WebSphere® Application Server for IBM i, Version 7.0

# Troubleshooting and support

**Note**

Before using this information, be sure to read the general information under “Notices” on page 149.

**Compilation date: September 3, 2008**

# Contents

# How to send your comments

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

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

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

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

# Changes to serve you more quickly

**Print sections directly from the information center navigation**

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

To print a section of the information center navigation:

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

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

**Under construction!**

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

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

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

# Chapter 1. Debugging applications

To debug your application, you must use a development environment like the IBM® Rational® Application Developer for WebSphere® to create a Java™ project. You must then import the program that you want to debug into the project.

## About this task

By following the steps below, you can import the WebSphere Application Server examples into a Java project. Two debugging styles are available:

- **Step-by-step** debugging mode prompts you whenever the server calls a method on a Web object. A dialog lets you step into the method or skip it. In the dialog, you can turn off step-by-step mode when you are finished using it.
- **Breakpoints** debugging mode lets you debug specific parts of programs. Add breakpoints to the part of the code that you must debug and run the program until one of the breakpoints is encountered.

Breakpoints actually work with both styles of debugging. Step-by-step mode just lets you see which Web objects are being called without having to set up breakpoints ahead of time.

You do not need to import an entire program into your project. However, if you do not import all of your program into the project, some of the source might not compile. You can still debug the project. Most features of the debugger work, including breakpoints, stepping, and viewing and modifying variables. You must import any source that you want to set breakpoints in.

The inspect and display features in the source view do not work if the source has build errors. These features let you select an expression in the source view and evaluate it.

1. Create a Java Project by opening the New Project dialog.
2. Select **Java** from the left side of the dialog and **Java Project** in the right side of the dialog.
3. Click **Next** and specify a name for the project, for example, WASExamples.
4. Click **Finish** to create the project.
5. Select the new project, choose **File > Import > File System**, then **Next** to open the import file system dialog.
6. Browse the directory for files.

   Go to the following directory: *profile_root*/installedApps/*node_name*/DefaultApplication.ear/DefaultWebApplication.war.
7. Select DefaultWebApplication.war in the left side of the Import dialog and then click **Finish**. This imports the JavaServer Pages files and Java source for the examples into your project.
8. Add any JAR files needed to build to the Java Build Path.

   Select **Properties** from the right-click menu. Choose the Java Build Path node and then select the Libraries tab. Click **Add External JARs** to add the following JAR files:
   - *profile_root*/installedApps/*node_name*/DefaultApplication.ear/Increment.jar.

     When you have added this JAR file, select it and use the **Attach Source** function to attach the Increment.jar file because it contains both the source and class files.
   - *app_server_root*/lib/j2ee.jar
   - *app_server_root*/plugins//com.ibm.ws.runtime.jar
   - *app_server_root*/plugins/com.ibm.ws.webcontainer.jar

   Click **OK** when you have added all of the JARs.
9. You can set some breakpoints in the source at this time if you like, however, it is not necessary as step-by-step mode will prompt you whenever the server calls a method on a Web object. Step-by-step mode is explained in more detail below.

10. To start debugging, you need to start the WebSphere Application Server in debug mode and make note of the JVM debug port. The default value of the JVM debug port is 7777.
11. When the server is started, switch to the debug perspective by selecting **Window > Open Perspective > Debug**. You can also enable the debug launch in the Java Perspective by choosing **Window > Customize Perspective** and selecting the **Debug** and **Launch** checkboxes in the **Other** category.
12. Select the workbench toolbar **Debug** pushbutton and then select **WebSphere Application Server Debug** from the list of launch configurations. Click the **New** pushbutton to create a new configuration.
13. Give your configuration a name and select the project to debug (your new WASExamples project). Change the port number if you did not start the server on the default port (7777).
14. Click **Debug** to start debugging.
15. Load one of the examples in your browser. For example: `http://your.server.name:9080/hitcount`

### What to do next

To learn more about debugging, launch the The IBM Rational Application Developer for WebSphere, select **Help > Help Contents** and choose the **Debugger Guide bookshelf** entry. To learn about known limitations and problems that are associated with the IBM Rational Application Developer for WebSphere, see the IBM Rational Application Developer for WebSphere release notes. For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the Must gather documents page for information to gather to send to IBM Support.

## Debugging components in the IBM Rational Application Developer for WebSphere

The IBM Rational Application Developer for WebSphere, included with the WebSphere Application Server on a separately-installable CD, includes debugging functionality that is built on the Eclipse workbench. Documentation for the IBM Rational Application Developer for WebSphere is provided with that product. To learn more about the debug components, launch the IBM Rational Application Developer for WebSphere, select **Help > Help Contents** and choose the **Developing > Debugging applications** bookshelf entries.

The IBM Rational Application Developer for WebSpheret includes the following:

**The WebSphere Application Server debug adapter**
: which allows you to debug Web objects that are running on WebSphere Application Server and that you have launched in a browser. These objects include enterprise beans, JavaServer Pages files, and servlets.

**The JavaScript™ debug adapter**
: which enables server-side JavaScript debugging.

**The Compiled language debugger**
: which allows you to detect and diagnose errors in compiled-language applications.

**The Java development tools (JDT) debugger**
: which allows you to debug Java code.

All of the debug components in the IBM Rational Application Developer for WebSphere can be used for debugging locally and for remote debugging. To learn more about the debug components, launch the IBM Rational Application Developer for WebSphere, select **Help > Help Contents** and choose the **Developing > Debugging applications** bookshelf entries.

# Chapter 2. Adding logging and tracing to your application

You can add logging and tracing to applications to help analyze performance and diagnose problems in WebSphere Application Server.

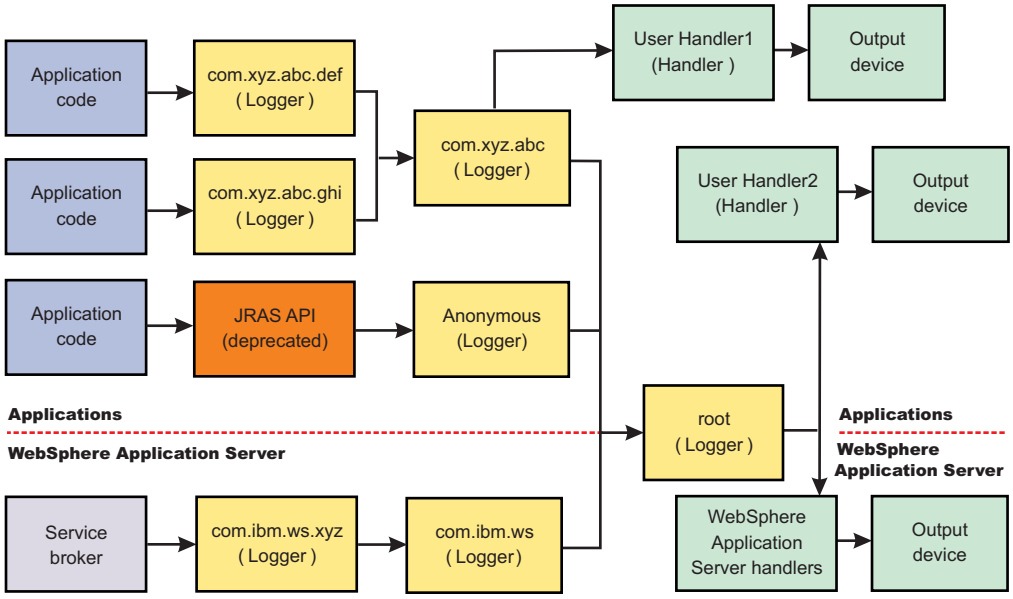
## About this task

**Deprecation:** The JRas framework that is described in this information center is deprecated. However, you can achieve the same results using Java logging.

