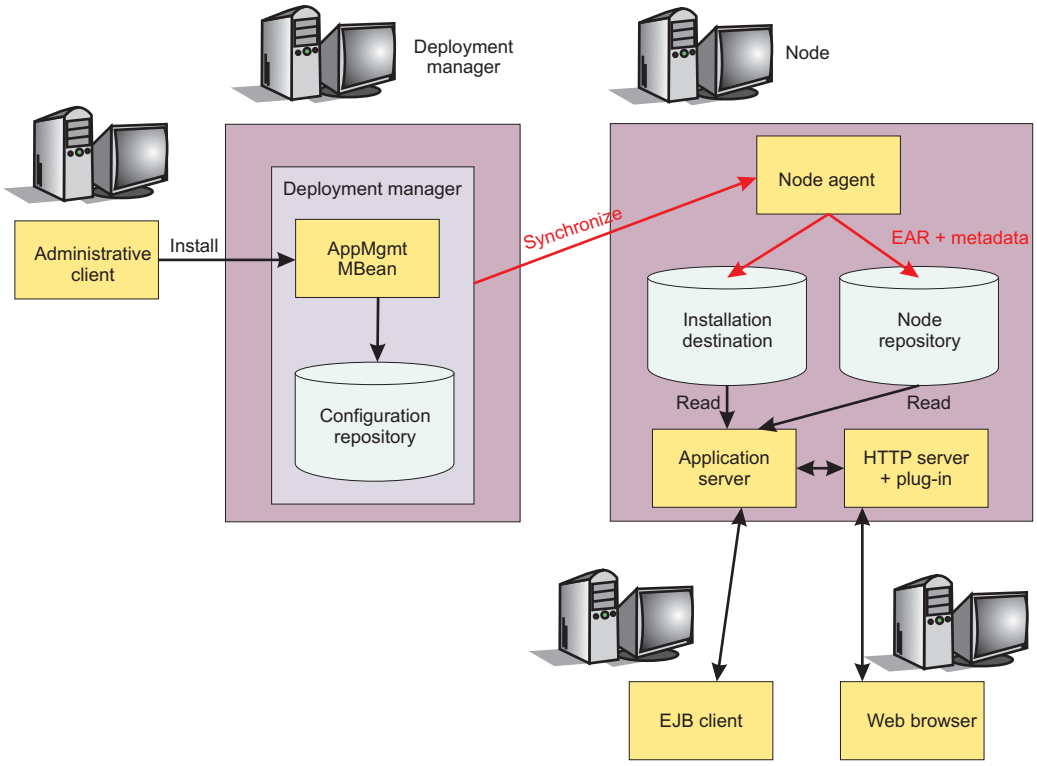
The application management architecture provides a set of classes with which deployers can collect WebSphere Application Server deployment information. This information is also referred to as *binding information*, and is stored in the application EAR file. The deployer can install the EAR file into a WebSphere Application Server configuration by using the AppManagement interface.

The application management support in WebSphere Application Server provides functions such as installing and uninstalling applications, editing binding information for installed applications, updating the entire application or part of the application, exporting the application, and so on. The com.ibm.websphere.management.application.AppManagement interface, which is exposed as a Java Management Extensions (JMX)-based AppManagement MBean in WebSphere Application Server, provides this functionality. Code that runs on the server or in a stand-alone administrative client program can access the interface. Access to the application management functions is also possible in the absence of WebSphere Application Server. This mode, known as *local mode*, is particularly useful for installing J2EE applications as part of product installation. For WebSphere Application Server Network Deployment, the AppManagement MBean is present in the deployment manager only, which facilitates centralized configuration and administration.

Application deployment in a Network Deployment configuration

The following diagram explains application deployment for the Network Deployment product:

1. The application EAR file that you install is stored in the master configuration repository of the deployment manager, with the application metadata.
2. The configuration synchronization operation replicates the application in the master repository into the repository of the target node.
3. The application EAR file is extracted to the installation destination of the target node at the end of the synchronization operation.
4. The WebSphere Application Server run time reads the application business logic such as EJB classes, servlets, and JavaServer Pages (JSP) from the installation destination while serving application client requests.
5. WebSphere Application Server reads the application metadata, such as deployment descriptors and WebSphere Application Server bindings during application startup from the configuration repository or the installation destination only, depending on the option that is specified during application installation.



Accessing the application management function

The com.ibm.websphere.management.application.AppManagementProxy class provides uniform access to application management functionality, regardless of whether the functionality is accessed from the server process, administrative client process, or a stand-alone Java program in the absence of WebSphere Application Server. This topic provides code excerpts that demonstrate how to obtain an AppManagementProxy instance in a variety of cases.

This task assumes a basic familiarity with WebSphere Application Server programming interfaces and MBean programming. For information on WebSphere Application Server programming interfaces, see application programming interface (API) documentation. For information on MBean programming, see MBean Java application programming interface (API) documentation.

Perform any of the following tasks to access application management functionality through programming.

- To access application management functionality from WebSphere Application Server code, for example, as a custom service, create the AppManagementProxy class.

```
AppManagement appMgmt =
             AppManagementProxy. getJMXProxyForServer();
```

- To access application management functionality from outside WebSphere Application Server through the AppManagement MBean, create an administrative client to establish a connection to WebSphere Application Server and then create the AppManagementProxy class.

```
AdminClient adminClient = ....

// create AppManagement proxy object
AppManagement appMgmt =  AppManagementProxy. getJMXProxyForClient (adminClient);
```

- To access application management functionality when WebSphere Application Server is not running (local mode), create the AppManagementProxy class.

```
AppManagement appMgmt = AppManagementProxy. getLocalProxy ();
```

  – When running in local mode set the com.ibm.ws.management.standalone system property to `true`. If you want to modify configuration documents in a non-default location, set the location of the configuration directory through the was.repository.root system property.

– Although you can use application management functions with or without WebSphere Application Server running, do not access application management functions concurrently through local mode and the AppManagement MBean. Otherwise, updates that are made using these modes can collide and break the integrity of the WebSphere Application Server configuration.

After you successfully create the AppManagementProxy class, you have access to application management functionality.

You can perform various management tasks such as installing, uninstalling, editing, and so on.

Installing an application through programming

You can install an application through the administrative console, the wsadmin tool, or programming. Use this example to install an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can install an application on WebSphere Application Server, you must first create or update your application and assemble it using an assembly tool.

Perform the following tasks to install an application through programming.

1. Populate the enterprise archive (EAR) file with WebSphere Application Server-specific binding information.
   a. Create the controller and populate the EAR file with appropriate options.
   b. Optionally run the default binding generator.
   c. Save and close the EAR file.
   d. Retrieve the saved options table that will be passed to the installApplication MBean (API).
2. Connect to WebSphere Application Server.
3. Create the application management proxy.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the installApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.

      Refer to the com.ibm.websphere.management.application.AppManagement class in the Application Server API documentation to understand various options that can be passed to the installApplication MBean API.
5. Create the notification filter for listening to installation events.
6. Add the listener.
7. Install the application.
8. Wait for some timeout so that the program does not end.
9. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
10. When the installation is done, remove the listener and quit.

After you successfully run the code, the application is installed.

The following example shows how to install an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
```

```
import com.ibm.websphere.management.*;

import javax.management.*;

public class Install {

    public static void main (String [] args) {

        try {
  String earFile = "C:/test/test.ear";
  String appName = "MyApp";

// Preparation phase: Begin
// Through the preparation phase you populate the enterprise archive (EAR) file with
// WebSphere Application Server-specific binding information. For example, you can specify
// Java Naming and Directory Interface (JNDI) names for enterprise beans, or virtual hosts
// for Web modules, and so on.

// First, create the controller and populate the EAR file with the appropriate options.
  Hashtable prefs = new Hashtable();
  prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());

// You can optionally run the default binding generator by using the following options.
// Refer to Java documentation for the AppDeploymentController class to see all the
// options that you can set.
  Properties defaultBnd = new Properties();
  prefs.put (AppConstants.APPDEPL_DFLTBNDG, defaultBnd);
  defaultBnd.put (AppConstants.APPDEPL_DFLTBNDG_VHOST, "default_host");

// Create the controller.
  AppDeploymentController controller = AppDeploymentController
    .readArchive(earFile, prefs);
  AppDeploymentTask task = controller.getFirstTask();
  while (task != null)
  {
// Populate the task data.
   String[][] data = task.getTaskData();
// Manipulate task data which is a table of stringtask.
   setTaskData (data);
   task = controller.getNextTask();
  }
  controller.saveAndClose();

  Hashtable options = controller.getAppDeploymentSavedResults();
// The previous options table contains the module-to-server relationship if it was set by
// using tasks.
//Preparation phase: End

// Get a connection to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

// Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

// If code for the preparation phase has been run, then you already have the options table.
// If not, create a new table and add the module-to-server relationship to it by uncommenting
// the next statement.
//Hashtable options = new Hashtable();
```

```
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());

// Uncomment the following statements to add the module to the server relationship table if
//  the preparation phase does not collect it.
//Hashtable module2server = new Hashtable();
//module2server.put ("*", target);
//options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, module2server);

//Create the notification filter for listening to installation events.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (AppConstants.NotificationType);


//Add the listener.
  NotificationListener listener = new AListener(_soapClient,
myFilter, "Install: " + appName, AppNotification.INSTALL);

// Install the application.
  proxy.installApplication (earFile, appName, options, null);
  System.out.println ("After install App is called..");

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit.
```

```
            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Once you install the application, you must explicitly start the application or you must stop and restart the server. For information on starting an application, see the Starting an application through programming topic in the *Using the administrative clients* PDF. For information on stopping or restarting the server, see the Stopping an application server topic or the Starting an application server topic, respectively, in the *Setting up the application serving environment* PDF.

Starting an application through programming

You can start an application through the administrative console, the wsadmin tool, or programming. Use this example to start an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can start an application on WebSphere Application Server, you must first install your application.

Perform the following tasks to start an application through programming.

1. Connect the administrative client to WebSphere Application Server.
2. Create the application management proxy.
3. Call the startApplication method on the proxy by passing the application name and optionally the list of targets on which to start the application.

After you successfully run the code, the application is started.

The following example shows how to start an application following the previously listed steps. Some statements are split on multiple lines for printing purposes.

```
//Do a get of the administrative client to connect to
//WebSphere Application Server.

AdminClient client = ...;
String appName = "myApp";
Hashtable prefs = new Hashtable();
// Use the AppManagement MBean to start and stop applications on all or some targets.
// The AppManagement MBean is on the deployment manager in the Network Deployment product.

// Query and get the AppManagement MBean.
ObjectName on = new ObjectName ("WebSphere:type=AppManagement,*");
Iterator iter = client.queryNames (on, null).iterator();
ObjectName appmgmtON = (ObjectName)iter.next();

//Start the application on all targets.
AppManagement proxy = AppManagementProxy.getJMXProxyForClient(client);
String started = proxy.startApplication(appName, prefs, null);
System.out.println("Application started on folloing servers: " + started);
```

```
//Start the application on some targets.
//String targets = "WebSphere:cell=cellname,node=nodename,
server=servername+WebSphere:cell=cellname,cluster=clusterName";
//String started1 = proxy.startApplication(appName, targets, prefs, null);
//System.out.println("Application started on following servers: " + started1)
```

Uninstalling an application through programming

You can uninstall an application through the administrative console, the wsadmin tool, or programming. Use this example to uninstall an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can uninstall an application on WebSphere Application Server, you must first install it.

Perform the following tasks to uninstall an application through programming.

1. Get a connection to WebSphere Application Server.
2. Get the application management proxy.
3. Create the notification filter for listening to uninstallation events.
4. Add the listener.
5. Uninstall the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the uninstallation is done, remove the listener and quit.

After you successfully run the code, the application is uninstalled.

The following example shows how to uninstall an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class Uninstall {

    public static void main (String [] args) {

        try {

// Get a connection to the server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Get the application management proxy.
```

```
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());


  //Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (AppConstants.NotificationType);


  //Add the listener.
  NotificationListener listener = new AListener(_soapClient,
myFilter, "Install: " + appName, AppNotification.UNINSTALL);

  // Uninstall the application.
  proxy.uninstallApplication (appName, options, null);
  System.out.println ("After uninstall App is called..");

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the unistallation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
```

```
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Manipulating additional attributes for a deployed application

You can manipulate attributes for a deployed application through the administrative console, the wsadmin tool, or by programming. Use this example to manipulate attributes that are not exposed during or after application installation through the AppDeploymentTask object.

This task assumes a basic familiarity with MBean programming and the ConfigService interfaces. For information on MBean programming, see MBean Java application programming interface (API) documentation. For information on ConfigService interfaces, see Java application programming interface (API) .

Before you can manipulate additional attributes for your deployed application on WebSphere Application Server, you must first install your application.

Perform the following tasks for your deployed application to manipulate attributes that are not exposed through the AppDeploymentTask object. The attributes are saved in the `deployment.xml` file that is created in the configuration repository for each deployed application.

1. Create a session.
2. Connect to WebSphere Application Server.
3. Locate the ApplicationDeployment object.
4. Manipulate the attributes.
5. Save your changes.
6. Clean up the session.

After you successfully run the code, the attributes are updated in the `deployment.xml` file for the deployed application.

The following example shows how to manipulate the startingWeight, warClassLoaderPolicy, and classloader attributes based on the previous steps.

```
import java.util.Properties;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.ObjectName;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.configservice.ConfigService;
import com.ibm.websphere.management.configservice.ConfigServiceHelper;
import com.ibm.websphere.management.configservice.ConfigServiceProxy;
import com.ibm.websphere.management.exception.ConfigServiceException;
import com.ibm.websphere.management.exception.ConnectorException;
```

```
public class AppManagementSample1 {

 public static void main(String[] args) {
  String hostName = "localhost";
  String port = "8880";
  String appName = "ivtApp";

  ConfigService configService = null;

  // create a session.
  Session session = new Session();

  // establish connection to the server.
  try {
   Properties props = new Properties();
   props.setProperty(AdminClient.CONNECTOR_TYPE,
     AdminClient.CONNECTOR_TYPE_SOAP);
   props.setProperty(AdminClient.CONNECTOR_HOST, hostName);
   props.setProperty(AdminClient.CONNECTOR_PORT, port);
   AdminClient adminClient =
    AdminClientFactory.createAdminClient(props);

   // create a config service proxy object.
   configService = new ConfigServiceProxy(adminClient);

   // Locate the application object.
   ObjectName rootID = configService.resolve(session,
     "Deployment="+appName)[0];
   System.out.println ("rootID is: " + rootID);

   // Locate the ApplicationDeployment object from the root.
   ObjectName appDeplPattern = ConfigServiceHelper.createObjectName
      (null, "ApplicationDeployment");
   /*
   ObjectName appDeplID = configService.queryConfigObjects(session,
     rootID, appDeplPattern, null)[0];
   */
   AttributeList list1 = configService.getAttributes(session,
     rootID, new String[]{"deployedObject"}, false);
   ObjectName appDeplID = (ObjectName)
    ConfigServiceHelper.getAttributeValue(list1, "deployedObject");
   System.out.println ("appDeplID: " + appDeplID);

   // Locate the class loader.

   // Change the starting weight through the startingWeight attribute. The starting weight
       // affects the order in which applications start.
   AttributeList attrList = new AttributeList();
   Integer newWeight = new Integer (10);
   attrList.add(new Attribute("startingWeight", newWeight));

   // Change the WAR class loader policy through the warClassLoaderPolicy attribute by
       // specifying SINGLE or MULTIPLE.
   // SINGLE=one classloader for all WAR modules
   attrList.add(new Attribute("warClassLoaderPolicy", "SINGLE"));

   // Set the class loader mode to PARENT_FIRST or PARENT_LAST.
   AttributeList clList = (AttributeList) configService.getAttribute
    (session, appDeplID, "classloader");
   ConfigServiceHelper.setAttributeValue (clList, "mode",
    "PARENT_LAST");
   attrList.add (new Attribute ("classloader", clList));

   // Set the new values.
   configService.setAttributes(session,  appDeplID, attrList);

   // Save your changes.
```

```
   configService.save(session, false);

  } catch (Exception ex) {
   ex.printStackTrace();
  } finally {
            // Clean up the session.
   try {
    configService.discard(session);
   }
   catch (ConfigServiceException csEx)
   {
    csEx.printStackTrace();
   }
   catch (ConnectorException cnEx)
   {
    cnEx.printStackTrace();
   }
  }
 }
}
```

Sharing sessions for application management

With the configuration service interface, ConfigService, you can create a session that is a temporary staging area, where you can save all the configuration modifications. Saving the session saves all the updates from the session into the WebSphere Application Server configuration repository. The application management logic supports session sharing with the configuration service. You can perform all the application management functions in the same session as the one that the configuration service creates. Saving such a session saves all the updates, including the ones that are application-specific.

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

Perform the following tasks for your deployed application to share and save application-specific updates through the configService configuration service.