Designers and developers of applications that run with or under WebSphere Application Server, such as servlets, JavaServer Pages (JSP) files, enterprise beans, client applications, and their supporting classes, might find it useful to use Java logging for generating their application logging.

This approach has advantages over adding `System.out.println` statements to your code:

- Your messages are displayed in the WebSphere Application Server standard log files, using a standard message format with additional data, such as a date and time stamp that are added automatically.
- You can more easily correlate problems and events in your own application to problems and events that are associated with WebSphere Application Server components.
- You can take advantage of the WebSphere Application Server log file management features.




1. Enable and configure one of the supported types of logging. Use one of the following methods:
   - “Configuring Java logging using the administrative console” on page 4
   - “Configuring applications to use Jakarta Commons Logging” on page 25
   - “Logging Common Base Events in WebSphere Application Server” on page 51.
2. Customize the properties to meet your logging needs. For example, enable or disable a particular log, specify the number of logs to be kept, and specify a format for log output.
3. Restart the application server after making static configuration changes.

## Configuring Java logging using the administrative console

Java logging provides a standard logging API for your applications. Before applications can log diagnostic information, you need to specify how you want the server to handle log output and what level of logging you require.

### About this task

Developing, deploying and maintaining applications are complex tasks. When an application encounters an unexpected condition, it might not be able to complete a requested operation. You might want the application to inform the administrator that the operation failed and tell the administrator why the operation failed. This information enables the administrator to take the proper corrective action. Application developers might need to gather detailed information that relates to the path of a running application to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as *logging* and *tracing*. For more information read “Java logging.”

Using the administrative console, you can:

- Enable or disable a particular log, specify where log files are stored and how many log files are kept.
- Specify the level of detail in a log, and specify a format for log output.
- Set a log level for each logger.

You can change the log configuration statically or dynamically. Static configuration changes affect applications when you start or restart the application server. Dynamic or run time configuration changes apply immediately.

When a log is created, the level value for that log is set from the configuration data. If no configuration data is available for a particular log name, the level for that log is obtained from the parent of the log. If no configuration data exists for the parent log, the parent of that log is checked, and so on up the tree, until a log with a non-null level value is found. When you change the level of a log, the change is propagated to the children of the log, which recursively propagates the change to their children, as necessary.

1. Optional: See the Java documentation for the java.util.logging class for a full description of the syntax and the construction of logging methods.
2. Set the logging levels for your logs:
   a. In the navigation pane, click **Servers > Application Servers**.
   b. Click the name of the server that you want to work with.
   c. Under Troubleshooting, click **Logs and Trace**.
   d. Click **Change Log Detail levels**.
   e. To make a static change to the configuration, click the **Configuration** tab. A list of well-known components, packages, and groups is displayed. To change the configuration dynamically, click the **Runtime** tab. The list of components, packages, and groups displays all the components that are currently registered on the running server.
   f. Select a component, package, or group to set a logging level.
   g. Click **Apply**.
   h. Click **OK**.
3. To have static configuration changes take effect, stop then restart the application server.

## Java logging

Java logging is the logging toolkit that is provided by the java.util.logging package. Java logging provides a standard logging API for your applications.

Message logging (messages) and diagnostic trace (trace) are conceptually similar, but do have important differences. These differences are important for application developers to understand to use these tools properly. The following operational definitions of messages and trace are provided.

**Message**
A message entry is an informational record that is intended for end users, systems administrators, and support personnel to view. The text of the message must be clear, concise, and interpretable by an end user. Messages are typically localized and displayed in the national language of the end user. Although the destination and lifetime of messages might be configurable, enable some level of message logging in normal system operation. Use message logging judiciously because of performance considerations and the size of the message repository.

**Trace** A trace entry is an information record that is intended for service engineers or developers to use. As such, a trace record might be considerably more complex, verbose, and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries can be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but can be enabled as needed to gather diagnostic information.

The application server redirects the system streams at the server startup. There is no way to allow the application to output logging to the console because the system streams can not be obtained by the application. If you would like to use console to monitor the application without using the console handler, you can either monitor the `SystemOut.log` file, or monitor a file created by another file handler.

**Note:** The application server uses Java logging internally and therefore certain restrictions apply for using system streams with this logging API by applications. During server startup, the standard output and error streams are replaced with special streams that write to the logging infrastructure, in order to include the output of the system streams in the log files. Because of this, applications can not use `java.util.logging.ConsoleHandler`, or any handler writing to `System.err` or `System.out` streams, attached to the root logger. If the user does attach the handler to the root logger, an infinite loop is created within the logging infrastructure, leading to stack overflow and server crash.

If the use of a handler that writes to system streams is necessary, attach it to a non-root logger so that it does not publish log records to parent handlers. The data written to the system streams is then formatted and written to the corresponding system stream log file. To monitor what is being written system streams, the configured log files (`SystemOut.log` and `SystemErr.log` by default) can be monitored.

## Log level settings

Use this topic to configure and manage log level settings.

Using log levels you can control which events are processed by Java logging. When you change the level for a logger, the change is propagated to the children of the logger.

**Change Log Detail Levels**

Enter a log detail level that specifies the components, packages, or groups to trace. The log detail level string must conform to the specific grammar described in this topic. You can enter the log detail level string directly, or generate it using the graphical trace interface.

If you select the Configuration tab, a static list of well-known components, packages, and groups is displayed. This list might not be exhaustive.

If you select the Runtime tab, the list of components, packages, and group are displayed with all the components that are registered on the running application server and in the static list.

The format of the log detail level specification is:

```
<component> = <level>
```

where <component> is the component for which to set a log detail level, and <level> is one of the valid logger levels (off, fatal, severe, warning, audit, info, config, detail, fine, finer, finest, all). Separate multiple log detail level specifications with colons (:).

Components correspond to Java packages and classes, or to collections of Java packages. Use an asterisk (*) as a wildcard to indicate components that include all the classes in all the packages that are contained by the specified component. For example:

***** Specifies all traceable code running in the application server, including the product system code and customer code.

**com.ibm.ws.***
Specifies all classes with the package name beginning with com.ibm.ws.

**com.ibm.ws.classloader.JarClassLoader**
Specifies the JarClassLoader class only.

An error can occur when setting a log detail level specification from the administrative console if selections are made from both the Groups and Components lists. In some cases, the selection made from one list is lost when adding a selection from the other list. To work around this problem, enter the log detail level specification directly into the log detail level entry field.

Select a component or group to set a log detail level. The table following lists the valid levels for application servers at WebSphere Application Server Version 6 and later, and the valid logging and trace levels for earlier versions:

| Version 6 logging level | Logging level before Version 6 | Trace level before Version 6 | Content / Significance |
|---|---|---|---|
| Off | Off | All disabled* | Logging is turned off.<br><br>* In Version 6, a trace level of All disabled turns off trace, but does not turn off logging. Logging is enabled from the Info level. |
| Fatal | Fatal | - | Task cannot continue and component, application, and server cannot function. |
| Severe | Error | - | Task cannot continue but component, application, and server can still function. This level can also indicate an impending fatal error. |
| Warning | Warning | - | Potential error or impending error. This level can also indicate a progressive failure (for example, the potential leaking of resources). |
| Audit | Audit | - | Significant event affecting server state or resources |
| Info | Info | - | General information outlining overall task progress |
| Config | - | - | Configuration change or status |
| Detail | - | - | General information detailing subtask progress |

| | | | |
|---|---|---|---|
| Fine | - | Event | Trace information - General trace + method entry, exit, and return values |
| Finer | - | Entry/Exit | Trace information - Detailed trace |
| Finest | - | Debug | Trace information - A more detailed trace that includes all the detail that is needed to debug problems |
| All | | All enabled | All events are logged. If you create custom levels, All includes those levels, and can provide a more detailed trace than finest. |

When you enable a logging level in Version 6.0 or above, you are also enabling all of the levels with higher severity. For example, if you set the logging level to warning on your Version 6.x application server, then warning, severe and fatal events are processed.

Trace information, which are events at the Fine, Finer and Finest levels, can be written only to the trace log. Therefore, if you do not enable diagnostic trace, setting the log detail level to Fine, Finer, or Finest will not have an effect on the data that is logged.

## Loggers

Loggers are used by applications and runtime components to capture message and trace events.

When situations occur that are significant either due to a change in state, for example when a server completes startup or because a potential problem is detected, such as a timeout waiting for a resource, a message is written to the logs. Trace events are logged in debugging scenarios, where a developer needs a clear view of what is occurring in each component to understand what might be going wrong. Logged events are often the only events available when a problem is first detected, and are used during both problem recovery and problem resolution.

Loggers are organized hierarchically. Each logger can have zero or more child loggers.

Loggers can be associated with a resource bundle. If specified, the resource bundle is used by the logger to localize messages that are logged to the logger. If the resource bundle is not specified, a logger uses the same resource bundle as its parent.

You can configure loggers with a level. If specified, the level is compared by the logger to incoming events. The events that are less severe than the level set for the logger are ignored by the logger. If the level is not specified, a logger takes on the level that is used by its parent. The default level for loggers is Level.INFO.

Loggers can have zero or more attached handlers. If supplied, all events that are logged to the logger are passed to the attached handlers. Handlers write events to output destinations such as log files or network sockets. When a logger finishes passing a logged event to all of the handlers that are attached to that logger, the logger passes the event to the handlers that are attached to the parents of the logger. This process stops if a parent logger is configured not to use its parent handlers. Handlers in WebSphere Application Server are attached to the root logger. Set the useParentHandlers logger property to `false` to prevent the logger from writing events to handlers that are higher in the hierarchy.

Loggers can have a filter. If supplied, the filter is invoked for each incoming event to tell the logger whether or not to ignore it.

Applications interact directly with loggers to log events. To obtain or create a logger, a call is made to the Logger.getLogger method with a name for the logger. Typically, the logger name is either the package qualified class name or the name of the package that the logger is used by. The hierarchical logger namespace is automatically created by using the dots in the logger name. For example, the com.ibm.websphere.ras logger has a com.ibm.websphere parent logger, which has a com.ibm parent. The parent at the top of the hierarchy is referred to as the *root logger*. This root logger is created during initialization. The root logger is the parent of the com logger.

Loggers are structured in a hierarchy. Every logger, except the root logger, has one parent. Each logger can also have 0 or more children. A logger inherits log handlers, resource bundle names, and event filtering settings from its parent in the hierarchy. The logger hierarchy is managed by the LogManager function.

Loggers create log records. A log record is the container object for the data of an event. This object is used by filters, handlers, and formatters in the logging infrastructure.

The logger provides several sets of methods for generating log messages. Some log methods take only a level and enough information to construct a message. Other, more complex logp (log precise) methods support the caller in passing class name and method name attributes, in addition to the level and message information. The logrb (log with resource bundle) methods add the capability of specifying a resource bundle as well as the level, message information, class name, and method name. Using methods such as severe, warning, fine, finer, and finest you can log a message at a particular level. For more information on logging and how to use it in your applications read “Using loggers in an application” on page 10. For a complete list of methods, see the java.util.logging documentation at http://java.sun.com/javase/.

## Log handlers

Log handlers write log record objects to output devices like log files, sockets, and notification mechanisms.

Loggers can have zero or more attached handlers. All objects that are logged to the logger are passed to the attached handlers, if handlers are supplied.

You can configure handlers with a level. The handler compares the level that is specified in the logged object to the level that is specified for the handler. If the level of the logged object is less severe than the level set in the handler, the object is ignored by the handler. The default level for handlers is ALL.

Handlers can have a filter. If a filter is supplied, the filter is invoked for each incoming object to tell the handler whether or not to ignore it.

Handlers can have a formatter. If a formatter is supplied, the formatter controls how the logged objects are formatted. For example, the formatter can decide to first include the time stamp, followed by a string representation of the level, followed by the message that is included in the logged object. The handler writes this formatted representation to the output device. Read “Example: Creating custom formatters with java.util.logging” on page 20 for information on using a custom formatter in your applications.

Both loggers and handlers can have levels and filters, and a logged object must pass all of these elements to be output. For example, you can set the logger level to FINE, but if the handler level is set at WARNING, only WARNING level messages are displayed in the output for that handler. Conversely, if your log handler is set to output all messages (level=All), but the logger level is set to WARNING, the logger never sends messages lower than WARNING to the log handler.

## Log levels

Levels control which events are processed by Java logging. WebSphere Application Server controls the levels of all loggers in the system.

The level value is set from configuration data when the logger is created and can be changed at run time from the administrative console. If a level is not set in the configuration data, a level is obtained by proceeding up the hierarchy until a parent with a level value is found. You can also set a level for each handler to indicate which events are published to an output device. When you change the level for a logger in the administrative console, the change is propagated to the children of the logger.

Levels are cumulative; a logger can process logged objects at the level that is set for the logger, and at all levels above the set level. Valid levels are:

| Level | Content / Significance |
|---|---|
| Off | No events are logged. |
| Fatal | Task cannot continue and component cannot function. |
| Severe | Task cannot continue, but component can still function |
| Warning | Potential error or impending error |
| Audit | Significant event affecting server state or resources |
| Info | General information outlining overall task progress |
| Config | Configuration change or status |
| Detail | General information detailing subtask progress |
| Fine | Trace information - General trace + method entry / exit / return values |
| Finer | Trace information - Detailed trace |
| Finest | Trace information - A more detailed trace - Includes all the detail that is needed to debug problems |
| All | All events are logged. If you create custom levels, All includes your custom levels, and can provide a more detailed trace than Finest. |

For instructions on how to set logging levels, see “Configuring Java logging using the administrative console” on page 4

**Note:** Trace information, which includes events at the Fine, Finer and Finest levels, can be written only to the trace log. Therefore, if you do not enable diagnostic trace, setting the log detail level to Fine, Finer, or Finest does not effect the logged data.

## Log filters

Log filters help control more detailed logging settings that are not handled by usual log level settings.

A filter provides an optional, secondary control over what is logged, beyond the control that is provided by setting the level. Applications can apply a filter mechanism to control logging output through the logging APIs. An example of filter usage is to suppress all the events with a particular message key.

A filter is attached to a logger or log handler using the appropriate setFilter method. Read “Example: Creating custom filters with java.util.logging” on page 19 for information on implementing custom filters. For a complete list of filter methods, see the java.util.logging documentation at http://java.sun.com/javase/

## Log formatters

Log formatters format log messages so they can be used by various log handlers.

Handlers can be configured with a log formatter that knows how to format log records. The event, which is represented by the log record object, is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which writes the output to the output device.

The formatter is responsible for rendering the event for output. This formatter uses the resource bundle that is specified in the event to look up the message in the appropriate language.

Formatters are attached to handlers using the setFormatter method.

You can find the java.util.logging documentation at http://java.sun.com/javase/.

# Using loggers in an application

This topic describes how to use Java logging within an application.

## About this task

To create an application using Java logging, perform the following steps:

1. Create the necessary handler, formatter, and filter classes if you need your own log files.
2. If localized messages are used by the application, create a resource bundle, as described in “Creating log resource bundles and message files” on page 14.
3. In the application code, get a reference to a logger instance, as described in “Using a logger.”
4. Insert the appropriate message and trace logging statements in the application, as described in “Using a logger.”

## Using a logger

You can use Java logging to log messages and add tracing.