1. Create a configuration service proxy object.
2. Create a session.
3. Pass the session information to the AppManagement MBean.

   Every method on the AppMangement interface takes session ID (workspace ID) as the last parameter. If the session information is passed in this parameter, the application management function uses the session. If you set the parameter to a null value:

   - No session sharing occurs
   - The configuration changes are always saved in the configuration repository if the operation succeeds.
4. Save the session after all the necessary changes are made.

   The following example outlines the general steps for session sharing through the configService configuration service. For a detailed example, see “Manipulating additional attributes for a deployed application” on page 903.

```
public void installApplication (String localEarPath,
            String appName, Hashtable properties, String workspaceID)
            throws AdminException;

AdminClient adminClient = ....;

// Create a configuration service proxy object.
ConfigService configService = new ConfigServiceProxy(adminClient);

// Create a session.
 Session session = new Session();
```

```
// Pass the session information to AppManagement MBean.
appMgmt = ...
appMgmt.installApplication
              (earPath, appName, properties, session.toString());
//Save the session after all necessary changes are made.
configService.save(session, false);
```

After you successfully complete the steps, you have saved application-specific updates for a deployed application to a session, and then to the configuration repository.

Updating an application through programming

You can update an existing application through the administrative console, the wsadmin tool, or programming. Use this example to completely update an application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update an application on WebSphere Application Server, you must first install your application.

Perform the following tasks to completely update an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Prepare the enterprise archive (EAR) file by populating it with binding information.
6. Update the application.
7. Wait for some timeout so that the program does not end.
8. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
9. When the update is done, remove the listener and quit.

After you successfully run the code, the application is updated.

The following example shows how to update an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class aa {

    public static void main (String [] args) {

        try {

  // Connect to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
```

```
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  String fileContents = "C:/test/test.ear";

  // Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.INSTALL);

// Refer to the installation example to see how you can prepare the enterprise archive (EAR)
// file by populating it with binding information.
// If code for the preparation phase has started, then you already have the options table.
// If not, create a new table and add the module-to-server relationship to it by uncommenting
// the next statement.
//Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
  options.put ((AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_APP);

// Uncomment the following statements to add the module to the server relationship table if
//  the preparation phase does not collect it
//Hashtable module2server = new Hashtable();
//module2server.put ("*", target);
//options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, module2server);
// Update the application.
  proxy.updateApplication (   appName,
                null,
                fileContents,
                AppConstants.APPUPDATE_UPDATE,
                options,
                null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
```

```
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Adding to, updating, or deleting part of an application through programming

You can add to, update, or delete part of an existing application through the administrative console, the wsadmin tool, or programming. This example changes part of an application through programming. You can use this example whether you add to, update, or delete part of an existing application. Multiple changes to an application can be packaged in a single .zip file.

To learn about the structure of the .zip file, see the Updating applications topic in the *Developing and deploying applications* PDF.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add to, update, or delete part of an application on WebSphere Application Server, you must first install your application.

Perform the following tasks to add to, update, or delete part of an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter.
4. Add the listener.
5. Partially change the existing application.
6. Wait for some timeout so that the program does not end.

7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the update is done, remove the listener and quit.

After you successfully run the code, you have changed the application.

The following example shows how to add to, update, or delete part of an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//partialApp specifies the location of the partial application.
//appName specifies the name of the application.

String partialApp = "C:/apps/partial.zip";
String appName = "MyApp";

//Do a get of the administrative client to connect to
//WebSphere Application Server.

AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

// Create the notification filter.
NotificationFilterSupport myFilter = new NotificationFilterSupport();
myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
//Add the listener.
NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);
//Partially change the existing application, MyApp.

Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_PARTIALAPP);

proxy.updateApplication ( appName,
    null,
    partialApp,
    null,
    options,
    null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
```

```
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Editing applications

You can edit deployed applications through the administrative console, the wsadmin tool, or by programming. Use this example to edit a deployed application through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

Before you can edit an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to edit your deployed application.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Get information about an installed application.
4. Manipulate task data as necessary.
5. Save changes for the installed application.

After you successfully run the code, the application is edited.

The following example shows how to edit an application, based on the previous steps.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
```

```
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class aa {

    public static void main (String [] args) {

        try {

  // Connect to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
// Get information for an application with name appName:
// Pass Locale information as the preference.
Hashtable prefs = new Hashtable();
 prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());
Vector allTasks =  appMgmt.getApplicationInfo (appName, prefs, null);

// Manipulate task data as necessary.
if (task.getName().equals ("MapRolesToUsers") && !task. isTaskDisabled())
{
       // find out column index for role and user column
       // refer to the previous table to find the column names
       int roleColumn = -1;
       int userColumn = -1;
       String[] colNames = task.getColumnNames();
       for (int i=0; i < colNames.length; i++)
       {
            if (colNames[i].equals ("role"))
             roleColumn = i;
            else if (colNames[i].equals ("role.user"))
               userColumn = i;
       }

       // iterate over task data starting at row 1 as row0 is
       // column names
       String[][]data = task.getTaskData();
       for (int i=1; i < data.length; i++)
       {
            if (data[i][roleColumn].equals ("Role1"))
            {
             data[i][userColumn]="User1|User2";
               break;
            }
       }

       // now that the task data is changed, save it back
       task.setTaskData (data);
}

// Save changes back into the installed application:
// Set information for an application with name appName.
```

```
// Pass Locale information as the preference.
prefs = new Hashtable();
prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());
appMgmt.setApplicationInfo (appName, prefs, null, allTasks);

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

Preparing a module and adding it to an existing application through programming

You can add a module to an existing application through the administrative console, the wsadmin tool, or programming. Use this example to add a module through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add a module to an application on WebSphere Application Server, you must install the application.

Perform the following tasks to add a module to an application through programming.

1. Create an application deployment controller instance to populate the module file with binding information.
2. Save the binding information in the module.
3. Get the installation options.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the updateApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.
5. Connect to WebSphere Application Server.
6. Create the application management proxy.
7. Create the notification filter.
8. Add the listener.
9. Add the module to the application.
10. Specify the target for the new module.
11. Wait for some timeout so that the program does not end.
12. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
13. When the module addition is done, remove the listener and quit.

After you successfully run the code, the module is added to the application.

The following example shows how to add a module to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//moduleName specifies the name of the module that you add to the application.
//moduleURI specifies a URI that gives the target location of the module
// archive contents on a file system. The URI provides the location of the new
// module after installation. The URI is relative to the application URL.
//uniquemoduleURI specfies the URI that gives the target location of the
// deployment descriptor file. The URI is relative to the application URL.
```

```
//target specifies the cell, node, and server on which the module is installed.

String moduleName = "C:/apps/foo,jar";
String moduleURI = "Increment.jar";
String uniquemoduleURI = "Increment.jar+META-INF/ejb-jar.xml";
String target = "WebSphere:cell=cellname,node=nodename,server=servername";

//Create an application deployment controller instance, AppDeploymentController,
//to populate the Java aArchive (JAR) file with binding information.
//The binding information is WebSphere Application Server-specific deployment information.

Hashtable preferences = new Hashtable();
preferences.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
preferences.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);
AppDeploymentController controller = AppManagementFactory.readArchiveForUpdate(
    moduleName,
    moduleURI,
    AppConstants.APPUPDATE_ADD,
    preferences,
    null);
```

If the module that you add to the application lacks any bindings, add the bindings so that the module addition works. Collect and add the bindings by using the public APIs provided with WebSphere Application Server. Refer to Java documentation for the com.ibm.websphere.management.application.client.AppDeploymentController instance to learn more about how to collect and populate tasks with WebSphere Application Server specific-binding information.

```
//After you collect all the binding information, save it in the module.
controller.saveAndClose();

//Get the installation options.
Hashtable options = controller. getAppDeploymentSavedResults();

//Connect the administrative client, AdminClient, to WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Update the existing application, MyApp, by adding the module.
String appName = "MyApp";

options.put (AppConstants.APPUPDATE_CONTENTTYPE,
   AppConstants. APPUPDATE_CONTENT_MODULEFILE);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);



//Specify the target for the new module.
Hashtable mod2svr = new Hashtable();
options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, mod2svr);
mod2svr.put (uniquemoduleURI, target);
proxy.updateApplication ( appName,
    moduleURI,
    moduleName,
    AppConstants.APPUPDATE_ADD,
    options,
    null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
```

```
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Preparing and updating a module through programming

You can update a module for an existing application through the administrative console, the wsadmin tool, or programming. When you update a module, you replace the existing module with a new version. Use this example to update a module through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update a module on WebSphere Application Server, you must first install the application.

Perform the following tasks to update a module through programming.

1. Create an application deployment controller instance to populate the Java archive file with binding information.
2. Save the binding information in the module.
3. Get the installation options.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the updateApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.
5. Connect to WebSphere Application Server.
6. Create the application management proxy.
7. Create the notification filter.
8. Add the listener.
9. Replace the module in the application.
10. Specify the target for the new module.
11. Wait for some timeout so that the program does not end.
12. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
13. When the module addition is done, remove the listener and quit.

After you successfully run the code, the existing module is replaced with the new one.

The following example shows how to add a module to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//moduleName specifies the name of the module that you add to the application.
//moduleURI specifies a URI that gives the target location of the module
// archive contents on a file system. The URI provides the location of the new
// module after installation. The URI is relative to the application URL.
//uniquemoduleURI specfies the URI that gives the target location of the
// deployment descriptor file. The URI is relative to the application URL.
//target specifies the cell, node, and server on which the module is installed.
//appName specifies the name of the application to update.
String moduleName = "C:/apps/foo,jar";
String moduleURI = "Increment.jar";
String uniquemoduleURI = "Increment.jar+META-INF/ejb-jar.xml";
String target = "WebSphere:cell=cellname,node=nodename,server=servername";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

Vector tasks = proxy.getApplicationInfo (appName, new Hashtable(), null);

//Create an application deployment controller instance, AppDeploymentController,
//to populate the Java archive (JAR) file with binding information.
//The binding information is WebSphere Application Server-specific deployment information.

Hashtable preferences = new Hashtable();
preferences.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
preferences.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);
AppDeploymentController controller = AppManagementFactory.readArchiveForUpdate(
```

```
    moduleName,
    moduleURI,
    AppConstants.APPUPDATE_UPDATE,
    preferences,
    tasks);
```

If the module that you update for the application lacks any bindings, add the bindings so that the module update works. Collect and add the bindings by using the public APIs that are provided with WebSphere Application Server. Refer to Java documentation for the AppDeploymentController instance to learn more about how to collect and populate tasks with WebSphere Application Server-specific binding information.

```
//After you collect all the binding information, save it in the module.
controller.saveAndClose();

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);


//Get the installation options.
Hashtable options = controller. getAppDeploymentSavedResults();

//Update the existing application by adding the module.

options.put (AppConstants.APPUPDATE_CONTENTTYPE,
   AppConstants. APPUPDATE_CONTENT_MODULEFILE);

//Specify the target for the new module
Hashtable mod2svr = new Hashtable();
options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, mod2svr);
mod2svr.put (uniquemoduleURI, target);

proxy.updateApplication ( appName,
    moduleURI,
    moduleName,
    AppConstants.APPUPDATE_UPDATE,
    options,
    null);
// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
```

```
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Deleting a module through programming

You can delete a module from an existing application through the administrative console, the wsadmin tool, or programming. Use this example to delete a module through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can delete a module from an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to delete a module through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Delete the module.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the module is deleted, remove the listener and quit.

After you successfully run the code, the existing module is deleted from the application.

The following example shows how to delete a module from an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//moduleURI specifies a URI that gives the target location of the module.
//appName specifies the name of the application to update.
String moduleURI = "Increment.jar";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.

AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

//Update the existing application, MyApp, by deleting the module.
Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);

proxy.updateApplication ( appName,
    moduleURI,
    null,
    AppConstants.APPUPDATE_DELETE,
    options,
    null);

// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }
```

```
    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Adding a file through programming

You can add a file to an existing application through the administrative console, the wsadmin tool, or programming. This example describes how to add a file through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add a file to an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to add a file to an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Add the file to the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the file is added to the application, remove the listener and quit.

After you successfully run the code, the file is added to the application.

The following example shows how to add a file to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;
```

```
public class FileAdd {

    public static void main (String [] args) {

        try {

// Get a connection to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

 // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  String fileURI = "test.war/com/acme/abc.jsp";
  String fileContents = "C:/temp/abc.jsp";

  //Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);

  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

  Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
  options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

  // Update the application
  proxy.updateApplication (   appName,
                fileURI,
                fileContents,
                AppConstants.APPUPDATE_ADD,
                options,
                null);

// Wait; the installation Application Programming Interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(90000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}


// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
```

```
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Updating a file through programming

You can update a file for an existing application through the administrative console, the wsadmin tool, or programming. This example describes how to update a file through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update a file for an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to update a file through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Update the file in the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.

8. When the installation is done, remove the listener and quit.

After you successfully run the code, the file is updated for the application.

The following example shows how to add a file to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//fileContents specifies the name of the file that you add to the application.
//appName specifies the name of the application.
//fileURI specifies a URI that gives the target location of the file. The URI
// provides the location of the new module after installation. The URI is
// relative to the application URL.

String fileContents = "C:/apps/test.jsp";
String appName = "MyApp";
String fileURI = "SomeWebMod.war/com/foo/abc.jsp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

proxy.updateApplication ( appName,
    fileURI,
    fileContents,
    AppConstants.APPUPDATE_UPDATE,
    options,
    null);

// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
```

```
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit.

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Deleting a file through programming

You can delete a file from an existing application through the administrative console, the wsadmin tool, or programming. Use this example to delete a file through programming.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can delete a file from an application on WebSphere Application Server, you must first install the application.

Perform the following tasks to delete a file through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Delete the file from the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the file is deleted from the application, remove the listener and quit.

After you successfully run the code, the file is deleted from the application.

The following example shows how to delete a file based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//fileURI specifies a URI that gives the target location of the file. The URI
// provides the location of the new module after installation. The URI is
// relative to the application URL.
//appName specifies the name of the application.

String fileURI = "Increment.jar/com/acme/Foo.class";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

//Update the existing application, MyApp, by deleting the file.
Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

proxy.updateApplication ( appName,
    fileURI,
    null,
    AppConstants.APPUPDATE_DELETE,
    options,
    null);

// Wait for some timeout. The installation Application Programming Interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;
```

```
        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //Once the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

Extending application management operations through programming

You can use the common deployment framework to add additional logic to application management operations. The additional logic can do such tasks as code generation, configuration operations, additional validation, and so on. This topic demonstrates, through programming, how to plug into the *common deployment framework* to extend application management operations.

This task assumes a basic familiarity with Java application programming interfaces (APIs). For information on the Java APIs, see Java Management Extensions (JMX) API documentation.

Before you can extend application management operations, you must first install WebSphere Application Server.

Use this example to extend application management through programming. The tasks that the extensions provide are available through all the administrative clients, such as the wsadmin tool, the administrative console, or through programmatic APIs that the AppManagement MBean provides.

1. Define your extension as an Eclipse plug-in and add a `plugin.xml` file to register your extension provider with the deployment framework.
   a. In the `plugin.xml` file, provide an extension provider implementation class for the common-deployment-framework-extensionprovider extension point.
   b. Put the plug-in Java archive (JAR) file in the `plugins` directory of your WebSphere Application Server installation.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.ibm.myproduct.MyExtensionProvider"
   name="My Extension"
   version="1.0.0">
```

```
    <extension point="common-deployment-framework-extensionprovider">
       <action class="com.acme.MyExtendProviderImpl"/>
    </extension>
</plugin>
```

2. Provide an extension provider.

   An extension provider class provides steps for a given operation on an application Enterprise archive (EAR) file. Before an operation runs, the deployment framework queries all the registered extension providers for additional steps. A single list of steps is passed to each provider. Each provider can add steps to the list. The default provider that the deployment framework provides is called first to populate the list with default steps. Other extension providers are called next.

   Various operations that you can extend through the common deployment framework are defined as constants in the DeploymentConstants class. These operations are described in the following table. Some operations are split on multiple lines for printing purposes.

Table 24.

Operation	Description
DeploymentConstants.CDF_OP_INSTALLJ2EE	Installs a Java 2 Platform, Enterprise Edition (J2EE) EAR file
DeploymentConstants.CDF_OP_EDITJ2EE	Edits a deployment application configuration
DeploymentConstants.CDF_OP_UPDATEJ2EE	Applies a fine-grained update to an application such as addition, removal, or update of a file or a module; or partial update of an application
DeploymentConstants.CDF_OP_UNINSTALLJ2EE	Uninstalls a J2EE application
`DeploymentConstants.CDF_OP_CREATE_EAR_WRAPPERJ2EE`	Wraps the contents input to the application installation into an EAR file

   The AppManagement MBean, which is responsible for deploying and managing J2EE applications on WebSphere Application Server, runs all the operations except the CDF_OP_CREATE_EAR_WRAPPERJ2EE operation. Deploy the extensions that extend these operations in the `plugins` directory of the deployment manager.

   Either the wsadmin utlity or the administrative console runs the CDF_OP_CREATE_EAR_WRAPPERJ2EE operation when the input contents that are supplied to the CDF_OP_INSTALLJ2EE operation are not packaged as an EAR file. Deploy an extension that extends the CDF_OP_CREATE_EAR_WRAPPERJ2EE operation in the `plugins` directory of the wsadmin installation.

   The following example provides an extension provider that does the following tasks:

   a. Adds two additional steps for the application installation operation

   b. Adds one step for wrapping input contents into an EAR file

```
package com.acme;

import com.ibm.websphere.management.deployment.registry.ExtensionProvider;
import com.ibm.websphere.management.deployment.core.DeploymentConstants;

public class MyExtensionProviderImpl extends ExtensionProvider {
  public void addSteps (String type, String op, String phase,
   List steps)
  {
   if (op.equals (DeploymentConstants.CDF_OP_INSTALLJ2EE))
   {
    // Add a code generation step.
    steps.add (0, new com.acme.CodeGenStep());
    // Add a configuration step.
    steps.add (new com.acme.ConfigStep());
   }
   else if (op.equals (DeploymentConstants.CDF_OP_CREATE_EAR_WRAPPERJ2EE))
   {
```

```
    // Add an ear-wrapper step.
    steps.add (new com.acme.EarWrapperStep());
   }
  }
}
```

3. Provide the deployment step implementation.

   An extension provider adds a deployment step. The step contains logic that performs additional processing in an application management operation. The logic provides the step access to the deployment context and the deployable object. The deployment context provides information, such as the name of the operation, the configuration session ID, a temporary location for creating temporary files, operation parameters, and so on. The deployable object wraps the deployment content input to the operation. For example, the deployable object wraps the J2EE EAR file for the installation operation or a file, a module, or a partial application for the update operation.

   - The following example illustrates how an extension during installation entirely changes an EAR file that is input to the installation operation. The example provides a deployment step during the installation operation that does the following tasks:
     a. Runs code generation to generate a new EAR file.
     b. Calls the setContentPath method in the DeployableObject class to set the new EAR file path. The default installation logic, such as steps that the default installation logic adds, uses this new EAR file for installation in the configuration repository.

     ```
     package com.acme;

     import com.ibm.websphere.management.deployment.core.DeploymentStep;
     import com.ibm.websphere.management.deployment.core.DeployableObject;

     public class CodeGenStep extends DeploymentStep
     {
       public void execute (DeployableObject dObject)
       {
        EARFile earFile = (EARFile)dObject.getHandle();
        String newEARPath = null;
        // Use step specific logic to create another EAR file after code generation.
        ...
        newEARPath = _context.getTempDir() + "new.ear";

        dObject.setContentPath (newEARPath);
       }
     }
     ```

   - The following example provides a deployment step that:
     a. Reads the contents of the input EAR file.
     b. Manipulates the configuration session accessed through the context instance, _context.

     ```
     package com.acme;

     public class ConfigStep extends DeploymentStep
     {
       public void execute (DeployableObject dObject)
       {
        EARFile earFile = (EARFile) dObject.getHandle();

        // Use the following example code to perform the configuration.
        String sessionID = _context.getSessionID();
        com.ibm.websphere.management.Session session = new
         com.ibm.websphere.management.Session (sessionID, true);
        // Use the configuration service to perform the configuration steps.
        ...

        // Read the application configuration.
        Application appDD = earFile.getDeploymentDescriptor();
        ...
     ```

```
   String newEARPath = null;
  }
}
```

- The following example provides a deployment step to wrap arbitrary content around an EAR file. Application management logic accepts only the EAR file for deployment. An extension is required if you want to input anything other than an EAR file to the deployment process.

```
package com.acme;

import com.ibm.websphere.management.deployment.core.DeploymentStep;
import com.ibm.websphere.management.deployment.core.DeployableObject;

public class EarWrapperStep extends DeploymentStep
{
  public void execute (DeployableObject dObject)
  {
   Archive archive = (Archive) dObject.getHandle();
   String newEARPath = null;
   // provide your logic to wrap the jar with the ear
   ...
   newEARPath = //;
   // Set the new ear path back into DeploymentContext
   this.getContext().getContextData()
     .put(DeploymentContext.RETURN_Object_key, newEARPath);


  }
}
```

Through programming, you have plugged into the common deployment framework to extend application management operations.

You can extend other application management operations, or do any other administrative operations you choose.

Chapter 5. Using command line tools

WebSphere Application Server Network Deployment provides many tools that you can call from a command line.

There are several command line tools that you can use to start, stop, and monitor WebSphere Application Server Network Deployment processes and nodes. These tools only work on local servers and nodes. They cannot operate on a remote server or node.

To administer a remote server, you can use the wsadmin scripting program connected to the deployment manager for the cell in which the target server or node is configured. See Deploying and managing using scripting for more information about using the wsadmin scripting program. You can also use the V6 administrative console which runs in the deployment manager for the cell. For more information about using the administrative console, see Deploying and managing with the GUI.

All command line tools function relative to a particular profile. If you run a command from a *app_server_root*/`bin` directory, the command runs within the default profile. To specify a different profile, use one of the following methods:

- Specify the -profileName option. The profile that you specify with this option will be used instead of the default profile. For example:
  1. Change to the *app_server_root*/`bin` directory.
  2. Type the following command: `startServer` *`server1`* `-profileName` *`AppServerProfile`*

  In this example, the command starts the application server named server1 which exists in the configuration for the *`AppServerProfile`* profile.
- Run the command from the `bin` directory of a specific profile. For example, run the command from the `bin` directory of the *`MyProfile`* profile:
  1. Change to the *profile_root*/`bin` directory.
  2. Type the following command: `startServer` *`server1`*

  In this example, the command will function inside the *`MyProfile`* profile.

To use the command line tools, perform the following steps:

1. Open a system command prompt.
2. Change directories to the *app_server_root*/`bin` directory.
3. Run the command.

The command runs the requested function and displays the results on the screen.

Refer to the command log file for additional information.

Example: Security and the command line tools

If you want to enable WebSphere Application Server security, you need to provide the command line tools with authentication information.

Without authentication information, the command line tools receive an `AccessDenied` exception when you attempt to use them with security enabled. There are multiple ways to provide authentication data:

- Most command line tools support a -username and -password option for providing basic authentication data. Specify the user ID and password for an administrative user. For example, you can use a member of the administrative console users with operator or administrator privileges, or the administrative user ID configured in the user registry. The following example demonstrates the **stopNode** command, which specifies command line parameters:

  ```
  stopNode -username adminuser -password adminpw
  ```

- You can place the authentication data in a properties file that the command line tools read. The default file for this data is the `sas.client.props` file in the `properties` directory for the WebSphere Application Server.

startServer command

The **startServer** command reads the configuration file for the specified application server and starts the server.

Depending on the options you specify, you can launch a new Java virtual machine (JVM) API to run the server process, or write the launch command data to a file.

You do not have to use a user name and password with the **startServer** command because this command launches a server process but does not invoke an MBean method.

For more information about where to run this command, see the Using command line tools article.

Syntax

The command syntax is as follows:

```
startServer <server> [options]
```

where `server` is the name of the application server you want to start.

This argument is required.

Parameters

The following options are available for the **startServer** command:

-quiet
Suppresses the progress information that the **startServer** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information to the log file for debugging purposes.

-timeout <seconds>
Specifies the waiting time before server initialization times out and returns an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for server status callback.

-script [<script fileName>] -background
Generates a launch script with the **startServer** command instead of launching the server process directly. The launch script name is an optional argument. If you do not supply the launch script name, the default script file name is `start_<server>` based on the *<server>* name passed as the first argument to the **startServer** command. The `-background` parameter is an optional parameter that specifies that the generated script will run in the background when you execute it.

-J <java_option>
Specifies options to pass through to the Java interpreter.

-recovery
Specifies that the server will start in recovery mode, perform a transactional recovery, and shut down. The server will not accept any new transactions while it is in recovery mode. When you start the server again, resources that were unavailable due to questionable transactions will be available.

Use this option if a server fails and you do not want to accept new transactions during the recovery process.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
startServer server1

startServer server1 -script (produces the start_server1.sh file)

startServer server1 -trace (produces the startserver.log file)
```

stopServer command

The **stopServer** command reads the configuration file for the specified server process. This command sends a Java Management Extensions (JMX) command to the server telling it to shut down.

By default, the **stopServer** command does not return control to the command line until the server completes the shut down process. There is a -nowait option to return immediately, as well as other options to control the behavior of the **stopServer** command. For more information about where to run this command, see the Using command tools article.

If security is enabled, you must use a user name and password with the **stopServer** command because the command invokes an MBean method on the server.

Syntax

The command syntax is as follows:

```
stopServer <server> [options]
```

where *server* is the name of the configuration directory of the server you want to stop.

This argument is required.

Parameters

The following options are available for the **stopServer** command:

-nowait
Tells the **stopServer** command not to wait for successful shutdown of the server process.

-quiet
Suppresses the progress information that the **stopServer** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes. The trace output is written to the stopServer.log file which is located in the *profile_root*/logs/*server* directory.

-timeout <seconds>
Specifies the time to wait for server shutdown before timing out and returning an error.

-statusport <portNumber>
Supports an administrator in setting the port number for server status callback.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are SOAP or Remote Method Invocation (RMI).

-port <portNumber>
Specifies the server Java Management Extensions (JMX) port to use explicitly, so that you can avoid reading the configuration files to obtain the information.

-username <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the server.

Note: If you are running in a secure environment but have not provided a user ID and password, you will receive the following error message:

```
ADMN0022E: Access denied for the stop operation on Server MBean due
to insufficient or empty credentials.
```

To solve this problem, provide the user ID and password information.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
stopServer server1

stopServer server1 -nowait

stopServer server1 -trace (produces the stopserver.log file)
```

startManager command

Use the startManager command to manipulate a deployment manager with scripting.

The **startManager** command reads the configuration file for the Network Deployment manager process and constructs a **launch** command. Depending on the options you specify, the **startManager** command launches a new Java virtual machine (JVM) API to run the manager process, or writes the **launch** command data to a file.

You must run this command from the *profile_root*/`bin` directory of a Network Deployment installation.

You do not have to use a user name and password with the **startManager** command because this command launches a server process but does not invoke an MBean method.

For more information about where to run this command, see the Using command line tools article.

Syntax

The command syntax is as follows:

```
startManager [options]
```

Parameters

The following options are available for the **startManager** command:

-quiet
Suppresses the progress information that the **startManager** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file using the **startManager** command for debugging purposes.

-timeout <seconds>
Specifies the waiting time before deployment manager initialization times out and returns an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for deployment manager status callback.

-script [<script fileName>] -background
Generates a launch script with the **startManager** command instead of launching the deployment manager process directly. The launch script name is an optional argument. If you do not provide the launch script name, the default script file name is `<start_dmgr>`. The `-background` parameter is an optional parameter that specifies that the generated script will run in the background when you execute it.

-J-<java_option>
Specifies options to pass through to the Java interpreter.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
startManager

startManager -script (produces the start_dmgr.sh file)

startManager -trace (produces the startmanager.log file)
```

stopManager command

The **stopManager** command reads the configuration file for the Network Deployment manager process.

It sends a Java Management Extensions (JMX) command to the manager telling it to shut down. By default, the **stopManager** command waits for the manager to complete the shutdown process before it returns control to the command line. There is a -nowait option to return immediately, as well as other options to control the behavior of the **stopManager** command. For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
stopManager [options]
```

Parameters

The following options are available for the **stopManager** command:

-nowait
Tells the **stopManager** command not to wait for successful shutdown of the deployment manager process.

-quiet
Suppresses the progress information that the **stopManager** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information to a file for debugging purposes. The trace output is written to the stopmanager.log file which is located in the *profile_root*/logs directory.

-timeout <seconds>
Specifies the waiting time for the manager to complete shutdown before timing out and returning an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for server status callback.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are SOAP or Remote Method Invocation (RMI).

-port <portNumber>
Specifies the deployment manager JMX port to use explicitly, so that you can avoid reading the configuration files to obtain information.

-username <name>
Specifies the user name for authentication if security is enabled in the deployment manager. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the deployment manager. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the deployment manager.

Note: If you are running in a secure environment but have not provided a user ID and password, you receive the following error message:

```
ADMN0022E: Access denied for the stop operation on Server MBean due
to insufficient or empty credentials.
```

To solve this problem, provide the user ID and password information.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
stopManager

stopManager -nowait

stopManager -trace (produces the stopmanager.log file)
```

startNode command

The **startNode** command reads the configuration file for the node agent process and constructs a **launch** command.

You do not have to use a user name and password with the **startNode** command because this command launches a server process but does not invoke an MBean method.

For more information about where to run this command, see the Using command line tools article.

Syntax

The command syntax is as follows:

```
startNode [options]
```

Parameters

The following options are available for the **startNode** command:

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-trace

-timeout <seconds>
Specifies the waiting time before node agent initialization times out and returns an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for node agent status callback.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

---

stopNode command

The **stopNode** command reads the configuration file for the Network Deployment node agent process and sends a Java Management Extensions (JMX) command telling the node agent to shut down.

By default, the **stopNode** command waits for the node agent to complete shutdown before it returns control to the command line. There is a -nowait option to return immediately, as well as other options to control the behavior of the **stopNode** command. For more information about where to run this command, see the Using command tools article.

If you stop the server before stopping the node agent using the **stopserver -servers** command, the server will not restart when you issue a **startserver** command.

Syntax

The command syntax is as follows:

```
stopNode [options]
```

Parameters

The following options are available for the **stopNode** command:

-nowait
Tells the **stopNode** command not to wait for successful shutdown of the node agent process.

-quiet
Suppresses the progress information that the **stopNode** command prints in normal mode.

-logfile <fileNname>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-timeout <seconds>
Specifies the waiting time for the agent to shut down before timing out and returning an error.

-statusport <portNumber>
Specifies that an administrator can set the port number for server status callback.

-stopservers
Stops all application servers on the node before stopping the node agent.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are SOAP or Remote Method Invocation (RMI).

-port <portNumber>
Specifies the node agent JMX port to use explicitly, so that you can avoid reading configuration files to obtain the information.

-username <name>
Specifies the user name for authentication if security is enabled in the node agent. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the node agent. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the node agent.

Note: If you are running in a secure environment but have not provided a user ID and password, you receive the following error message:

```
ADMN0022E: Access denied for the stop operation on Server MBean due
to insufficient or empty credentials.
```

To solve this problem, provide the user ID and password information.

-help
Prints a usage statement.

Note: When requesting help for the usage statement for the **stopNode** command, a reference to the **stopServer** command displays. All of the options displayed for this usage statement apply to the **stopNode** command.

-? Prints a usage statement.

Note: When requesting help for the usage statement for the **stopNode** command, a reference to the **stopServer** command displays. All of the options displayed for this usage statement apply to the **stopNode** command.

Usage scenario

The following examples demonstrate correct syntax:

```
stopNode

stopNode -nowait

stopNode -trace (produces the stopnode.log file)
```

addNode command

The **addNode** command incorporates a WebSphere Application Server installation into a cell.

For more information about where to run this command, see the Using command line tools article. Depending on the size and location of the new node you incorporate into the cell, this command can take a few minutes to complete. You must have Administrator privileges to use the **addNode** function.

The node agent server is automatically started as part of the **addNode** command unless you specify the `-noagent` option. If you recycle the system that hosts an application server node, and did not set up the node agent to act as an operating system daemon, you must issue a **startNode** command to start the node agent before starting any application servers.

The following items are new in V6.1:

- Ports generated for the node agent are unique for all the profiles in the installation. For development purposes, you can create multiple profiles on the same installation and add them to one or more cells without worrying about ports conflicts.
- If you want to specify the ports that the node agent uses, specify it in a file with the file name passed with the -portprops option. The format of the file is key=value pairs, one on each line, with the key being the same as the port name in the serverindex.xml file.
- If you want to use a number of sequential ports, the -startingport option works the same as it does in V5.x. This means that port conflicts with other profiles will not be detected.

Syntax

The command syntax is as follows:

```
addNode dmgr_host [dmgr_port] [-conntype type] [-includeapps]
[-startingport portnumber] [-portprops qualified_filename]
[-nodeagentshortname name] [-nodegroupname name] [-includebuses name]
[-registerservice] [-servicename name] [-servicepassword password]
[-coregroupname name] [-noagent] [-statusport port] [-quiet] [-nowait]
[-logfile filename] [-replacelog] [-trace] [-username uid]
[-password pwd] [-localusername localuid]
[-localpassword localpwd] [-help]
```

The `dmgr_host` argument is required. All of the other arguments are optional. The default port number is 8879 for the default SOAP port of the deployment manager. SOAP is the default Java Management Extensions (JMX) connector type for the command. If you have multiple WebSphere Application Server installations or multiple profiles, the SOAP port may be different than 8879. Examine the deployment manager SystemOut.log to see the current ports in use.