### About this task

Use WsLevel.DETAIL level and above for messages, and lower levels for trace. The WebSphere Application Server Extension API (the com.ibm.websphere.logging package) contains the WsLevel class.

For messages use:

```
WsLevel.FATAL
Level.SEVERE
Level.WARNING
WsLevel.AUDIT
Level.INFO
Level.CONFIG
WsLevel.DETAIL
```

For trace use:

```
Level.FINE
Level.FINER
Level.FINEST
```

1. Use the logp method instead of the log or the logrb method. The logp method accepts parameters for class name and method name. The log and logrb methods will generally try to infer this information, but the performance penalty is prohibitive. In general, the logp method has less performance impact than the log or the logrb method.
2. Avoid using the logrb method. This method leads to inefficient caching of resource bundles and poor performance.
3. Use the isLoggable method to avoid creating data for a logging call that does not get logged. For example:

```
if (logger.isLoggable(Level.FINEST)) {
  String s = dumpComponentState(); // some expensive to compute method
  logger.logp(Level.FINEST, className, methodName, "componentX state
dump:\n{0}", s);
 }
```

## Example

The following sample applies to localized messages:

```
// note - generally avoid use of FINE, FINER, FINEST levels for messages to be consistent with
// WebSphere Application Server

String componentName = "com.ibm.websphere.componentX";
String resourceBundleName = "com.ibm.websphere.componentX.Messages";
Logger logger = Logger.getLogger(componentName, resourceBundleName);

// "Convenience" methods - not generally recommended due to lack of class
/ method names
//   - cannot specify message substitution parameters
//   - cannot  specify class and method names
if (logger.isLoggable(Level.SEVERE))
 logger.severe("MSG_KEY_01");

if (logger.isLoggable(Level.WARNING))
 logger.warning("MSG_KEY_01");

if (logger.isLoggable(Level.INFO))
 logger.info("MSG_KEY_01");

if (logger.isLoggable(Level.CONFIG))
 logger.config("MSG_KEY_01");


// log methods are not generally used due to lack of class and method
names
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(WsLevel.FATAL))
 logger.log(WsLevel.FATAL, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.log(Level.SEVERE, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.log(Level.WARNING, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.log(WsLevel.AUDIT, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.log(Level.INFO, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.log(Level.CONFIG, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.log(WsLevel.DETAIL, "MSG_KEY_01", "parameter 1");


// logp methods are the way to log
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
if (logger.isLoggable(WsLevel.FATAL))
 logger.logp(WsLevel.FATAL, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.logp(Level.SEVERE, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.logp(Level.WARNING, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.logp(WsLevel.AUDIT, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.logp(Level.INFO, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.logp(Level.CONFIG, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.logp(WsLevel.DETAIL, className, methodName, "MSG_KEY_01",
"parameter 1");


// logrb methods are not generally used due to diminished performance
```

```
of switching resource bundles dynamically
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
String resourceBundleNameSpecial =
"com.ibm.websphere.componentX.MessagesSpecial";

if (logger.isLoggable(WsLevel.FATAL))
 logger.logrb(WsLevel.FATAL, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.logrb(Level.SEVERE, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.logrb(Level.WARNING, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.logrb(WsLevel.AUDIT, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.logrb(Level.INFO, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.logrb(Level.CONFIG, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.logrb(WsLevel.DETAIL, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");
```

For trace, or content that is not localized, the following sample applies:

```
// note - generally avoid use of FATAL, SEVERE, WARNING, AUDIT,
// INFO, CONFIG, DETAIL levels for trace
// to be consistent with WebSphere Application Server

String componentName = "com.ibm.websphere.componentX";
Logger logger = Logger.getLogger(componentName);

// Entering / Exiting methods are used for non trivial methods
if (logger.isLoggable(Level.FINER))
 logger.entering(className, methodName);

if (logger.isLoggable(Level.FINER))
 logger.entering(className, methodName, "method param1");

if (logger.isLoggable(Level.FINER))
 logger.exiting(className, methodName);

if (logger.isLoggable(Level.FINER))
 logger.exiting(className, methodName, "method result");


// Throwing method is not generally used due to lack of message - use
logp with a throwable parameter instead
if (logger.isLoggable(Level.FINER))
 logger.throwing(className, methodName, throwable);


// Convenience methods are not generally used due to lack of class
/ method names
//   - cannot specify message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(Level.FINE))
 logger.fine("This is my trace");

if (logger.isLoggable(Level.FINER))
 logger.finer("This is my trace");

if (logger.isLoggable(Level.FINEST))
 logger.finest("This is my trace");


// log methods are not generally used due to lack of class and
method names
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(Level.FINE))
 logger.log(Level.FINE, "This is my trace", "parameter 1");

if (logger.isLoggable(Level.FINER))
 logger.log(Level.FINER, "This is my trace", "parameter 1");
```

```
if (logger.isLoggable(Level.FINEST))
 logger.log(Level.FINEST, "This is my trace", "parameter 1");


// logp methods are the recommended way to log
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
if (logger.isLoggable(Level.FINE))
 logger.logp(Level.FINE, className, methodName, "This is my trace",
"parameter 1");

if (logger.isLoggable(Level.FINER))
 logger.logp(Level.FINER, className, methodName, "This is my trace",
"parameter 1");

if (logger.isLoggable(Level.FINEST))
 logger.logp(Level.FINEST, className, methodName, "This is my trace",
"parameter 1");


// logrb methods are not applicable for trace logging because no localization
is involved
```

## Configuring the logger hierarchy

WebSphere Application Server handlers are attached to the Java root logger, which is at the top of the logger hierarchy. As a result, any request from anywhere in the logger tree can be processed by WebSphere Application Server handlers.

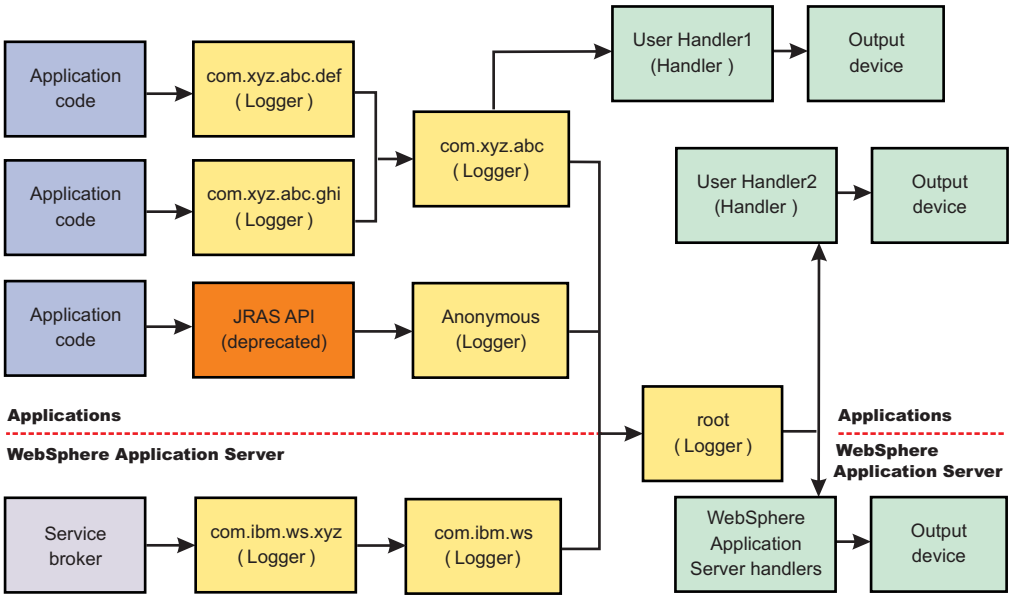
### About this task

You can configure your application server to handle logs in many different ways. Configure your log settings based upon your configuration and the logging structure that best suits your needs.