Parameters

The following options are available for the **addNode** command:

-conntype <type>
Specifies the JMX connector type to use for connecting to the deployment manager. Valid types are SOAP or RMI, which stands for Remote Method Invocation.

-includeapps
By default the **addNode** command does not carry over applications from the stand-alone servers on the new node to the cell. In general, you should install applications using the deployment manager. The -includeapps option tells the **addNode** command to carry over the applications from a node. If the application already exists in the cell, a warning is printed and the application does not install in the cell.

The applications will be mapped to the server that you federated using the **addNode** command. When the **addNode** command operation completes, the applications will run on that server when the server is started. Since these applications are part of the network deployment cell, you can map them to other servers and clusters in the cell using the administrative console. See the Mapping modules to servers article in the *Developing and deploying applications* PDF for more information.

You should not use the -includeapps option if your node to be federated includes WebSphere Application Server supplied applications, such as the samples. If you do, the second node to be federated that includes these applications will be rejected because the applications already exist in the cell and application merge is not supported.

If you use the -includeapps option on a node that includes a large number of applications, the timeout for the administrative connector needs to be extended to account for the additional time required to transfer all of the applications to the deployment manager during addNode and to remotely install them into the cell. The -includeapps option is not a recommended approach unless only a few unique applications exist on the node.

By default, during application installation, application binaries are extracted in the *app_server_root*/installedApps/*cellName* directory. After the **addNode** command, the cell name of the configuration on the node that you added changes from the base cell name to the deployment manager cell name. The application binaries are located where they were before the **addNode** command ran, for example, *app_server_root*/installedApps/*old_cellName*.

If the application was installed by explicitly specifying the location for binaries as the following example:

${*app_server_root*}/${CELL}

where the variable ${CELL}, specifies the current cell name, then when the **addNode** command runs, the binaries are moved to the following directory:

${*app_server_root*}/currentCellName

Federating the node to a cell using the **addNode** command does not merge any cell level configuration, including virtual host information. If the virtual host and aliases for the new cell do not match WebSphere Application Server, you cannot access the applications running on the servers. You have to manually add all the virtual host and host aliases to the new cell, using the administrative console running on the deployment manager.

Note: When the -includeapps parameter is specified, an OutOfMemoryError might occur if the Java Virtual Machine heap size isn't large enough. When this error occurs, the following error message is issued:

```
ADMU0111E: Program exiting with error: java.lang.OutOfMemoryError
```

This error can occur when large applications are processed, or when there is a large number of applications in the Base Application Server.

To recover from this error and successfully federate the Base Application Server, you must:

1. Issue the cleanupNode.sh command on your deployment manager server. See "cleanupNode command" on page 945 for more information about this command.
2. Increase the JVM heap size for the addNode script. When you issue the addNode.sh command, the JVM heap size is set to -Xms128m -Xmx512m. To increase these values, edit the JVM_EXTRA_CMD_ARGS variable in the *config_root*/bin/setupCmdLine.sh file of the Base Application Server being federated. For example, you might specify the following (all on one line):
3. Reissue the addNode.sh command.

-profileName

Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile. If you are adding a non-default profile to the deployment manager cell, this parameter is required.

-user <name> or -username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option. The user name that you choose must be a pre-existing user name.

-nowait
Tells the **addNode** command not to wait for successful initialization of the launched node agent process.

-quiet
Suppresses the progress information that the **addNode** command prints in normal mode.

-logfile <filename>
Specifies the location of the log file to which information gets written. By default, the log file is called addNode.log and is created in the `logs` directory of the profile for the node being added.

-trace
Generates additional trace information in the log file for debugging purposes.

-replacelog
Replaces the log file instead of appending to the current log. By default, the **addNode** command appends to the existing trace file. This option causes the **addNode** command to overwrite the trace file.

-noagent
Tells the **addNode** command not to launch the node agent process for the new node.

-password <password>
Specifies the password for authentication if security is enabled. The password that you choose must be one that is associated with a pre-existing user name.

-startingport <portNumber>
Supports the specification of a port number to use as the base port number for all node agent and Java Messaging Service (JMS) server ports created during the **addNode** command. With this support you can control which ports are defined for these servers, rather than using the default port values. The starting port number is incremented by one to calculate the port number for every node agent port and JMS server port configured during the **addNode** command.

If multiple node agents exist on the same physical server, you can define the base port number for each by using the -startingport parameter prior to federation, or by modifying the ports in the node agent section of serverindex.xml.

-registerservice
(Windows only) Registers the node agent as a Windows service.

-servicename <user>
(Windows only) Use the given user name as the Windows service user.

-servicepassword <password>
(Windows password) Use the given password as the Windows service password.

-portprops <filename>
Passes the name of the file that contains key-value pairs of explicit ports that you want the new node agent to use. For example, to set your SOAP and RMI ports to 3000 and 3001, create a file with the following two lines and pass it as the parameter:

```
SOAP_CONNECTOR_ADDRESS=3000
BOOTSTRAP_ADDRESS=3001
```

-coregroupname <name>
The name of the core group in which to add this node. If you do not specify this option, the node will be added to the DefaultCoreGroup.

-nodegroupname <name>
The name of the node group in which to add this node. If you do not specify, the node is added to the DefaultNodeGroup.

-includebuses
Copies the buses from the node to be federated to the cell. This parameter will also attempt to copy the service integration bus configuration of the remote node into the cell. If the destination cell already contains a bus with the same name as any bus at the remote node, the add node will fail. To prevent this problem, there are actions you can take before using the **addNode** command. You can delete the bus with that name in the destination cell, rename the bus to be added to the cell, or manually configure the bus already located in the cell.

-nodeagentshortname <name>
The shortname to use for the new node agent.

-localusername <name>
Specifies the user name for authentication for existing application servers on the node that you want to federate. This parameter is only applicable if security is enabled for the application server.

-localpassword <password>
Specifies the password for authentication for existing application servers on the node that you want to federate. The password that you choose must be one that is associated with a preexisting user name. This parameter is only applicable if security is enabled for the application server.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
addNode testhost 8879 (adds an application server to the deployment manager)

addNode deploymgr 8879 -trace (produces the addNode.log file)

addNode host25 8879 -nowait (does not wait for a node agent process)
```

where 8879 is the default port.

Best practices for adding nodes using command line tools

Use the **addNode** command to add a standalone node into a cell.

The **addNode** command does the following:

- Copies the base WebSphere Application Server cell configuration to a new cell structure. This new cell structure matches the structure of deployment manager.
- Creates a new node agent definition for the node that the cell incorporates.
- Sends commands to the deployment manager to add the documents from the new node to the cell repository.
- Performs the first configuration synchronization for the new node, and verifies that this node is synchronized with the cell.
- Launches the node agent process for the new node.
- Updates the setupCmdLine.bat or setupCmdline.sh files and the wsadmin.properties file to point to the new cell.

- After federating the node, the **addNode** command backs up the `plugin-cfg.xml` file from the *app_server_root*`/config/cells` directory to the `config/backup/base/cells` directory. The **addNode** command regenerates a new `plugin-cfg.xml` file at the Deployment Manger and the nodeSync operation copies the files to the node level.

Tips for using the **addNode** command:
- Verify that the deployment manager and nodes are updated to the same revision level within the WebSphere Application Server. For example, a deployment manager at level 6.0.1 will be unable to federate with nodes at 6.0.2.
- Do not put WebSphere Application Server .jar files on the generic `CLASSPATH` variable (default class path) for the overall system.
- By default, applications that are installed on the node will not copy to the cell. If you install an application after using the **addNode** command, the application will install on the cell. By specifying the `-includeapps` option, you force the **addNode** command to copy applications from the node to the cell. Applications with duplicate names will not copy to the cell.
- Cell-level documents are not merged. Any changes that you make to the standalone cell-level documents before using the **addNode** command must be repeated on the new cell. For example, virtual hosts.
- If you receive an OutOfMemory exception while using the **addNode** command, you may need to increase the heap size of the deployment manager. To increase the heap size of the deployment manager, adjust the Maximum Heap Size parameter using the administrative console. In the administrative console, go to System Administration > Deployment Manager > Java and Process Management > Process Definition > Java Virtual Machine > Maximum Heap Size.
- In some instances it may take longer than anticipated for the deployment manager to respond to the **addNode** command. The default timeout value, which determines how long the client will wait for a server response, is appropriate in the majority of cases. However, you may require more time for the server to respond under heavier processing conditions. For example, if you include the `-includeapps` option and have a large number of applications, or the applications are very large, the default value of 180 seconds may be insufficient. To change the default timeout value, open the file `$WAS_HOME/profiles/<profile name>/properties/soap.client.props` in any ASCII text editor and find the following line (shown here with default value of 180 seconds):

  ```
  com.ibm.SOAP.requestTimeout=180
  ```

  If you need to change the default you can edit this line to set the timeout to a value more appropriate for your situation (**Note:** setting the above value to 0 will disable the timeout check altogether). If the timeout value is set too high you will have to wait a long time to determine if the **addNode** command will successfully complete its request to the deployment manager. If the value is set too short the deployment manager will not have sufficient time to complete the request before the **addNode** command concludes that the deployment manager is not responding and will respond with an error. Other factors that may affect server timeouts include the processing load or excessive paging on the deployment manager and network latency. Some of these conditions may be transient.

serverStatus command

Use the **serverStatus** command to obtain the status of one or all of the servers configured on a node.

For more information about where to run this command, see the Using command line tools article.

Syntax

The command syntax is as follows:

```
serverStatus <server>|-all [options]
```

The first argument is required. The argument is either the name of the server for which status is desired, or the -all keyword which requests status for all servers defined on the node.

Parameters

The following options are available for the **serverStatus** command:

-quiet
Suppresses the progress information that the **serverStatus** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
serverStatus server1

serverStatus -all (returns status for all defined servers)

serverStatus -trace (produces the serverStatus.log file)
```

removeNode command

The **removeNode** command returns a node from a Network Deployment distributed administration cell to a standalone WebSphere Application Server installation.

For more information about where to run this command, see the Using command tools article. You must have Administrator privileges to use the **removeNode** function.

The **removeNode** command only removes the node-specific configuration from the cell. This command does not uninstall any applications that were installed as the result of executing an **addNode** command. Such applications can subsequently deploy on additional servers in the Network Deployment cell. As a consequence, an **addNode** command with the -includeapps option ran after a **removeNode** command does not move the applications into the cell because they already exist from the first **addNode** command. The resulting application servers added on the node do not contain any applications. To deal with this situation, add the node and use the deployment manager to manage the applications. Add the applications to the servers on the node after it is incorporated into the cell.

The **removeNode** command does the following:
- Stops all of the running server processes in the node, including the node agent process.
- Removes the configuration documents for the node from the cell repository by sending commands to the deployment manager.
- Copies the original application server cell configuration into the active configuration.

Depending on the size and location of the new node you remove from the cell, this command can take a few minutes to complete.

Note:
An application server node that is built as part of a cell profile creation does not have an original configuration. Therefore, the **removeNode** command is not able to restore the node to a usable base configuration. When removing a node that was federated as part of a cell profile creation, the **removeNode** command indicates that removing a node of this type causes the node to be unusable and the node cannot be federated again. To delete the profile for this node, run the **cleanupNode** command. A new profile can be created using the Profile Management Tool or the **manageprofiles** command.

Syntax

The command syntax is as follows:

```
removeNode [options]
```

All arguments are optional.

Parameters

The following options are available for the **removeNode** command:

-quiet
Suppresses the progress information that the **removeNode** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information is written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-statusport <portNumber>
Specifies that an administrator can set the port number for the node agent status callback.

-username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled.

-force
Cleans up the local node configuration regardless of whether you can reach the deployment manager for cell repository cleanup. After using the `-force` parameter, you may need to use the **cleanupNode** command on the deployment manager.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
removeNode -quiet
removeNode -trace (produces the removeNode.log file)
```

cleanupNode command

The **cleanupNode** command cleans up a node configuration from the cell repository.

Only use this command to clean up a node if you have a node defined in the cell configuration, but the node no longer exists. For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
cleanupNode <node name> <deploymgr host> <deploymgr port> [options]
```

where the first argument is required.

Parameters

The following options are available for the **cleanupNode** command:

-quiet
Suppresses the progress information that the **cleanupNode** command prints in normal mode.

-trace
Generates trace information into a file for debugging purposes.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile. The -profileName parameter specifies the deployment manager profile to run the command against. Because the dmgr profile is not the default profile, you must specify this parameter.

Usage scenario

The following examples demonstrate correct syntax:

```
cleanupNode myNode -profileName dmgr

cleanupNode myNode -trace -profileName mydmgr
```

syncNode command

The **syncNode** command forces a configuration synchronization to occur between the node and the deployment manager for the cell in which the node is configured.

The node agent server runs a configuration synchronization service that keeps the node configuration synchronized with the master cell configuration. If the node agent is unable to run because of a problem in the node configuration, you can use the **syncNode** command to perform a synchronization when the deployment manager is not running in order to force the node configuration back in sync with the cell configuration.

The syncNode.log file is located in the *app_server_root*/*profile_name*/logs directory.

For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
syncNode <deploymgr host> <deploymgr port> [options]
```

where the <deploymgr *host*> argument is required.

Parameters

The following options are available for the **syncNode** command:

-stopservers
Tells the **syncNode** command to stop all servers on the node, including the node agent, before performing configuration synchronization with the cell.

-restart
Tells the **syncNode** command to launch the node agent process after configuration synchronization completes.

-nowait
Tells the **syncNode** command not to wait for successful initialization of the launched node agent process.

-quiet
Suppresses the progress information that the **syncNode** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into a file for debugging purposes.

-timeout <seconds>
Specifies the waiting time before node agent initialization times out and returns an error.

-statusport <portnumber>
Specifies that an administrator can set the port number for node agent status callback.

-username <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled.

-conntype <type>
Specifies the Java Management Extensions (JMX) connector type to use for connecting to the deployment manager. Valid types are SOAP or Remote Method Invocation (RMI).

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

```
syncNode.sh testhost 8879

syncNode.sh deploymgr 8879 -trace (produces the syncNode.log file)

syncNode.sh host25 4444 -stopservers -restart
(assumes that the deployment manager JMX port is 4444)
```

renameNode command

Use the **renameNode** command to modify the node name of a federated server. Launch this command from the node whose name you want to modify. When you issue this command the following occurs:

1. Connects to the deployment manager.
2. Stops all servers.
3. Changes the node configuration on the deployment manager.
4. Synchronizes the node.

For more information about where to run this command, see the Using command line tools article.

Syntax

The command syntax is as follows:

```
renameNode.sh dmgr_host dmgr_port node_name
[-nodeshortname name][-trace][-conntype type][-username uid]
[-password pwd][-logfile filename] [-help]
```

Parameters

The following options are available for the **renameNode** command:

-nodeshortname <name>
The short name of the node.

-trace
Generates additional trace information in the log file for debugging purposes.

-conntype <type>
Specifies the JMX connector type to use for connecting to the deployment manager. Valid types are SOAP or RMI, which stands for Remote Method Invocation.

-username <uid>
Specifies the user name for authentication if security is enabled. Acts the same as the -user option. The user name that you choose must be a pre-existing user name.

-password <pwd>
Specifies the password for authentication if security is enabled. The password that you choose must be one that is associated with a pre-existing user name.

-logfile <filename>
Specifies the location of the log file to which information gets written. By default, the log file is called renameNode.log and is created in the logs directory of the profile for the node being renamed.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following examples demonstrate correct syntax:

Using Windows platforms:

```
renameNode.bat localhost 8879 newnode
```

Using UNIX platforms:

```
renameNode.sh localhost 8879 newnode
```

backupConfig command

The **backupConfig** command is a simple utility to back up the configuration of your node to a file.

By default, all servers on the node stop before the backup is made so that partially synchronized information is not saved. For more information about where to run this command, see the Using command line tools article. If you do not have root authority, you must specify a path for the backup file in a location where you have write permission. The backup file will be in zip format and a `.zip` extension is recommended.

In a UNIX or Linux environment, the **backupConfig** command does not save file permissions or ownership information. The **restoreConfig** command uses the current umask and effective user ID (EUID) to set the permissions and ownership when restoring a file. If it is required that the restored files have the original permissions and ownership, use the **tar** command (available on all UNIX or Linux systems) to back up and restore the configuration.

Syntax

The command syntax is as follows:

```
backupConfig.sh <backup_file> [options]
```

where *backup_file* specifies the file to which the backup is written. If you do not specify one, a unique name is generated.

Parameters

The following options are available for the **backupConfig** command:

-nostop
Tells the **backupConfig** command not to stop the servers before backing up the configuration.

-quiet
Suppresses the progress information that the **backupConfig** command prints in normal mode.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName <profileName>
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into the log file for debugging purposes.

-username <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the server.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following example creates a new file that includes the current date:

```
backupConfig.sh WebSphereConfig_2003-04-22.zip
```

The following example creates a file called `myBackup.zip` and does not stop any servers before beginning the backup process.