- Forward all application logging requests to the WebSphere Application Server handlers. This behavior is the default.
- Forward all application logging requests to your own custom handlers. Set the **useParentHandlers** option to `false` on one of your custom loggers, and then attach your handlers to that logger.
- Forward all application logging requests to both WebSphere Application Server handlers, and your custom handlers, but do not forward WebSphere Application Server logging requests to your custom handlers. Set the **useParentHandlers** option to `true` on one of your non-root custom loggers, and then attach your handlers to that logger.`True` is the default setting.
- Forward all WebSphere Application Server logging requests to both WebSphere Application Server handlers, and your custom handlers. Logging requests are always forwarded to WebSphere Application Server handlers. To forward WebSphere Application Server requests to your custom handlers, attach your custom handlers to the Java root logger, so that they are at the same level in the hierarchy as the WebSphere Application Server handlers.

### Example

The following example shows how these requirements can be met using the Java logging infrastructure:




## Creating log resource bundles and message files

You can forward messages that are written to the internal WebSphere Application Server logs to other processes for display. Messages that are displayed on the administrative console, which can be running in a different location than the server process, can be localized using the *late binding* process. Late binding means that WebSphere Application Server does not localize messages when they are logged, but defers localization to the process that displays the message.

### About this task

Every method that accepts messages localizes those messages. The mechanism for providing localized messages is the resource bundle support provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.ResourceBundle, java.util.ListResourceBundle and java.util.PropertyResourceBundle classes, as well as the java.text.MessageFormat class.

The PropertyResourceBundle class is the preferred mechanism to use.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. You must package the resource bundle separately from the application, and install it in a location where the viewing process can access it.

By default, the WebSphere Application Server runtime localizes all the messages when they are logged. This localization eliminates the need to pass a `.jar` file to the application, unless you need to localize in a different location. However, you can use the early binding technique to localize messages as they log. An application that uses early binding must localize the message before logging it. The application looks up the localized text in the resource bundle and formats the message. Use the early binding technique to package the application resource bundles with the application.

To create a resource bundle, perform the following steps.

1. Create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:
   - Each property in the file is terminated with a line-termination character.
   - If a line contains white space only, or if the first non-white space character of the line is the pound sign symbol (#) or exclamation mark (!), the line is ignored. The # and ! characters can therefore be used to put comments into the file.
   - Each line in the file, unless it is a comment or consists of white space only, denotes a single property. A backslash (\) is treated as the line-continuation character.
   - The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (=), colon (:), and white space ( ).
   - The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but doing this process is not recommended, because escaping characters is error prone and confusing. Instead, use a valid separator character that does not display in any keys in the properties file.
   - White space after the key and separator is ignored until the first non-white space character is encountered. All characters remaining before the line-termination character define the element.

   See the Java documentation for the java.util.Properties class for a full description of the syntax and the construction of properties files.
2. Translate the file into localized versions of the file with language-specific file names. For example, a file named `DefaultMessages.properties` can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese.
3. When the translated resource bundles are available, put the bundle in a directory that is part of the application class path.
4. When a message logger is obtained from the log manager, configure it to use a particular resource bundle. Messages logged with the Logger API use this resource bundle when message localization is performed. At run time, the user locale setting determines the properties file from which to extract the message that is specified by a message key, ensuring that the message is delivered in the correct language.
5. If the message loggers msg method is called, a resource bundle name must be explicitly provided.

## What to do next

The application locates the resource bundle based on the file location relative to any directory in the class path. For instance, if the DefaultMessages.properties property resource bundle is located in the *baseDir*/*subDir1*/*subDir2*/`resources` directory and *baseDir* is in the class path, the name *subdir1*.*subdir2*.`resources.DefaultMessage` is passed to the message logger to identify the resource bundle.

### *Example: Logging resource bundles by creating a properties file:*

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a properties resource bundle. This sample shows how to create such a properties file.

#### Resource bundle sample

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs are inserted. All the normal properties file conventions and rules apply to this file. In addition, the creator must be aware of other restrictions that are imposed on the values by the Java MessageFormat class. For example, apostrophes must be escaped or they cause a problem. Avoid the use of non-portable characters. WebSphere Application Server does not support the use of extended formatting conventions that the MessageFormat class supports, such as {1, date} or {0,number, integer}.

Assume that the base directory for the application that uses this resource bundle is `baseDir` and that this directory is in the class path. Assume that the properties file is stored in the subdirectory `baseDir` that is

not in the class path (for example, baseDir/subDir1/subDir2/resources). To allow the messages file to resolve, the subDir1.subDir2.resources.DefaultMessage name is used to identify the property resource bundle and is passed to the message logger.

For this sample, the properties file is named DefaultMessages.properties.

```
# Contents of the DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three parameter: parm1={0}, parm2 = {1}, parm3={2}
```

When the DefaultMessages.properties file is created, the file can be sent to a translation center where the localized versions are generated.

## Changing the message IDs used in log files

You can change the default format for message IDs in server logs by setting the com.ibm.websphere.logging.messageId.version system property.

### Before you begin

**Note:** Beginning with WebSphere Application Server Version 6.0, logging files are formatted according to a standardized system. However, the default runtime behavior is still configured to use the older format. In new releases of WebSphere Application Server, the message IDs that are written to log files will be changed to ensure they do not conflict with other IBM products. The default runtime behavior is still configured to use the older message IDs, deprecated in Version 7.0.

As a result of the default runtime behavior, you might see a mixture of messages that use 4–letter message prefixes and 5–letter message prefixes. The information in this topic explains how to change your configuration so that the messages consistently show with 5–letter message prefixes. The default behavior has not changed to minimize the impact on customers that depend on the existence of the 4–letter message prefixes.

The following is a sample of an entry in a trace.log file using a default message ID. Note that the message ID is PMON0001A

```
[1/26/05 10:17:12:529 EST] 0000000a PMIImpl       A   PMON0001A: PMI is enabled
```

A sample of the same entry using a new message ID follows. Note that the message ID is CWPMI0001A. All new WebSphere Application Server message IDs begin with 'CW'.

```
[1/26/05 10:17:12:529 EST] 0000000a PMIImpl       A   CWPMI0001A: PMI is enabled.
```

### About this task

If you are using a logging tool that uses the standardized format, you might want to change the default configuration settings to format the logging output appropriately. You will need to change the configuration for each Java virtual machine (JVM) in the cell if you want the output formatting to be the same across application servers.

- To configure logging files so that they use the newer, 5–letter error message prefixes for each process, use the following commands with the wsadmin utility:
  - Using Jacl:

```
$AdminConfig list JavaVirtualMachine
set cfgJvm [$AdminConfig list "JavaVirtualMachine"]
$AdminConfig create Property $cfgJvm {{name com.ibm.websphere.logging.messageId.version} {value 6} {required false}}
$AdminConfig save
```

  - Using Jython:

```
ls = java.lang.System.getProperty("line.separator")
cfgJvmList = AdminConfig.list("JavaVirtualMachine").split(ls)
print cfgJvmList
cfgJvm = cfgJvmList[JavaVirtualMachine]
AdminConfig.create('Property', cfgJvm, [['name', 'com.ibm.websphere.logging.messageId.version'], ['value', '6'], ['required', 'false']]
AdminConfig.save()
```

Where *JavaVirtualMachine* is the number of the process that you want to use.

When you specify the process, the first process listed is zero (0), the second process is one (1), and so on. Make the changes for each JVM in the cell for consistent output formatting.

**Note:** Restart the application server for the changes to take effect.

- To change the configuration so that the log files contain the newer, 5–letter message prefixes in the startServer.log or stopServer.log files, modify the startServer and stopServer scripts in the *install_root*/bin directory. Within these files, add the following line of code:

  ```
  >> %TMPJAVAPROPFILE% echo com.ibm.websphere.logging.messageId.version=6
  ```

  **Note:** Restart the application server for the changes to take effect.

## Results

Message IDs written to log files will now be compliant with the new standard.