```
backupConfig.sh myBackup.zip -nostop
```

restoreConfig command

Use the restoreConfig command to restore the configuration of your node after backing up the configuration using the backupConfig command.

The **restoreConfig** command is a simple utility to restore the configuration of your node after backing up the configuration using the **backupConfig** command. By default, all servers on the node stop before the configuration restores so that a node synchronization does not occur during the restoration. If the configuration directory already exists, it is renamed before the restoration occurs. For more information about where to run this command, see the Using command line tools article.

In a UNIX or Linux environment, the **backupConfig** command does not save file permissions or ownership information. The **restoreConfig** command uses the current umask and effective user ID (EUID) to set the permissions and ownership when restoring a file. If it is required that the restored files have the original permissions and ownership, use the **tar** command (available on all UNIX or Linux systems) to back up and restore the configuration.

For AIX only, if you are using a logical directory for *app_server_root*/`config`, the **restoreConfig** command will not work.

If you make changes to the application files in the *app_server_root*/`installedApps` directory but do not make the same changes to the application files in the *app_server_root*/`config` directory, the changes may be overwritten if you use the **restoreConfig** command. This is known as hot deployment.

Syntax

The command syntax is as follows:

```
restoreConfig <backup_file> [options]
```

where *backup_file* specifies the file to be restored. If you do not specify one, the command will not run.

Parameters

The following options are available for the **restoreConfig** command:

-nowait
Tells the **restoreConfig** command not to stop the servers before restoring the configuration.

-quiet
Suppresses the progress information that the **restoreConfig** command prints in normal mode.

-location <directory_name>
Specifies the directory where the backup file is restored.The location defaults to the *app_server_root*/`config` directory.

-logfile <fileName>
Specifies the location of the log file to which information gets written.

-profileName
Defines the profile of the Application Server process in a multiple profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-replacelog
Replaces the log file instead of appending to the current log.

-trace
Generates trace information into the log file for debugging purposes.

-username <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -user option.

-user <name>
Specifies the user name for authentication if security is enabled in the server. Acts the same as the -username option.

-password <password>
Specifies the password for authentication if security is enabled in the server.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

The following example demonstrates correct syntax:

```
restoreConfig.sh WebSphereConfig_2006-04-22.zip
```

The following example restores the given file to the `/tmp` directory and does not stop any servers before beginning the restoration:

```
restoreConfig.sh WebSphereConfig_2006-04-22.zip -location /tmp -nostop
```

Be aware that if you restore the configuration to a directory that is different from the directory that was backed up when you performed the **backupConfig** command, you may need to manually update some of the paths in the configuration directory.

EARExpander command

Use the **EARExpander** command to expand an enterprise archive file (EAR) into a directory to run the application in that EAR file.

You can collapse a directory containing application files into a single EAR file. You can type `EARExpander` with no arguments to learn more about its options. For more information about where to run this command, see the Using command tools article.

Syntax

The command syntax is as follows:

```
EarExpander -ear earName -operationDir dirName -operation
<expand | collapse> [-expansionFlags <all|war>]
```

Parameters

The following options are available for the **EARExpander** command:

-ear
Specifies the name of the input EAR file for the expand operation or the name of the output EAR file for the collapse operation.

-operationDir
Specifies the directory where the EAR file is expanded or specifies the directory from where files are collapsed.

-operation <expand | collapse>
The `expand` value expands an EAR file into a directory structure required by the WebSphere Application Server run time. The `collapse` value creates an EAR file from an expanded directory structure.

-expansionFlags <all | war>
(Optional) The `all` value expands all files from all of the modules. The `war` value only expands the files from Web archive file (WAR) modules.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

Usage scenario

The following examples demonstrate correct syntax:

```
EARExpander -ear C:\WebSphere\AppServer\installableApps\DefaultApplication.ear
-operationDir  C:\MyApps -operation expand -expansionFlags war

EARExpander -ear C:\backup\DefaultApplication.ear
-operationDir C:\MyAppsDefaultApplication.ear -operation collapse
```

GenPluginCfg command

The GenPluginCfg command is used to regenerate the WebSphere Web server plug-in configuration file, plugin-cfg.xml.

For more information about where to run this command, see the Using command tools article.

CAUTION:
Regenerating the plug-in configuration can overwrite manual configuration changes that you might want to preserve. Before performing this task, understand its implications as described in the Communicating with Web servers topic in the *Setting up the application serving environment* **PDF.**

Another option is to issue the following command:

app_server_root/bin/GenPluginCfg.sh

Both methods for regenerating the plug-in configuration create a `plugin-cfg.xml` file in EBCDIC format, which is the proper format for execution in a z/OS environment.

You can use the -profileName option to define the profile of the application server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

Syntax

The command syntax is as follows:

```
GenPluginCfg.sh [[-option.name optionValue]...]
```

When the **GenPluginCfg** command is issued with the option -webserver.name webservrName, wsadmin generates a plug-in configuration file for the Web server. This settings in this generated configuration file are based on the list of applications that are deployed on the Web server. When this command is issued without the option -webserver.name webservrName, the plug-in configuration file is generated based on topology.

Parameters

The following options are available for the **GenPluginCfg** command:

-config.root configroot_dir
Defaults to environment variable CONFIG_ROOT.

-profileName
Defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment. The default for this option is the default profile.

-cell.name cell
Defaults to environment variable WAS_CELL.

-node.name node
Defaults to environment variable WAS_NODE.

-webserver.name webserver1
Required for creating plug-in configuration file for a given Web server.

-propagate yes/no
Applicable only when the option webserver.name is specified. Defaults to no.

-cluster.name cluster1,cluster2 | ALL
Optional list of clusters. Ignored when the option webserver.name is specified.

-server.name server1,server2
Optional list of servers. Required for single server plug-in generation. Ignored when the option webserver.name is specified.

-output.file.name file_name
Defaults to the configroot_dir/plugin-cfg.xml file. Ignored when the option webserver.name is specified.

-destination.root root
Installation root of the machine configuration is used on. Ignored when the option webserver.name is specified.

-destination.operating.system windows/unix
Operating system of the machine configuration is used on. Ignored when the option webserver.name is specified.

-debug yes/no
Defaults to no.

-help
Prints a usage statement.

-? Prints a usage statement.

Usage scenario

To generate a plug-in configuration for all of the clusters in a cell:

```
GenPluginCfg.sh -cell.name NetworkDeploymentCell
```

To generate a plug-in configuration for a single server:

```
GenPluginCfg.sh -cell.name BaseApplicationServerCell -node.name appServerNode -server.name appServerName
```

To generate a plug-in configuration file for a Web server:

```
GenPluginCfg.sh -cell.name BaseApplicationServerCell -node.name webserverNode -webserver.name webserverName
```

versionInfo command

The **versionInfo** command generates a report from data extracted from XML files in the properties/version folder. The report includes a list of changed components and installed or uninstalled maintenance packages.

Product version information

The versionInfo tool displays important data about the product and its installed components, such as the build version and build date. History information for installation and removal of maintenance packages also displays in the report. This tool is particularly useful when working with support personnel to determine the cause of any problem.

Product version reports

The following report generation scripts extract data from XML data files in the properties/version folder:

- versionInfo script

  Lets you use parameters to create a version report.
- genVersionReport script

  Generates the versionReport.html report file in the current working directory, which is usually the bin directory.

Location of the command file

The versionInfo command is a script.

Syntax for the versionInfo command

Parameters

-? or /?
Displays command syntax.

-components
Adds a list of installed components to the report.

-componentDetail
Adds details about installed components to the report.

-file *file_name*
Specifies the output file name. The report goes to standard output (stdout) by default.

-format text | html
Selects the format of the report. The default is ″text″.

-help or /help
Displays command syntax.

-long
Creates the long version of the report.

-maintenancePackages
Adds a list of applied maintenance packages to the report.

-maintenancePackageDetail
Adds details about an applied maintenance package to the report.

-usage
Displays command syntax.

Report description

The versionInfo command reports the following information:

Installation information

Displays the following general information about the current installation:

- Report date and time - The date and time that the report was generated. The timestamp is formatted according to the current locale.
- Product directory - The file path to the installation root directory defined by the WAS_HOME environment variable.
- Version directory - The file path of the version directory of the current IBM WebSphere Application Server - ND installation.
- DTD directory - The file path of the DTD directory of the current IBM WebSphere Application Server - ND installation.
- Log directory - The file path of the log directory of the current IBM WebSphere Application Server - ND installation. The maintenance package log files are in the directory.
- Backup directory - The file path of the backup directory of the current IBM WebSphere Application Server - ND installation. The backup files generated during the installation of maintenance packages are in this directory.
- TMP directory - The file path of the temporary directory of the current machine.

Product list information

Displays a list of installed WebSphere products:

- Product ID - The product ID of the installed product.
- Status - The status of the product, either installed or uninstalled.

Installed product information

This information and the other information topic descriptions are hierarchal for each installed product, component, component update, installed maintenance package, included APARs, and component updates.

This section of the report displays the following information:

- Name - The name of the installed product.
- Version - The current version of the product. Installing or uninstalling fix packs or refresh packs modifies this version.
- ID - The product ID of the product installed, such as BASE, BASETRIAL, ND, EXPRESS, EXPRESSTRIAL, embeddedEXPRESS, IHS, XD, PLG, or CLIENT.
- Build level - The build level of the installed product.
- Build date - The build date of the installed product.

Installed component information: Displays the following component-level information of the installed component from the .component file under the /properties/version directory:

- Component name - The name of the installed component
- Spec version - The spec version of the current component
- Build level - The build level of the current component
- Build date - The build date of the current component

Installed component update information: Displays the general maintenance package information:

- Component name - the name of the installed component
- Update type - All interim fixes, fix packs and refresh packs are maintenance packages. The update type field is always set to *maintenance package*.
- Maintenance package ID - The ID of the maintenance package that is responsible for the current maintenance level of the product.
- Update effect - The updated action taken on a particular component. The default effect is *replace*.
- Log file name - The file path of the log file that records the maintenance actions for the current maintenance level.
- Backup file name - The file path of the backup file generated during the installation of the maintenance package. Not applicable if the component is restored as a result of an uninstall of a maintenance package.
- Timestamp - The time at which the component is updated. The timestamp is stated in GMT offset values.

Installed maintenance package information: Displays the general maintenance package information:

- Maintenance Package ID - the maintenance package ID
- Description - the description of the maintenance package
- Build Date - the build date of this maintenance package

Included APARs information: Displays the list of APARs fixed by this maintenance package.

Component updates information: Displays the following information about each component that is updated by the installed maintenance package:

- Component name - The name of the installed component.

- Updated effect - The update action taken on the component by the maintenance package. The default action is *replace*.
- Timestamp - The time at which the component update is installed (GMT offset).

Sample versionInfo report

When the WebSphere Application Server product has no interim fixes or fix packs applied, the genVersionReport.bat script creates the following information in the versionReport.html report file, which is edited to show only the first few components:

```
IBM WebSphere Application Server Product Installation Status Report

----------------------------------------

 Report at date and time 2005-05-18 15:58:40-0400

 Installation
----------------------------------------
 Product Directory:   /opt/WebSphere/AppServer
 Version Directory:   /opt/WebSphere/AppServer/properties/version
 DTD Directory:       /opt/WebSphere/AppServer/properties/version/dtd
 Log Directory:       /opt/WebSphere/AppServer/logs/update
 Backup Directory:    /opt/WebSphere/AppServer/properties/version/backup
 TMP Directory:       /tmp


 Installation Platform
----------------------------------------
 Name                 IBM WebSphere Application Server
 Version              6.0


 Product List
----------------------------------------
  BASE                installed

Installed Product
----------------------------------------

Name                 IBM WebSphere Application Server
Version              6.0.1
ID                   BASE
Build Level          m0451.03
Build Date           12/19/2004



Installed Component
--------------------------------------------------
Component Name:          activity.impl
Spec Version:                 6.0
Build Version:                m0451.03
Build Date:                   12/19/04

Installed Component Update
--------------------------------------------------
Component Name:                    activity.impl
Update Type:                       maintenance package
Maintenance Package ID:  was60_fp1_linux
Update Effect:                     replace
Log File Name:
  /opt/WebSphere/AppServer/logs/update/was60_fp1_linux.install/updatelog.txt
Backup File Name:
  /opt/WebSphere/AppServer/properties/version/backup/was60_fp1_linux.pak
Timestamp:                            2004-12-17 18:24:34-0500
```

```
Installed Component Update
-----------------------------------------------------
Component Name:                   activity.impl
Update Type:                      maintenance package
Maintenance Package ID: was60_fp2
Update Effect:                    replace
Log File Name:
  /opt/WebSphere/AppServer/logs/update/was60_fp2_linux.install/updatelog.txt
Backup File Name:
  /opt/WebSphere/AppServer/properties/version/backup/was60_fp2_linux.pak
Timestamp:                           2004-12-19 06:24:34-0500


Installed Component
-----------------------------------------------------
Component Name:   activity.session
Spec Version:     6.0
Build Level:      m0451.03
Build Date:       12/19/04

Installed Component Update
-----------------------------------------------------
Component Name:          activity.session
Update Type:                      maintenance package
Maintenance Package ID:  was60_fp2
Update Effect:                    replace
Log File Name:
  /opt/WebSphere/AppServer/logs/update/was60_fp2_linux.install/updatelog.txt
Backup File Name:
  /opt/WebSphere/AppServer/properties/version/backup/was60_fp2_linux.pak
Timestamp:               2004-12-19 06:24:34-0500

Installed Maintenance Package
-----------------------------------------------------
Maintenance Package ID:   was60_fp1_linux
Description:              IBM WebSphere Application Server,
                          Version 6.0.1 Fix Pack for Linux
Build Date:                          12/17/2004

Included Apars
-----------------------------------------------------
PQ12345

Component Updates
-----------------------------------------------------
activity       updated
installed on 2004-12-17 06:24:30-0500

Installed Maintenance Package
-----------------------------------------------------
Maintenance Package ID:   was60_fp2
Description:              IBM WebSphere Application Server,
                          Version 6.0.2 Fix Pack for Linux
Build Date:               12/19/2004

Included Apars
-----------------------------------------------------
PQ12345
PQ23456
PQ34567

Component Updates
-----------------------------------------------------
activity       updated
installed on 2004-12-19 06:24:30-0500
activity.impl       updated
installed on 2004-12-19 06:24:30-0500
```

```
------------------------------------------------
End Installation Status Report
------------------------------------------------
```

Summary of Version 6 changes for the versionInfo command

Changes are in two areas: command syntax and report information.

Changes to command syntax

The following changes are in effect:

- Version 6 replaces the terms *efixes* and *ptfs* with *maintenancePackages* to describe a specific maintenance package. This matches the terminology used in the Version 6 update installer application.
- Version 6 replaces the terms *efixesDetail* and *ptfDetail* with *maintenancePackageDetail* to describe the detailed information that relates to a specific maintenance package.

Changes to report information

The following changes are in effect:

- Version 6 replaces the term *technology list* with *product list*. The technology list displays the list of installed products in the current WAS_HOME directory.
- Version 6 uses an update type value of *maintenance package*.
- Version 6 replaces the term *PTF ID* with *Maintenance Package ID*. This change is consistent with the terminology used in the Update Installer for WebSphere Software.
- Version 6 removes *IsRequired*, *IsOptional*, *IsExternal*, and *IsCustom* from the *Installed component update* section of the report. The older terminology is not supported per component by the Version 6 Update Installer for WebSphere Software.
- Version 6 replaces the term *Install Date* with *Timestamp* for consistency with the History report.
- Version 6 replaces the *Installed PTF* section of the report with the *Installed Maintenance Package* section.
- Version 6 removes the term *Exposition* because it is similar to the term *Description*.
- Version 6 removes the *Build Version* field from the *Installed Maintenance Package* section.
- Version 6 removes the *Supported Platforms* section of the report.
- Version 6 replaces the *Included Fixes* section of the report with the with by *Included APARs* section, which lists the APARs.
- Version 6 removes the *Custom Properties* section of the report.

---

genVersionReport command

The **genVersionReport** command uses the **versionInfo** command to generate the versionReport.html report file in the current working directory, which is usually the bin directory. The report includes a list of changed components and installed or uninstalled maintenance packages. The genVersionReport script invokes the versionInfo script specifying the correct parameters to place the information generated into an HTML file in the current working directory.

Product version information

The versionInfo tool displays important data about the product and its installed components, such as the build version and build date. History information for installation and removal of maintenance packages also displays in the report. This tool is particularly useful when working with support personnel to determine the cause of any problem.

Product version reports

The following report generation scripts extract data from XML data files in the properties/version folder:

- versionInfo script

  Use the versionInfo command to specify your own report parameters when creating a customized version report.
- genVersionReport script

  Use the genVersionReport command to generate the versionReport.html report file in the current working directory, which is usually the bin directory. The report includes the list of components, fixes, fix packs, and refresh packs.

Location of the command file

The genVersionReport command is a script.

The command file is named genVersionReport in the bin directory of the *app_server_root* directory.

Syntax for the genVersionReport command

The command syntax is:

Issue the command from the bin directory of the *app_server_root* directory.

Report description

The versionInfo command reports the following information:

Installation information

Displays the following general information about the current installation:

- Report date and time - The date and time that the report was generated. The timestamp is formatted according to the current locale.
- Product directory - The file path to the installation root directory defined by the WAS_HOME environment variable.
- Version directory - The file path of the version directory of the current IBM WebSphere Application Server - ND installation.
- DTD directory - The file path of the DTD directory of the current IBM WebSphere Application Server - ND installation.
- Log directory - The file path of the log directory of the current IBM WebSphere Application Server - ND installation. The maintenance package log files are in the directory.
- Backup directory - The file path of the backup directory of the current IBM WebSphere Application Server - ND installation. The backup files generated during the installation of maintenance packages are in this directory.
- TMP directory - The file path of the system temporary directory.

Product list information

Displays a list of installed WebSphere products:

- Product ID - The product ID of the installed product.
- Status - The status of the product, either installed or uninstalled.

Installed product information

This information and the other information topic descriptions are hierarchal for each installed product, component, component update, installed maintenance package, included APARs, and component updates.

This section of the report displays the following information:

- Name - The name of the installed product.
- Version - The current version of the product. Installing or uninstalling fix packs or refresh packs modifies this version.

- ID - The product ID of the product installed, such as BASE, BASETRIAL, ND, EXPRESS, EXPRESSTRIAL, embeddedEXPRESS, IHS, XD, PLG, or CLIENT.
- Build level - The build level of the installed product.
- Build date - The build date of the installed product.

Installed component information: Displays the following component-level information of the installed component from the .component file in the *app_server_root*/properties/version directory:
- Component name - The name of the installed component
- Spec version - The spec version of the current component
- Build level - The build level of the current component
- Build date - The build date of the current component

Installed component update information: Displays the general maintenance package information:
- Component name - the name of the installed component
- Update type - All interim fixes, fix packs and refresh packs are maintenance packages. The update type field is always set to *maintenance package*.
- Maintenance package ID - The ID of the maintenance package that is responsible for the current maintenance level of the product.
- Update effect - The updated action taken on a particular component. The default effect is *replace*.
- Log file name - The file path of the log file that records the maintenance actions for the current maintenance level.
- Backup file name - The file path of the backup file generated during the installation of the maintenance package. Not applicable if the component is restored as a result of an uninstall of a maintenance package.
- Timestamp - The time at which the component is updated. The timestamp is stated in GMT offset values.

Installed maintenance package information: Displays the general maintenance package information:
- Maintenance Package ID - the maintenance package ID
- Description - the description of the maintenance package
- Build Date - the build date of this maintenance package

Included APARs information: Displays the list of APARs fixed by this maintenance package.

Component updates information: Displays the following information about each component that is updated by the installed maintenance package:
- Component name - The name of the installed component.
- Updated effect - The update action taken on the component by the maintenance package. The default action is *replace*.
- Timestamp - The time at which the component update is installed (GMT offset).

Summary of Version 6 changes for the VersionInfo command

Changes are in two areas: command syntax and report information.

Changes to command syntax

The following changes are in effect:
- Version 6 replaces the terms *efixes* and *ptfs* with *maintenancePackages* to describe a specific maintenance package. This matches the terminology used in the Version 6 update installer application.
- Version 6 replaces the terms *efixesDetail* and *ptfDetail* with *maintenancePackageDetail* to describe the detailed information that relates to a specific maintenance package.

Changes to report information for Version 6.x

The following changes are in effect:

- Version 6 replaces the term *technology list* with *product list*. The technology list displays the list of installed products in the current WAS_HOME directory.
- Version 6 uses an update type value of *maintenance package*.
- Version 6 replaces the term *PTF ID* with *Maintenance Package ID*. This change is consistent with the terminology used in the Update Installer for WebSphere Software.
- Version 6 removes *IsRequired*, *IsOptional*, *IsExternal*, and *IsCustom* from the *Installed component update* section of the report. The older terminology is not supported per component by the Version 6 Update Installer for WebSphere Software.
- Version 6 replaces the term *Install Date* with *Timestamp* for consistency with the History report.
- Version 6 replaces the *Installed PTF* section of the report with the *Installed Maintenance Package* section.
- Version 6 removes the term *Exposition* because it is similar to the term *Description*.
- Version 6 removes the *Build Version* field from the *Installed Maintenance Package* section.
- Version 6 removes the *Supported Platforms* section of the report.
- Version 6 replaces the *Included Fixes* section of the report with the with by *Included APARs* section, which lists the APARs.
- Version 6 removes the *Custom Properties* section of the report.

historyInfo command

The **historyInfo** command generates a report from data extracted from XML files in the properties/version folder and the properties/version/history folder. The report includes a list of changed components and a history of installed or uninstalled maintenance packages.

Product history information

The historyInfo tool displays important data about the product and its installed components, such as the build version and build date. History information for installation and removal of maintenance packages also displays in the report. This tool is particularly useful when working with support personnel to determine the cause of any problem.

Product history reports

The following report generation scripts extract data from XML data files in the properties/version folder and the properties/version/history folder:

- historyInfo script

  Lets you use parameters to create a history report.
- genHistoryReport script

  Generates the historyReport.html report file in the current working directory, which is usually the bin directory.

Location of the command file

The historyInfo command is a script.

Syntax for the historyInfo command

Parameters

-? or /?
Displays command syntax.

-component *component_name*
Specifies the name of a component. When specified, the product history report displays events for only the named component. When not specified, the report displays events for all components.

-file *file_name*
Specifies the output file name. The report goes to standard output (stdout) by default.

-format text | html
Selects the format of the report. The default is ″text″.

-help or /help
Displays command syntax.

-maintenancePackageID *ID_of_maintenance_package*
Specifies the ID of the interim fix, fix pack, or refresh pack. When specified, the product history report displays events for only the named maintenance package. When not specified, the report displays events for all maintenance packages.

-usage
Displays command syntax.

Report description

The historyInfo command reports the following information:

Installation information

Displays the following general information about the current installation:

- Report date and time - The date and time that the report was generated. The timestamp is formatted according to the current locale.
- Product directory - The file path to the installation root directory of the product.
- Version directory - The file path of the version directory of the current WAS installation.
- DTD directory - The file path of the DTD directory of the current installation.
- Log directory - The file path of the log directory of the current installation. The maintenance package log files are in the directory.
- Backup directory - The file path of the backup directory of the current WAS installation. The backup files generated during the installation of maintenance packages are in this directory.
- TMP directory - The file path of the temporary directory of the current machine.
- History directory - The file path of the history directory of the current installation. The history files are in the directory.
- History File - The file path of the event.history file.

Installation event information

Displays the list of installed maintenance packages (interim fix, fix pack, and refresh pack) and the following related information:

- Maintenance package ID - The ID of the maintenance package.
- Action - The action taken with this maintenance package, which is either *install* or *uninstall*.
- Package file name - The file name of the maintenance package that was installed.
- Log file name - The file path of the log file generated during the installation or removal of the maintenance package.

- Backup file name - The file path of the backup file generated during the installation of the maintenance package. This field does not apply for an uninstall action.
- Timestamp - The time when the maintenance action (install or uninstall) occurs. The time is stated in relation to GMT.
- Result - The result of the installation or removal action. The result is either success, partial success, or failure.

Component installation event information

Displays the following component-level information of the event for the current maintenance package:

- Maintenance package ID - The ID of the maintenance package to which this particular installation event belongs.
- Component name - The name of the current component.
- Action - The action taken on this component due to the action of the current maintenance package, either install or uninstall
- Update action - The updated action taken on this component. By default, the update action for an installation action is *replace*.
- Timestamp - The time at which the action occurs for the maintenance package (GMT offset values).
- Result - The result of the install or uninstall action. The result is either success, partial success or failure.

The event.history file

The historyInfo command also generates the event.history file. This file represents the raw data of the history report information. The following example of an event.history file corresponds to the history report in the preceding example:

```
<!DOCTYPE event-history SYSTEM "eventHistory.dtd">
<event-history>
  <update-event
    event-type="ptf"
    id="was60_fp1_linux"
   update-action="install"
   primary-content="was60_fp1_linux.pak"
   update-type="replace"
   log-name=
      "/opt/WebSphere/AppServer/logs/update/was60_fp1_linux.install/updatelog.txt"
   backup-name=
      "/opt/WebSphere/AppServer/properties/version/backup/was60_fp1_linux.pak"
    start-time-stamp="2004-12-14 06:15:14-0500"
    result="success">
    <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id="activity"
      update-action="install"
      update-type="replace"
      start-time-stamp="2004-12-14 06:15:14-0500"
      result="success">
   </update-event>
   <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id=" activity.impl"
      update-action="install"
      update-type="replace"
      start-time-stamp="2004-12-14 06:15:14-0500"
      result="success">
   </update-event>
 </update-event>
 <update-event
    event-type="ptf"
```

```
   id="was60_fp2"
  update-action="install"
  primary-content="was60_fp1_linux.pak"
  update-type="replace"
  log-name="/opt/WebSphere/AppServer/logs/update/was60_fp2.install/updatelog.txt"
  backup-name="/opt/WebSphere/AppServer/properties/version/backup/was60_fp2.pak"
    start-time-stamp="2004-12-14 10:25:34-0500"
    result="partialSuccess">
   <update-event
     event-type="component"
     parent-id="was60_fp2"
     id="activity"
     update-action="install"
     update-type="replace"
    start-time-stamp="2004-12-14 10:25:34-0500"
    result="partialSuccess">
  </update-event>
  <update-event
     event-type="component"
     parent-id="was60_fp2"
     id=" activity.impl"
     update-action="install"
     update-type="replace"
     start-time-stamp="2004-12-14 10:25:34-0500"
     result="partialSuccess">
  </update-event>
</update-event>
<update-event
   event-type="ptf"
   id="was60_fp2"
  update-action="uninstall"
    primary-content=" was60_fp2.pak"
  update-type="replace"
  log-name=
     "/opt/WebSphere/AppServer/logs/update/was60_fp2.uninstall/updatelog.txt"
  backup-name="not applicable"
   start-time-stamp="2004-12-18 17:29:12-0500"
   result="partialSuccess">
   <update-event
     event-type="component"
     parent-id="was60_fp2"
     id="activity"
     update-action="uninstall"
     update-type="replace"
    start-time-stamp="2004-12-18 17:29:12-0500"
     result="partialSuccess">
  </update-event>
  <update-event
     event-type="component"
     parent-id="was60_fp2"
     id=" activity.impl"
     update-action="uninstall"
     update-type="replace"
    start-time-stamp="2004-12-18 17:29:12-0500"
     result="partialSuccess">
  </update-event>
</update-event>
<update-event
   event-type="ptf"
   id="was60_fp1_linux"
  update-action="uninstall"
    primary-content=" was60_fp1_linux.pak"
    update-type="replace"
  log-name=
     "/opt/WebSphere/AppServer/logs/update/was60_fp1_linux.install/updatelog.txt"
  backup-name="not applicable"
   start-time-stamp="2004-12-23 15:15:14-0500"
```

```
    result="faiurel">
    <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id="activity"
      update-action="uninstall"
      update-type="replace"
      start-time-stamp="2004-12-23 15:15:14-0500"
      result="failure">
   </update-event>
   <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id=" activity.impl"
      update-action="uninstall"
      update-type="replace"
      start-time-stamp="2004-12-23 15:15:14-0500"
      result="failure">
   </update-event>
  </update-event>
</event-history>
```

Summary of Version 6 changes for the historyInfo command

Changes are in three areas: command syntax, report information, and the event.history file.

Changes to command syntax

Version 6 replaces the term *updateID* with *maintenancePackageID* to describe a specific maintenance package. This matches the terminology used in the Version 6 update installer application.

Changes to report information

The following changes are in effect:

- Version 6 replaces the term *PTF ID* with *Maintenance package ID*. This change is consistent with the terminology used in the Update Installer for WebSphere Software.
- Version 6 replaces the term *Primary content* with *Package file name*.
- Version 6 replaces the terms *Start Time* and *End Time* with a *Timestamp* for the completion of the event.
- Version 6 removes the terms *IsExternal*, and *IsCustom*.
- Version 6 removes the *Result Message* section of the report because the information is reported in the *Result and Action* section.
- Version 6 sets the value of *Backup file name* to ″not applicable″ if the update action is *uninstall*. No backup file is generated when a maintenance package is uninstalled.
- Version 6 adds a new result state, partial success. The Version 6 Update Installer for WebSphere Software can generate a partial success.

Changes to the event.history file

The following changes are in effect:

- Version 6 replaces the terms *Start Time* and *End Time* with a *Timestamp* for the completion of the event.
- Version 6 removes the terms *IsRequired*, *IsExternal*, *IsCustom*, *root-property-file*, *root-property-name*, and *root-property-value*, which are not supported by the Update Installer for WebSphere Software.
- Version 6 removes the *Result Message* section of the report because the information is reported in the *Result and Action* section.
- Version 6 sets the value of *update-event* to ″replace″.
- Version 6 sets the value of *Backup file name* to ″not applicable″ if the update action is *uninstall*. No backup file is generated when a maintenance package is uninstalled.

- Version 6 adds a new result state, partial success. The Version 6 Update Installer for WebSphere Software can generate a partial success.

genHistoryReport command

The **genHistoryReport** command generates the historyReport.html report file in the current working directory, which is usually the bin directory. The report includes a list of changed components and installed or uninstalled maintenance packages. The genHistoryReport script invokes the historyInfo script specifying the correct parameters to place the information generated into an HTML file in the current directory.

Product history information

The historyInfo tool displays historical data about the product and the installation and removal of maintenance packages for the product. This tool is particularly useful when working with support personnel to determine the cause of any problem.

Product history reports

The following report generation scripts extract data from XML data files in the properties/version folder and the properties/version/history folder:

- historyInfo script

  Lets you use parameters to create a history report.
- genHistoryReport script

  Generates the historyReport.html report file in the current working directory, which is usually the bin directory. The report includes a list of components, fixes, fix packs, and refresh packs.

Location of the command file

The command file is a script.

Syntax for the genHistoryReport command

Report description

The historyInfo command generates the report. The genHistoryReport command calls the historyInfo command with a set of report parameters that reports the following information:

Installation information

Installation information displays the following general information about the current installation:

- Report date and time - The date and time that the report was generated. The timestamp is formatted according to the current locale.
- Product directory - The file path to the installation root directory of the product.
- Version directory - The file path of the version directory of the current WAS installation.
- DTD directory - The file path of the DTD directory of the current installation.
- Log directory - The file path of the log directory of the current installation. The maintenance package log files are in the directory.
- Backup directory - The file path of the backup directory of the current WAS installation. The backup files generated during the installation of maintenance packages are in this directory.
- TMP directory - The file path of the temporary directory of the current machine.
- History directory - The file path of the history directory of the current installation. The history files are in the directory.
- History File - The file path of the event.history file.

Installation event information

Installation event information displays the list of installed maintenance packages (interim fix, fix pack, and refresh pack) and the following related information:

- Maintenance package ID - The ID of the maintenance package.
- Action - The action taken with this maintenance package, which is either *install* or *uninstall*.
- Package file name - The file name of the maintenance package that was installed.
- Log file name - The file path of the log file generated during the installation or removal of the maintenance package.
- Backup file name - The file path of the backup file generated during the installation of the maintenance package. This field does not apply for an uninstall action.
- Timestamp - The time when the maintenance action (install or uninstall) occurs. The time is stated in relation to GMT.
- Result - The result of the installation or removal action. The result is either success, partial success, or failure.

Component installation event information

Component installation event information displays the following component-level information of the event for the current maintenance package:

- Maintenance package ID - The ID of the maintenance package to which this particular installation event belongs.
- Component name - The name of the current component.
- Action - The action taken on this component due to the action of the current maintenance package, either install or uninstall
- Update action - The updated action taken on this component. By default, the update action for an installation action is *replace*.
- Timestamp - The time at which the action occurs for the maintenance package (GMT offset values).
- Result - The result of the install or uninstall action. The result is either success, partial success or failure.

The event.history file

The genHistoryReport command also generates the event.history file. This file represents the raw data of the history report information. The following example of an event.history file corresponds to the history report in the preceding example:

```
<!DOCTYPE event-history SYSTEM "eventHistory.dtd">
<event-history>
  <update-event
    event-type="ptf"
    id="was60_fp1_linux"
   update-action="install"
   primary-content="was60_fp1_linux.pak"
   update-type="replace"
   log-name=
      "/opt/WebSphere/AppServer/logs/update/was60_fp1_linux.install/updatelog.txt"
   backup-name=
      "/opt/WebSphere/AppServer/properties/version/backup/was60_fp1_linux.pak"
    start-time-stamp="2004-12-14 06:15:14-0500"
    result="success">
    <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id="activity"
      update-action="install"
      update-type="replace"
      start-time-stamp="2004-12-14 06:15:14-0500"
      result="success">
   </update-event>
```

```
  <update-event
     event-type="component"
     parent-id="was60_fp1_linux"
     id=" activity.impl"
     update-action="install"
     update-type="replace"
     start-time-stamp="2004-12-14 06:15:14-0500"
     result="success">
  </update-event>
</update-event>
<update-event
   event-type="ptf"
   id="was60_fp2"
  update-action="install"
  primary-content="was60_fp1_linux.pak"
  update-type="replace"
  log-name="/opt/WebSphere/AppServer/logs/update/was60_fp2.install/updatelog.txt"
  backup-name="/opt/WebSphere/AppServer/properties/version/backup/was60_fp2.pak"
    start-time-stamp="2004-12-14 10:25:34-0500"
    result="partialSuccess">
   <update-event
     event-type="component"
     parent-id="was60_fp2"
     id="activity"
     update-action="install"
     update-type="replace"
    start-time-stamp="2004-12-14 10:25:34-0500"
    result="partialSuccess">
  </update-event>
  <update-event
     event-type="component"
     parent-id="was60_fp2"
     id=" activity.impl"
     update-action="install"
     update-type="replace"
     start-time-stamp="2004-12-14 10:25:34-0500"
     result="partialSuccess">
  </update-event>
</update-event>
<update-event
   event-type="ptf"
   id="was60_fp2"
  update-action="uninstall"
    primary-content=" was60_fp2.pak"
  update-type="replace"
  log-name=
     "/opt/WebSphere/AppServer/logs/update/was60_fp2.uninstall/updatelog.txt"
  backup-name="not applicable"
   start-time-stamp="2004-12-18 17:29:12-0500"
   result="partialSuccess">
   <update-event
     event-type="component"
     parent-id="was60_fp2"
     id="activity"
     update-action="uninstall"
     update-type="replace"
    start-time-stamp="2004-12-18 17:29:12-0500"
     result="partialSuccess">
  </update-event>
  <update-event
     event-type="component"
     parent-id="was60_fp2"
     id=" activity.impl"
     update-action="uninstall"
     update-type="replace"
    start-time-stamp="2004-12-18 17:29:12-0500"
     result="partialSuccess">
```

```
  </update-event>
 </update-event>
 <update-event
    event-type="ptf"
    id="was60_fp1_linux"
   update-action="uninstall"
     primary-content=" was60_fp1_linux.pak"
     update-type="replace"
   log-name=
      "/opt/WebSphere/AppServer/logs/update/was60_fp1_linux.install/updatelog.txt"
   backup-name="not applicable"
    start-time-stamp="2004-12-23 15:15:14-0500"
    result="faiurel">
    <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id="activity"
      update-action="uninstall"
      update-type="replace"
      start-time-stamp="2004-12-23 15:15:14-0500"
      result="failure">
   </update-event>
   <update-event
      event-type="component"
      parent-id="was60_fp1_linux"
      id=" activity.impl"
      update-action="uninstall"
      update-type="replace"
      start-time-stamp="2004-12-23 15:15:14-0500"
      result="failure">
   </update-event>
  </update-event>
</event-history>
```

Summary of Version 6 changes for the historyInfo command

Changes are in three areas: command syntax, report information, and the event.history file.

Changes to command syntax

Version 6 replaces the term *updateID* with *maintenancePackageID* to describe a specific maintenance package. This matches the terminology used in the Version 6 update installer application.

Changes to report information

The following changes are in effect:

- Version 6 replaces the term *PTF ID* with *Maintenance package ID*. This change is consistent with the terminology used in the Update Installer for WebSphere Software.
- Version 6 replaces the term *Primary content* with *Package file name*.
- Version 6 replaces the terms *Start Time* and *End Time* with a *Timestamp* for the completion of the event.
- Version 6 removes the terms *IsExternal*, and *IsCustom*.
- Version 6 removes the *Result Message* section of the report because the information is reported in the *Result and Action* section.
- Version 6 sets the value of *Backup file name* to ″not applicable″ if the update action is *uninstall*. No backup file is generated when a maintenance package is uninstalled.
- Version 6 adds a new result state, partial success. The Version 6 Update Installer for WebSphere Software can generate a partial success.

Changes to the event.history file

The following changes are in effect:

- Version 6 replaces the terms *Start Time* and *End Time* with a *Timestamp* for the completion of the event.
- Version 6 removes the terms *IsRequired*, *IsExternal*, *IsCustom*, *root-property-file*, *root-property-name*, and *root-property-value*, which are not supported by the Update Installer for WebSphere Software.
- Version 6 removes the *Result Message* section of the report because the information is reported in the *Result and Action* section.
- Version 6 sets the value of *update-event* to ″replace″.
- Version 6 sets the value of *Backup file name* to ″not applicable″ if the update action is *uninstall*. No backup file is generated when a maintenance package is uninstalled.
- Version 6 adds a new result state, partial success. The Version 6 Update Installer for WebSphere Software can generate a partial success.

Chapter 6. Using MVS console commands

This topics discusses the MVS console commands that you can use to start, stop, and modify WebSphere Application Server for z/OS nodes and servers.

This topic assumes a basic familiarity with MVS console commands. See the *z/OS MVS System Commands* manual at http://www.ehone.ibm.com/public/applications/publications/cgibin/pbi.cgi for information on how to use MVS operator commands.

You can issue commands from the MVS console yourself, or optionally use standard console automation products to automate WebSphere Application Server for z/OS operations. Products such as Netview are presented copies of messages that are to be displayed on the MVS console. These automation products can also enter commands into the system using a virtual MVS console as a source.

You can use the MVS console commands to start, stop, and modify WebSphere Application Server for z/OS nodes and servers.

- Start the application server, the deployment manager, or the node agent.
- Stop the application server, the deployment manager, or the node agent.
- Modify Application Server operations.
- Monitor your Application Server through display commands.

Depending on which commands you issued, you might have stopped or started the Application Server, stopped or started the deployment manager, stopped or started the node agent, modified the Application Server operations, or monitored your Application Server.

START command

The **START** command reads the configuration file for the specified server process and starts the server, deployment manger, or node agent. Depending on the options you specify, you can launch a new Java Virtual Machine (JVM) API to run the server process or write the launch command data to a file. You can run this command from the MVS console of a WebSphere Application Server installation or a network deployment installation.

The following examples demonstrate correct syntax:

```
START appserver_proc_name,JOBNAME=server_short_name,
       ENV=cell_short_name.node_short_name.server_short_name
START dmgr_proc_name,JOBNAME=server_short_name,
       ENV=cell_short_name.node_short_name.server_short_name
START nodeagent_proc_name,JOBNAME=server_short_name,
       ENV=cell_short_name.node_short_name.server_short_name
```

You must enter this command on a single line. It is split here for display purposes.

STOP command

The **STOP** command reads the configuration file for the specified server process and stops the server, deployment manger, or node agent. Depending on the options you specify, you can launch a new Java Virtual Machine (JVM) API to run the server process or write the launch command data to a file. You can run this command from the MVS console of a WebSphere Application Server installation or a network deployment installation.

The following examples demonstrate correct syntax:

```
STOP appserver_proc_name,JOBNAME=server_short_name,
       ENV=cell_short_name.node_short_name.server_short_name
STOP dmgr_proc_name,JOBNAME=server_short_name,
       ENV=cell_short_name.node_short_name.server_short_name
STOP nodeagent_proc_name,JOBNAME=server_short_name,
       ENV=cell_short_name.node_short_name.server_short_name
```

You must enter this command on a single line. It is split here for display purposes.

Modify command

Use the **modify** command from the MVS console to dynamically modify WebSphere Application Server for z/OS operations.

You can use the modify command to display status of various server components and activities, including:
- Active controllers
- Trace settings
- Servants
- Sessions
- JVM Heap
- Java trace

```
f <server>, options
```

The first argument is required. The argument is the name of the server for which the modify command will be directed.

Parameters

The options for the **modify** command follow:

CANCEL
Used to cancel a server. *server* refers to the server short name.

You can specify the following options:

ARMRESTART
Specify if you are using ARM and want ARM to restart the server after it terminates. If you don't specify the ARMRESTART option on the CANCEL parameter, ARM will not restart the server.

HELP Get help for the CANCEL syntax.

Note: You cannot use the CANCEL parameter to cancel a cluster from the MVS console. Instead, you must cancel each of the servers that make up the cluster.

TRACEALL=n
Use TRACEALL to establish a general trace level for the server.

Valid trace levels are 0 (none), 1 (exception), 2 (basic), and 3 (detailed tracing). Under normal conditions and in production, use 1 (exception).

Note: Be careful when using a level of 3 (detailed for all components) because it can potentially yield more data than can be handled reasonably.

TRACEBASIC=n
Specify the WebSphere Application Server for z/OS components for which you want to switch on a basic level of tracing.

This command has the ability to override a different tracing level established by TRACEALL for those components.

Note: Do not change this variable unless directed by IBM service personnel.

You can specify one or more of the following options for either TRACEBASIC or TRACEDETAIL:

0 RAS

1 Common Utilities

3 COMM

4 ORB

6 OTS

7 Shasta

9 OS/390 Wrappers

A Daemon

E Security

F Externalization

J JRAS (internal tracing-via direction from IBM support)

L J2EE

TRACEDETAIL=n
Specify the WebSphere Application Server for z/OS components for which you want to switch on a detailed level of tracing.

This command activates the most detailed tracing for the specified WebSphere Application Server for z/OS components and overrides different settings inTRACEALL. The selected components *(n,...)* are identified by their component-ID (valid values are the same as for TRACEBASIC above. Subcomponents, specified by numbers, receive detailed traces. Other parts of WebSphere Application Server for z/OS receive tracing as specified on the TRACEALL variable.

Note: Do not change this variable unless directed by IBM service personnel.

TRACESPECIFIC=xxyyyzzz
Specifies tracing overrides for specific WebSphere Application Server for z/OS trace points.

Trace points are specified by 8-digit, hexadecimal numbers. To specify more than one trace point, use parentheses and separate the numbers with commas. You can also specify an environment variable name by enclosing the name in single quotes. The value of the environment variable will be handled as if you had specified that value on TRACESPECIFIC.

Note: Do not use TRACESPECIFIC unless directed by IBM service personnel.

TRACE_EXCLUDE_SPECIFIC=xxyyyzzz
Specifies trace points to exclude.

Trace points to exclude are specified by 8-digit, hexadecimal numbers. To specify more than one trace point, use parentheses and separate the numbers with commas. You can also specify an environment variable name by enclosing the name in single quotes. The value of the environment variable will be handled as if you had specified that value on TRACE_EXCLUDE_SPECIFIC. You can use TRACE_EXCLUDE_SPECIFIC as a mask to turn off otherwise-on traces. For example, use the TRACESPECIFIC command to turn on tracing for a whole part and then use TRACE_EXCLUDE_SPECIFIC to turn off one trace within that part.

Note: Do not use TRACE_EXCLUDE_SPECIFIC unless directed by IBM service personnel.

TRACEINIT
Reset to the initial trace settings.

TRACENONE
Turns off all trace settings.

TRACETOSYSPRINT={YES|NO}
Select whether or not to send the trace to sysprint.

YES specifies to send the trace to sysprint and NO stops sending the trace to sysprint.

TRACETOTRCFILE={YES|NO}
Select whether or not to direct the trace to the TRCFILE DD card.

YES specifies to send the trace to the TRCFILE DD card and NO stops sending the trace to the TRCFILE DD card.

TRACEJAVA
Modify the Java trace string.

The java trace specification is used to control java tracing and conforms to the java trace specification rules. *=all=enabled means to enable all types of tracing for all registered trace components.

HELP
Display a list of all the keywords you may use with the modify command.

You can also use the HELP parameter after the CANCEL, and DISPLAY parameters to display lists of all the keywords you can use with either of these parameters.

PAUSELISTENERS
Prevents work from being accepted into the server. Use this command if you want to shut down the communication listeners and purge any pending work in the work registry.

RESUMELISTENERS
Use this command to restart the communication listeners after issuing a **PAUSELISTENERS** command. Allows work to be accepted into the server.

DISPLAY | DISPLAY,
Displays the server's name, the system name where it is running, and the current code level.

You can specify the following options:

- **SERVERS** This command is directed to a server and displays the name, system name, and code level for each active server in the sysplex that is in the same cell.
- **SERVANTS** Displays a list of ASIDs of servants attached to the server against which you issued the display command.
- **TRACE** Display trace information for a server controller. You can further modify this command with one of the following options:
  - **SRS** Display trace information for all servants, one at a time.
  - **ALL** Display trace information for the controller and all servants one at a time.
  - **JAVA** Display the Java trace string settings for a server controller. You can further modify this command with one of the following options:
    - **SRS** Display Java trace information for all servants, one at a time.
    - **ALL** Display Java trace information for the controller and all servants one at a time.
    - **HELP** Display a list of all the keywords you may use with the modify display trace Java command.
  - **HELP** Display a list of all the keywords you may use with the modify display trace command.
- **JVMHEAP** Display the JVM heap information for a server controller. You can further modify this command with one of the following options:
  - **SRS** Display the JVM heap information for all servants, one at a time.
  - **ALL** Display the JVM heap information for the controller and all servants, one at a time.

  - **HELP** Display a list of all the keywords you may use with the modify display Javaheap command.
- **SESSIONS** Display session information for the server. You can further modify this command with one of the following options:
  - **LISTENERS** Display the listening port numbers for each protocol. This is actually the default, so the *f server,sessions* and the *f server,sessions,listeners* commands would have the same outcome.
  - **SERVER** Display the number of sessions in use for each protocol on the server. You can further modify this command with one of the following options:
    - **TCPIIOP** Display the number of TCP/IP IIOP sessions active on the server. You can further modify this command with one of the following options:
      - **LIST** List the server session information for the TCP/IP IIOP protocol. The LIST parameter is also available with the other session protocol display commands.
      - **HELP** Display a list of all the keywords you may use with the modify display session server **tcpipiiop** command. The HELP parameter is also available with the other session protocol display commands.
    - **LOCALIIOP** Display the number of LOCALIIOP sessions active on the server.
    - **SSLIIOP** Display the number of SSLIIOP sessions active on the server.
    - **HTTP** Display the number of HTTP sessions active on the server.
    - **HTTPS** Display the number of HTTPS sessions active on the server.
    - **HELP** Display a list of all the keywords you may use with the modify display session server command.
  - **HELP** Display help for the modify display sessions command.
- **HELP** Display a list of all the keywords you may use with the modify command.

Example: Canceling application clusters and servers with the modify command

This example demonstrates how to cancel application clusters and servers with the modify command.

Before you begin: You cannot cancel a cluster from the MVS console. Instead, you must cancel each of the servers that make up the cluster.

Example 1: The following command will cancel the *bbo6acr* server:

```
f bbo6acr,cancel
```

Example 2: The following command will cancel the *bbo6acr* server and instruct ARM to restart it after it terminates:

```
f bbo6acr,cancel,armrestart
```

Example: Establishing a general level of trace

This example demonstrates how to establish a general trace level using scripting.

To establish a general trace level for the server, use the following command:

```
f server,traceall=n
```

Valid trace levels are 0 (none), 1 (exception), 2 (basic), and 3 (detailed tracing). Under normal conditions and in production, use 1 (exception).

Example: The following command will turn on exception level tracing for the bbo6acr server:

```
f bbo6acr,traceall=1
```

Here is a sample display:

```
F bbo6acr,TRACEALL=1
BBOO0211I MODIFY COMMAND TRACEALL=1 COMPLETED SUCCESSFULLY
```

Example: Setting basic and detailed trace levels

This example demonstrates how to specify the WebSphere for z/OS components for which you want to switch on a basic level of tracing.

Use the following command:

```
f server,tracebasic=(n,...)
```

Example 1: The following command will turn on a basic level of tracing for the Daemon component on server *bbo6acr*:

```
f bbo6acr,tracebasic=a
```

Here is a sample display:

```
F bbo6acr,TRACEBASIC=A
BBOO0211I MODIFY COMMAND TRACEBASIC=A COMPLETED SUCCESSFULLY
```

To specify the WebSphere for z/OS components for which you want to switch on a detailed level of tracing use the following command:

```
f server,tracedetail=(n,...)
```

Example 2: The following command will turn on a detailed level of tracing for security on server *bbo6acr*:

```
f bbo6acr,tracedetail=e
```

Here is a sample display:

```
F bbo6acr,TRACEDETAIL=E
BBOO0211I MODIFY COMMAND TRACEDETAIL=E COMPLETED SUCCESSFULLY
```

Example: Setting specific trace points

This example demonstrates how to set specific trace points using scripting.

To set specific trace points, use the following command:

```
f server,tracespecific=n | (n,...)
```

Example 1: The following command will turn on the specific trace point 04006001:

```
f bbo6acr,tracespecific=04006001
```

Example 2: The following command will turn on the specific trace points 04006001 and 04006027:

```
f bbo6acr,tracespecific=(04006001,04006027)
```

Example 3: The following command will turn on the specific trace points set in the environment variable *tracepoints*:

```
f bbo6acr,tracespecific='tracepoints'
```

Example: Excluding specific trace points

This example shows how to exclude specific trace points using scripting.

To exclude specific trace points, use the following command:

```
f server,trace_exclude_specific=n | (n,...)
```

Example: The **tracespecific** command below turns on tracing for a whole part and then the *trace_exclude_specific* turns off tracing for the point 04006031 within that part.:

```
f bbo6acr,tracespecific=04006000

f bbo6acr,trace_exclude_specific=04006031
```

Example: Resetting to the initial trace settings

This example shows how to reset to the initial trace settings using scripting.

To reset to the initial trace settings use the following command:

```
f server,traceinit
```

Example: Turning off tracing

This example shows how to turn off tracing using scripting.

To turn off tracing, use the following command:

```
f server,tracenone
```

Example: Sending the trace to sysprint

This example shows how to send the trace output to sysprint using scripting.

To send the trace to sysprint, use the following command:

```
f server,tracetosysprint=yes
```

To stop sending the trace to sysprint, use the following command:

```
f server,tracetosysprint=no
```

Example: Getting help for the modify command

You can get syntax help for the various levels of the modify command.

Example 1: The following command will display a list of all the keywords you may use with the modify command:

```
f bbo6acr,help
```

Here is a sample display:

```
F bbo6acr,HELP
BBOO0178I THE COMMAND MODIFY  MAY BE FOLLOWED BY ONE OF THE FOLLOWING KEYWORDS:
BBOO0179I CANCEL - CANCEL THIS CONTROL REGION
BBOO0179I TRACEALL - SET OVERALL TRACE LEVEL
BBOO0179I TRACEBASIC - SET BASIC TRACE COMPONENTS
BBOO0179I TRACEDETAIL - SET DETAILED TRACE COMPONENTS
BBOO0179I TRACESPECIFIC - SET SPECIFIC TRACE POINTS
BBOO0179I TRACEINIT - RESET TO INITIAL TRACE SETTINGS
BBOO0179I TRACENONE - TURN OFF ALL TRACING
BBOO0179I TRACETOSYSPRINT - SEND TRACE OUTPUT TO SYSPRINT (YES/NO)
BBOO0179I TRACETOTRCFILE - SEND TRACE OUTPUT TO TRCFILE DD CARD (YES/NO)
BBOO0179I DISPLAY - DISPLAY STATUS
BBOO0179I TRACE_EXCLUDE_SPECIFIC - EXCLUDE SPECIFIC TRACE POINTS
BBOO0179I TRACEJAVA - SET JAVA TRACE OPTIONS
```

Example 2: The following command will display a list of all the keywords you may use with the modify display command:

```
f bbo6acr,display,help
```

Here is a sample display:

```
F bbo6acr,DISPLAY,HELP
BBOO0178I THE COMMAND DISPLAY, MAY BE FOLLOWED BY ONE OF THE FOLLOWING KEYWORDS:
BBOO0179I SERVERS - DISPLAY ACTIVE CONTROL REGIONS
BBOO0179I SERVANTS - DISPLAY SERVANT PROCESSES OWNED BY THIS CONTROL PROCESS
BBOO0179I SESSIONS - DISPLAY INFORMATION ABOUT COMMUNICATIONS SESSIONS
BBOO0179I TRACE - DISPLAY INFORMATION ABOUT TRACE SETTINGS
BBOO0179I JVMHEAP - DISPLAY JVM HEAP STATISTICS
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,HELP
```

Example: Modifying the Java trace string

This example shows how to use scripting to modify a Java trace string.

To modify the Java trace string, use the following command:

```
f server, tracejava='trace specification'
```

Example: The following command enables all types of tracing for all registered trace components:

```
f bbo6acr,tracejava='*=all=enabled'
```

Note: The single quotes are a required part of the syntax.

Display command

The display command is useful for obtaining data on a variety of processes.

Use the following display command examples to monitor your WebSphere Application Server.

- Display active replies
- Display active address spaces
- Display the status of address spaces registered for automatic restart management
- Display units of work (transactions) for the Information Management System

Example: Displaying active replies

Displaying active replies from the MVS console allows you to observe system activity and determine if the system requires an operator response.

Issue the following command to display (list) all active replies:

```
d r,r
```

Example: Displaying active address spaces

Use the display command to display active address spaces.

For example, use the display command to determine if the location service daemon is up:

d a,l	Displays a list of all address spaces.
d a,a	Displays a list of all active address spaces.
d a,*address-space-name*	Displays only the address space in which you are interested. This command is recommended over the first two because it does not yield such a lengthy list on a production system. Of course, you need to know the name of the address space for which you are looking. Example: d a,bboasr1
d a,bbo*	Displays a list of all active address spaces that start with BBO.

Example: Displaying the status of address spaces registered for automatic restart management

This example shows how to display the status of addresses registered with automatic restart management.

Perform the following steps to use automatic restart management (ARM) to display the status of ARM registered address spaces (including the address spaces of server instances) in the WebSphere for z/OS environment:

1. Initialize all servers.
2. Issue one or both of the following commands:
   - To display all registered address spaces (including the address spaces of server instances), issue the command:

     ```
     d xcf,armstatus,detail
     ```
   - To display the status of a particular server instance, use the display command and identify the job name. For example, to display the status of the Daemon server instance (job BBODMN), issue the following command:

     ```
     d  xcf,armstatus,jobname=bbodmn,detail
     ```

Example: Displaying units of work (transactions) for the Information Management System

This example shows how to use scripting to display the units of work for the Information Management System.

You can display units of work (transactions) for Information Management System (IMS):

1. To display the status of a specific transaction, issue the command:

   ```
   /dis tran trans-name
   ```
2. To display the status of a specific program, issue the command:

   ```
   /dis prog program-name
   ```
3. To display the number of Message Processing Regions (MPRs) that are currently active, issue the command:

   ```
   /display active region
   ```

   For more information about IMS commands, see the *IMS/ESA Summary of Operator Commands* manual on the IBM Publications Center Web site.

Example: Displaying servants

This example shows how to display servants using scripting.

To display servants, use the following command:

```
f server,display,servants
```

Example: The following command will display servants for the bbo6acr server:

```
f bbo6acr,display,servants
```

Here is a sample display:

```
F bbo6acr,DISPLAY,SERVANTS
BBOO0185I SERVER BBO5SR4/BBOASR4A HAS 1 SERVANT PROCESS (ASID: 0038x)
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,SERVANTS
```

Example: Displaying trace settings and Java string trace settings

This example shows how to display trace settings and Java string trace settings with scripting.

To display the trace settings for a server instance, use the following command:

```
f server,display,trace
```

Example 1: The following command will display trace settings for the bbo6acr server:

```
f bbo6acr,display,trace
```

Here is a sample display:

```
F bbo6acr,DISPLAY,TRACE
BBOO0224I TRACE INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00047
BBOO0197I LOCATION = SYSPRINT BUFFER
BBOO0197I AGGREGATE TRACE LEVEL = 1
BBOO0197I EXCEPTION TRACING = RAS(0), Common Utilities(1), COMM(3),
ORB(4), OTS(6), Shasta(7), OS/390 Wrappers(9), Daemon(A), Security(E),
Externalization(F), JRAS(J), J2EE(L)
BBOO0197I BASIC TRACING =
BBOO0197I DETAILED TRACING =
BBOO0197I TRACE SPECIFIC = NONE SPECIFIED
BBOO0197I TRACE EXCLUDE SPECIFIC = NONE SPECIFIED
BBOO0225I TRACE INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00047 COMPLETE
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE
```

To display the Java trace string settings for a server, use the following command:

```
f server,display,trace,java
```

Example 2: The following command will display Java trace settings for the bbo6acr server:

```
f bbo6acr,display,trace,java
```

Here is a sample display:

```
F bbo6acr,DISPLAY,TRACE,JAVA
BBOO0196I TRACE INFORMATION FOR SERVER BBO5SR4/BBOASR4A
BBOJ0050I CTL(STC00047):*=all=disabled
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE,JAVA
```

To display the Java trace information for all servants, one at a time.

```
f server,display,trace,java.srs
```

Example 3: The following command will display Java trace settings for the bbo6acr server:

```
f bbo6acr,display,trace,java,srs
```

Here is a sample display:

```
F bbo6acr,DISPLAY,TRACE,JAVA,SRS
BBOO0196I TRACE INFORMATION FOR SERVER BBO5SR4/BBOASR4A
BBOJ0050I SR(STC00048):*=all=disabled
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE,JAVA,SRS
```

To display the Java trace information for the controller and all servants, one at a time.

```
f server,display,trace,java.all
```

Example 4: The following command will display Java trace settings for the controller and all servants:

```
f bbo6acr,display,trace,java,all
```

Here is a sample display:

```
F bbo6acr,DISPLAY,TRACE,JAVA,ALL
BBOO0196I TRACE INFORMATION FOR SERVER BBO5SR4/BBOASR4A
BBOJ0050I CTL(STC00047):*=all=disabled
BBOJ0050I SR(STC00048):*=all=disabled
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,TRACE,JAVA,ALL
```

Example: Displaying JVM heap information

This example demonstrates how to display JVM heap information with scripting.

To display the JVM heap information for a server controller, use the following command:

```
f server,display,jvmheap
```

Example 1: The following command will display JVM heap information for the bbo6acr server:

```
f bbo6acr,display,jvmheap
```

Here is a sample display:

```
F bbo6acr,DISPLAY,JVMHEAP
BBOO0201I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00047
BBOO0202I (STC00047) HEAP(MIDDLEWARE), COUNT(00000000), FREE STORAGE(
396FA70), TOTAL STORAGE( 7FFFA00)
BBOO0204I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00047 COMPLETE
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,JVMHEAP
```

To display the JVM heap information for all servants one at a time, use the following command:

```
f server,display,jvmheap,srs
```

Example 2: The following command will display JVM heap information for all servants in the bbo6acr server:

```
f bbo6acr,display,jvmheap,srs
```

Here is a sample display:

```
F bbo6acr,DISPLAY,JVMHEAP,SRS
+BBOO0201I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00048
+BBOO0202I (STC00048) HEAP(MIDDLEWARE), COUNT(00000001), FREE STORAGE(
 25F4030), TOTAL STORAGE( 7FFFA00)
+BBOO0204I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00048 COMPLETE
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,JVMHEAP,SRS
```

To display the JVM heap information for the controller and all servants of a server one at a time, use the following command:

```
f server,display,jvmheap,all
```

Example 3: The following command will display JVM heap information for the controller and all servants in the bbo6acr server:

```
f bbo6acr,display,jvmheap,all
```

Here is a sample display:

```
F bbo6acr,DISPLAY,JVMHEAP,ALL
BBOO0201I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00047
BBOO0202I (STC00047) HEAP(MIDDLEWARE), COUNT(00000000), FREE STORAGE(
396FA70), TOTAL STORAGE( 7FFFA00)
BBOO0204I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00047 COMPLETE
+BBOO0201I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00048
+BBOO0202I (STC00048) HEAP(MIDDLEWARE), COUNT(00000001), FREE STORAGE(
 25F4030), TOTAL STORAGE( 7FFFA00)
+BBOO0204I JVM HEAP INFORMATION FOR SERVER BBO5SR4/BBOASR4A/STC00048 COMPLETE
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,JVMHEAP,ALL
```

Example: Displaying status of a server

This example shows how to use scripting to display the status of a server.

To display the status of a server, use the following command:

```
f server,display
```

Example: The following command will display status for the bbo6acr server:

```
f bbo6acr,display
```

Here is a sample display:

```
F bbo6acr,DISPLAY
BBOO0173I SERVER BBO5SR4/BBOASR4A ACTIVE ON SY1 AT LEVEL wd5xo03.
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY
```

Example: Displaying status of clusters

This example shows how to use scripting to display the status of clusters.

To display the status of clusters in the active controller, use the following command:

```
f server,display,servers
```

Example: The following command will display status for the bbo6acr server:

```
f bbo6acr,display,servers
```

Here is a sample display:

```
F bbo6acr,DISPLAY,SERVERS
BBOO0182I SERVER             ASID   SYSTEM    LEVEL
BBOO0183I CBDAEMON/DAEMON01    31x  SY1       wd5xo03
BBOO0183I BBO5SR4 /BBOASR4A   1F6x  SY1       wd5xo03
BBOO0188I END OF OUTPUT FOR COMMAND DISPLAY,SERVERS
```

Chapter 7. Using Ant to automate tasks

To support using Apache Ant with Java 2 Platform, Enterprise Edition (J2EE) applications running on the application server, the product provides a copy of the Ant tool and a set of Ant tasks that extend the capabilities of Ant to include product-specific functions. Ant has become a very popular tool among Java programmers.

Apache Ant is a Java-based build tool. In theory, it is similar to Make, but Ant is different. Instead of a model in which it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, XML-based configuration files are used. These files reference a target tree in which various tasks are run. Each task is run by an object that implements a particular Task interface.

By combining the following tasks with those provided by Ant, you can create build scripts that compile, package, install, and test your application on the application server:

- Install and uninstall applications
- Start and stop servers in a base configuration
- Run administrative scripts or commands
- Run the Enterprise JavaBeans (EJB) deployment tool
- Run the JavaServer Pages (JSP) file precompilation tool

For more detailed information about Ant, refer to the Apache organization Web site.

- To run Ant and have it automatically see the WebSphere classes, use the “ws_ant command.”
- Use “Ant tasks for deployment and server operation.”

  This topic describes where to find the API documentation for the Apache Ant tasks for deploying applications and operating application servers.
- Use “Ant tasks for building application code” on page 984.

  This topic describes where to find the API documentation for the Apache Ant tasks for building applications.

ws_ant command

This topic describes where to find information about the ws_ant command, which is provided for using with Apache Ant, a Java-based build tool that is popular among Java programmers.

In theory, Ant is similar to Make, but Ant is different. Instead of a model in which it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, XML-based configuration files are used. These files reference a target tree in which various tasks are run. Each task is run by an object that implements a particular Task interface.

For the Apache Ant tool that is provided by this product, see the file *app_server_root*/bin/ws_ant.bat|sh.

Ant tasks for deployment and server operation

This topic describes where to find the API documentation for the Apache Ant tasks for deploying applications and operating application servers.

The Apache Ant tasks for the product reside in the Java package: `com.ibm.websphere.ant.tasks`. The API documentation for this package contains detailed information about all of the Ant tasks that are provided and how to use them. The API documentation is available in the **Reference** section of the information center.

Ant tasks for building application code

This topic describes where to find the documentation for the Ant tasks provided for building application code using the Application Server Toolkit (which is a CD included with WebSphere Application Server as a separately installable toolkit).

Note that this toolkit includes an Automated Deployment example ″Example: Automated Deploy″ for JACL scripted deployment of multiple application updates to multiple servers and clusters in a WebSphere Network Deployment cell.

Within the Application Server Toolkit product documentation, open the section **Working with Ant**. You can locate the topic by searching for **Working with Ant**, or from the navigation view, select **Help > Help Contents > Application Server Toolkit > J2EE applications > Working with Ant**.

Appendix. Directory conventions

References in the product information to *app_server_root*, *profile_root*, and other directories imply specific directory locations. This topic describes the conventions in use for WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E.

The corresponding product variable is smpe.install.root.

The default is /usr/lpp/zWebSphere/V6R1.

configuration_root
Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The configuration_root contains the various app_server_root directories and certain symbolic links associated with them. Each different configuration_root normally requires its own cataloged procedures under z/OS.

The default is /WebSphere/V6R1.

app_server_root
Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own app_server_root. Before Version 6.0 of the product information, this was referred to as the ″WAS_HOME″ directory. Corresponding product variables are was.install.root and WAS_HOME.

The default varies based on node type. Common defaults are *configuration_root*/Appserver and *configuration_root*/DeploymentManager.

profile_root
Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are server.root and user.install.root.

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will be always be *app_server_root*/profiles/default because only the profile name ″default″ is used in WebSphere Application Server for z/OS.

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.

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks and service marks

For trademark attribution, visit the IBM Terms of Use Web site (http://www.ibm.com/legal/us/).