### *Converting log files to use IBM unique Message IDs:*

The convertlog command creates a new log file with either new or old message IDs substituted in place of the message IDs in the source file.

#### Before you begin

Prior to Version 6.x, components were assigned message IDs that are not necessarily unique across IBM software products. In Version 6.0, a system property was provided to map the message IDs in output logs to a set of IBM unique message IDs (all WebSphere Application Server message IDs now start with CW) that do not conflict with other IBM software products. The default runtime behavior still uses the old message IDs.

#### About this task

To facilitate the migration of logging tools that are reliant on the old message IDs, the convertlog command is provided to convert the message IDs of log entries from the old standard to the new standard, or the new standard back to the old. By default, the software is configured to use the old message IDs when logging, but you can change the default output with the com.ibm.websphere.logging.messageId.version system property. Read “Changing the message IDs used in log files” on page 16 for more information.

Use the convertlog command to convert the log output:

```
convertlog <source file name> <destination file name> [options]
        options: -newMessageFormat convert message IDs to CCCCCnnnnS format
                  (cannot be used with -m5)
                 -oldMessageFormat convert message IDs to CCCCnnnnS format
                  (cannot be used with -m6)
```

#### Results

After using the convertlog command you have a new file with message IDs in the chosen format.

### *convertlog command:*

The convertlog command is used to convert the message IDs in log entries from the old standard to the new standard, or the new standard back to the old.

Previous versions of WebSphere Application Server used message IDs that are deprecated in WebSphere Application Server Version 7.0. To facilitate the migration of tools based on the old message IDs, the convertlog command is implemented to translate log files from one message ID standard to the other.

Use the convertlog command as follows:

```
convertlog <source file name> <destination file name> [options]
        options: -newMessageFormat convert message IDs to CCCCCnnnnS format
                  (cannot be used with -m5)
                 -oldMessageFormat convert message IDs to CCCCnnnnS format
                  (cannot be used with -m6)
```

***MessageConverter class:***

The com.ibm.websphere.logging.MessageConverter class provides a method to convert a message ID at the front of a String into either a new message ID or an old message ID. The direction of the conversion is controlled with the *conversionType* argument.

Use the MessageConverter class with log analysis tools to convert message IDs from earlier versions of WebSphere Application Server into the corresponding message IDs that are used in later releases, or to revert message IDs to an earlier format.

**Method**

`public static java.lang.String` **`convert`**`(java.lang.String` *`in`*`, short` *`conversionType`*`)`

**Parameters**

Use the following parameters with the `MessageConverter` class:

| Parameter Name | Description |
|---|---|
| *in* | The message to convert. The method assumes the message ID is the first part of the supplied message with no leading white space. |
| *conversionType* | CONVERSION_TYPE_WASV5_TO_WASV6 |
| | CONVERSION_TYPE_WASV6_TO_WASV5 |

## Example: Creating custom log handlers with java.util.logging

There may be occasions when you want to propagate log records to your own log handlers rather than participate in integrated logging.

To use a stand-alone log handler, set the useParentHandlers flag to `false` in your application.

The mechanism for creating a customer handler is the Handler class support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with handlers, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.logging API.

The following sample shows a custom handler:

```
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

/**
 * MyCustomHandler outputs contents to a specified file
 */
public class MyCustomHandler extends Handler {
```

```
 FileOutputStream fileOutputStream;
 PrintWriter printWriter;

 public MyCustomHandler(String filename) {
  super();

  // check input parameter
  if (filename == null || filename == "")
   filename = "mylogfile.txt";

  try {
   // initialize the file
   fileOutputStream = new FileOutputStream(filename);
   printWriter = new PrintWriter(fileOutputStream);
    setFormatter(new SimpleFormatter());
  }
  catch (Exception e) {
   // implement exception handling...
  }
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#publish(java.util.logging.LogRecord)
  */
 public void publish(LogRecord record) {
  // ensure that this log record should be logged by this Handler
  if (!isLoggable(record))
   return;

  // Output the formatted data to the file
  printWriter.println(getFormatter().format(record));
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#flush()
  */
 public void flush() {
  printWriter.flush();
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#close()
  */
 public void close() throws SecurityException {
  printWriter.close();
 }
}
```

## Example: Creating custom filters with java.util.logging

A custom filter provides optional, secondary control over what is logged, beyond the control that is provided by the level.

The mechanism for creating a customer filter is the Filter interface support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with filters, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation the for the java.util.logging API.

The following example shows a custom filter:

```
/**
 * This class filters out all log messages starting with SECJ022E, SECJ0373E, or SECJ0350E.
 */
import java.util.logging.Filter;
import java.util.logging.Handler;
import java.util.logging.Logger;
import java.util.logging.LogRecord;
```

```
public class MyFilter implements Filter {
 public boolean isLoggable(LogRecord lr) {
  String msg = lr.getMessage();
  if (msg.startsWith("SECJ0222E") || msg.startsWith("SECJ0373E") || msg.startsWith("SECJ0350E")) {
   return false;
  }
  return true;
 }
}

//This code will register the above log filter with the root Logger's handlers (including the WAS system logs):
...
Logger rootLogger = Logger.getLogger("");
rootLogger.setFilter(new MyFilter());
```

## Example: Creating custom formatters with java.util.logging

A formatter formats events. Handlers are associated with one or more formatters.

The mechanism for creating a customer formatter is the Formatter class support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with formatters, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.logging API.

The following example shows a custom formatter:

```
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;

/**
 * MyCustomFormatter formats the LogRecord as follows:
 * date   level   localized message with parameters
 */
public class MyCustomFormatter extends Formatter {

 public MyCustomFormatter() {
  super();
 }

 public String format(LogRecord record) {

  // Create a StringBuffer to contain the formatted record
  // start with the date.
  StringBuffer sb = new StringBuffer();

  // Get the date from the LogRecord and add it to the buffer
  Date date = new Date(record.getMillis());
  sb.append(date.toString());
  sb.append(" ");

  // Get the level name and add it to the buffer
  sb.append(record.getLevel().getName());
  sb.append(" ");

  // Get the formatted message (includes localization
  // and substitution of paramters) and add it to the buffer
  sb.append(formatMessage(record));
  sb.append("\n");

  return sb.toString();
 }
}
```

## Example: Adding custom handlers, filters, and formatters

In some cases you might want to have your own custom log files. Adding custom handlers, filters, and formatters enables you to customize your logging environment beyond what can be achieved by the configuration of the default WebSphere Application Server logging infrastructure.

The following example demonstrates how to add a new handler to process requests to the com.myCompany subtree of loggers (see “Configuring the logger hierarchy” on page 13). The main method in this sample gives an example of how to use the newly configured logger.

```
import java.util.Vector;
import java.util.logging.Filter;
import java.util.logging.Formatter;
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyCustomLogging {

 public MyCustomLogging() {
  super();
 }

 public static void initializeLogging() {

  // Get the logger that you want to attach a custom Handler to
  String defaultResourceBundleName = "com.myCompany.Messages";
  Logger logger = Logger.getLogger("com.myCompany", defaultResourceBundleName);

  // Set up a custom Handler (see MyCustomHandler example)
  Handler handler = new MyCustomHandler("MyOutputFile.log");

  // Set up a custom Filter (see MyCustomFilter example)
  Vector acceptableLevels = new Vector();
  acceptableLevels.add(Level.INFO);
  acceptableLevels.add(Level.SEVERE);
  Filter filter = new MyCustomFilter(acceptableLevels);

  // Set up a custom Formatter (see MyCustomFormatter example)
  Formatter formatter = new MyCustomFormatter();

  // Connect the filter and formatter to the handler
  handler.setFilter(filter);
  handler.setFormatter(formatter);

  // Connect the handler to the logger
  logger.addHandler(handler);

  // avoid sending events logged to com.myCompany showing up in WebSphere
  // Application Server logs
  logger.setUseParentHandlers(false);

 }

 public static void main(String[] args) {
  initializeLogging();

  Logger logger = Logger.getLogger("com.myCompany");

  logger.info("This is a test INFO message");
  logger.warning("This is a test WARNING message");
  logger.logp(Level.SEVERE, "MyCustomLogging", "main", "This is a test SEVERE message");
 }
}
```

When the above program is run, the output of the program is written to the MyOutputFile.log file. The content of the log is in the expected log file, as controlled by the custom handler, and is formatted as defined by the custom formatter. The warning message is filtered out, as specified by the configuration of the custom filter. The output is as follows:

```
C:\>type MyOutputFile.log
Sat Sep 04 11:21:19 EDT 2004 INFO This is a test INFO message
Sat Sep 04 11:21:19 EDT 2004 SEVERE This is a test SEVERE message
```

# HTTP error, FRCA, and NCSA access log settings

Use this page to configure the global HTTP error log, and National Center for Supercomputing Applications (NCSA) access log settings for an HTTP inbound channel. If you are running the product on z/OS, you can also use this page to configure the global Fast Response Cache Accelerator (FRCA) log settings for an HTTP inbound channel. FRCA logs are a specialized form of NCSA logs and can only be created in a z/OS environment.

To view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* **> HTTP error, NCSA access and FRCA logging**. This console page has separate sections for each type of logging. The FRCA logging section only appears if you are running the product on z/OS.

The HTTP error log contains a record of HTTP processing errors that occur. The level of error logging that occurs is dependent on the value that is selected for the Error log level field.

The NCSA access log contains a record of all inbound client requests that the HTTP transport channel handles. All of the messages that are contained in these logs are in NCSA format.

After you configure the HTTP error log, NCSA access logs, and FRCA logs, you must explicitly enable each type of logging on the settings page for the HTTP channels for which you want a specific types of logging to occur. To view the settings page for an HTTP channel, click **Servers > Server Types > Application servers >** *server* **> Web Container Settings > Web container transport chains > HTTP inbound channel**.

**Note:** The settings for any of these logs can also be modified on the settings page for a specific HTTP inbound channel. Any changes that you make on the HTTP inbound channel settings page only apply to that specific inbound channel. and override any global configuration settings that you specify on this page.

## Enable logging service at server start-up

Select this option if you want any of the following logging to start when the server starts:

- NCSA access logging
- HTTP error logging

**Note:** Even if you select this option, you must explicitly enable the type of logging that you want to occur on this page and on the settings page for the HTTP transport channel for which you want that type of logging to occur.

## Enable NCSA access logging

When selected, a record of inbound client requests that the HTTP transport channel handles is kept in the NCSA access log.

## NCSA access log file path

Specifies the directory path and name of the NCSA access log. Standard variable substitutions, such as $(SERVER_LOG_ROOT), can be used when specifying the directory path.

## NCSA access log maximum size

Specifies the maximum size, in megabytes, of the NCSA access log. When this size is reached, the *logfile_name*.1 archive log is created. However, every time that the original log file overflows this archive file, the file is overwritten with the most current version of the original log file.

## Maximum number of historical files

Specifies the maximum number of historical versions of the NCSA access log file that are kept for future reference.

### NCSA access log format
Specifies which NCSA format is used when logging client access information. If you select Common, the log entries contain the requested resource and a few other pieces of information, but does not contain referral, user agent, and cookie information. If you select Combined, referral, user agent, and cookie information is included.

### Enable error logging
When selected, HTTP errors that occur while the HTTP channel processes client requests are recorded in the HTTP error log.

### Error log file path
Specifies the directory path and the name of the HTTP error log. Standard variable substitutions, such as $(SERVER_LOG_ROOT), can be used when specifying the directory path.

### Error log maximum size
Specifies the maximum size, in megabytes, of the HTTP error log file. When this size is reached, the *logfile_name*.1 archive log is created. However, every time that the original log file overflows this archive file, this file is overwritten with the most current version of the original log file.

### Maximum number of historical files
Specifies the maximum number of historical versions of the Error log file that are kept for future reference.

### Error log level
Specifies the type of error messages that are included in the HTTP error log.

You can select:

**Critical**
Only critical failures that stop the Application Server from functioning properly are logged.

**Error** The errors that occur in response to clients are logged. These errors require Application Server administrator intervention if they result from server configuration settings.

**Warning**
Information on general errors, such as socket exceptions that occur while handling client requests, are logged. These errors do not typically require Application Server administrator intervention.

**Information**
The status of the various tasks that are performed while handling client requests is logged.

**Debug**
More verbose task status information is logged. This level of logging is not intended to replace RAS logging for debugging problems, but does provide a steady status report on the progress of individual client requests. If this level of logging is selected, you must specify a large enough log file size in the Error log maximum size field to contain all of the information that is logged.

## Logger.properties file for configuring logger settings
Use the Logger.properties file to set logger attributes for specific loggers.

The properties file is loaded the first time that the Logger.getLogger(*logger_name*) method is called within an application.

**Important:** The name of the Logger.properties file is case sensitive. Use a capital ″L″ in the file name.

When an application calls the Logger.getLogger method for the first time, all the available logger properties files are loaded. Applications can provide Logger.properties files in:

- the META-INF directory of the Java archive (JAR) file for the application
- directories included in the class path of an application module

- directories included in the application class path

The properties file contains two categories of parameters, logger control and logger data:
- Logger control information
  - Minimum localization level: The minimum LogRecord level for which localization is attempted
  - Group: The logical group that this component belongs to
  - Event factory: The Common Base Event template file to use with the event factory. The naming convention for this template is the fully qualified component name, with a file extension of `.event.xml`. For example, a template that applies to the com.ibm.compXYZ package is called `com.ibm.compXYZ.event.xml`.
- Logger data information
  - Product name
  - Organization name
  - Component name
  - Extensions and additional properties

### Syntax of the Logger.properties file

Use the following syntax to set logger properties:

`<`*`logger base name`*`>.<`*`property`*`>=value`

where:

*`logger base name`* is the starting part of the logger name to which the property applies. All loggers with names starting with this string have the property applied.

*`property`* is one of the following properties:
- organization
- product
- component
- minimum_localization_level
- group
- eventfactory

### Sample Logger.properties file

In the following sample, the com.ibm.xyz.MyEventFactory event factory is used by any loggers in the com.ibm.websphere.abc package or any sub packages that do not override this value in their configuration file.

```
com.ibm.websphere.abc.eventfactory=com.ibm.xyz.MyEventFactory
```

### Group Logger.properties file

In the following example, the group is MyTraceGroup and the components are com.ibm.stuff and com.ibm.morestuff:

```
com.ibm.stuff.group=MyTraceGroup
com.ibm.morestuff.group=MyTraceGroup
```

## Example: Sample security policy for logging

Set up a security policy to allow your applications to modify logging and handler properties.

The sample security policy that follows grants access to the file system and runtime classes. Include this security policy, with the entry `permission java.util.logging.LoggingPermission "control"`, in the META-INF directory of your application if you want your applications to programmatically alter controlled properties of loggers and handlers. The META-INF file is located in the following locations for the different module types:

| | |
|---|---|
| EJB projects | ejbModule/META-INF/MANIFEST.MF |
| Application client projects | appClientModule/META-INF/MANIFEST.MF |
| Dynamic Web projects | WebContent/META-INF/MANIFEST.MF |
| Connector projects | connectorModule/META-INF/MANIFEST.MF |

Below is a sample security policy that grants permission to modify logging properties:

```
////////////////////////////////////////////////
//
// WebSphere Application Server Security Policy
//
////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// Allow all access to the file system and runtime classes
/////////////////////////////////////////////////////////////////////
grant codeBase "file:${application}" {
   permission java.util.logging.LoggingPermission "control";
};
```

# Configuring applications to use Jakarta Commons Logging

Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging systems. WebSphere Application Server supports Jakarta Commons Logging by providing a logger. The support does not change interfaces defined by Jakarta Commons Logging.

## Before you begin

The WebSphere Application Server logger is a thin wrapper for the WebSphere Application Server logging facility. The logger name is *com.ibm.websphere.commons.logging.WsJDK14Logger*. The logger can handle logging objects defined by either of the following:

- Java Logging found in Java Specification Request 47: Logging API Specification
- Common Base Event

A *logging object* is an object that holds logging entry information.

To better understand Jakarta Commons Logging, read Jakarta Commons and the specifications for Java Logging and for Common Base Event. To better understand use of the WebSphere Application Server logger, read “Jakarta Commons Logging” on page 26.

## About this task

WebSphere Application Server provides the Jakarta Commons Logging binary distribution in its `libraries` directory. By default, the product uses the Jakarta Commons Logging LogFactory implementation and JDK14Logger.

**Note:** The default configuration of Jakarta Commons Logging is stored in the commons-logging.properties file. To specify the factory class to use with Jakarta Commons Logging in an application, provide a file named org.apache.commons.logging.LogFactory, located in META-INF/services directory, that

contains the name of the factory class on the first line. This is the configuration mechanism for the JAR file service provider, as defined in JDK 1.3 and above.

For an application to use the WebSphere Application Server logger, the application must provide its own configuration for the logger. To configure an application to use the WebSphere Application Server logger, complete the steps that follow.

1. Examine “Configurations for the WebSphere Application Server logger” on page 28 and determine which configuration best suits your application.
2. Change your application configuration as needed to enable use of the WebSphere Application Server logger.

### Results

After the application starts, Jakarta Commons Logging routes the application’s logging output to the WebSphere Application Server logger.

## Jakarta Commons Logging

Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging systems. The logging interface enables application logging to be simple and independent of the logging system that the application uses. You can change the logging implementation for a deployed application without having to change the application logging code. However, the simplicity of the logging interface prevents the application from leveraging all the functionality of the logging systems.

This topic provides the following information about Jakarta Commons Logging in WebSphere Application Server:



### Support for Jakarta Commons Logging

The product supports Jakarta Commons Logging by providing a logger, a thin wrapper for the WebSphere Application Server logging facility. The logger can handle both Java Logging (JSR-47) and Common Base Event logging objects. A *logging object* is an object that holds logging entry information.

The product support for Jakarta Commons Logging does not change interfaces defined by Jakarta Commons Logging.

### Benefits of support for Jakarta Commons Logging

The WebSphere Application Server support for Jakarta Commons Logging provides the following benefits:

- WebSphere Application Server is pre-configured to use Jakarta Commons Logging.

  All of the functionality of Jakarta Commons Logging is provided for any application or WebSphere Application Server component. Logging calls are routed by default to the underlying WebSphere Application Server logging facility.
- A logger that uses the WebSphere Application Server logging facility.

  Applications and components can pass both Java Logging and Common Base Event logging objects to the WebSphere Application Server logger without conversion to strings, providing applications with enhanced logging. Further, Jakarta Commons Logging Logger levels are integrated into WebSphere Application Server administrative facilities.

## Overview of the process for using Jakarta Commons Logging

Logging with Jakarta Commons Logging consists of the steps that follow. “Configurations for the WebSphere Application Server logger” on page 28 provides details on configuring your application to use the WebSphere Application Server logger.

1. Obtain an instance of a logger factory.

   To obtain a logger factory, use Jakarta Commons Logging code. You can configure the code to meet your needs. In WebSphere Application Server, Jakarta Commons Logging is configured by default to instantiate the Jakarta Commons Logging default logger factory. Applications or WebSphere Application Server components can provide their own configuration if they use a different logger factory implementation. Applications can use more than one factory.

2. Obtain an instance of a logger.

   To obtain a logger, use code implemented by a logger factory. Configuration of the code is implementation specific.

   The WebSphere Application Server logger implements the methods defined in the logging interface. The logging methods take at least one argument, which can be any Java object. The WebSphere Application Server logger, the WsJDK14Logger logger described in “Classes used to obtain a logger factory and logger,” handles the following objects passed into the following logging methods:

   **CommonBaseEvent**
   Wrapped into CommonBaseEventLogRecord

   **CommonBaseEventLogRecord**
   Passed without change

   **LogRecord**
   Passed without change

   **Other objects**
   Converted to String

   Applications or WebSphere Application Server components can provide their own configuration if they use an implementation of a logger that is not specific to WebSphere Application Server. An application must know what factory is being used in order to configure it.

3. Start your application. Jakarta Commons Logging routes the application’s logging output to the designated logger

## Classes used to obtain a logger factory and logger

| Class name | Description |
|---|---|
| LogFactory | *LogFactory* is a Jakarta Commons Logging class that implements initialization logic. LogFactory is an abstract class that every logger factory implementation has to extend. It provides static methods for obtaining:<br>• An instance of a factory class<br>• Instances of a logger, using an instance of the factory class<br><br>LogFactory provides methods for obtaining instances of loggers, although these methods delegate the logger instantiation and configuration to an instance of a logger factory class.<br><br>Logger factories, once instantiated, are cached on a per context class loader basis. The instances in a cache can be released. This functionality is designed for platform container implementations rather than for applications. |
| LogFactoryImpl | *LogFactoryImpl* is a Jakarta Commons Logging concrete class that implements the default logger factory using methods in LogFactory. To use Java Logging, there must always be at least one instance of a logger factory class, even if the application has not explicitly obtained one. If the configuration does not name a logger factory class, LogFactoryImpl is used as the default. |

| Class name | Description |
|---|---|
| Log | *Log* is a Jakarta Commons Logging interface for loggers. Commons logging loggers have to implement the Log interface. Because the goal of Jakarta Commons Logging is to wrapper any logging system, the Log interface defines a small set of common logging methods. In WebSphere Application Server, WsJDK14Logger implements the Log interface.<br><br>Logger instantiation and configuration is specific to every logger factory. Logging in WebSphere Application Server uses the default logger factory provided in Jakarta Commons Logging, which keeps instantiated loggers in cache, on a per class loader context basis. |
| WsJDK14Logger | *WsJDK14Logger* is a WebSphere Application Server class that provides a Jakarta Commons Logging logger by implementing the Log interface. The WsJDK14Logger logger differs from the Java Logging logger in that the WsJDK14Logger logger enables Java Logging or Common Base Event objects to be passed over without converting them into String objects. This prevents any information loss the conversion to String might cause as well as allows the logging output to be more descriptive and precise. In contrast, the Java Logginglogger that is provided in Jakarta Commons Logging converts objects passed into the logging calls to String objects before passing them over to the underlying Java Logging. |

### Logger level configuration and mapping

Because Jakarta Commons Logging loggers are thin wrappers for specific logging systems, the loggers do not have their own level, but use the level of the logger from the underlying logging system. Although the underlying system can provide methods for changing level, there are no methods for changing level defined on the Log interface, which all Jakarta Commons Logging logger must implement. WsJDK14Logger uses the level of its underlying Java Logging logger.

Following table shows, on the left, the mapping of Jakarta Commons Logging levels within WsJDK14Logger to levels in the WebSphere Application Server implementation of Java Logging. On the right, it shows the levels defined in Java Logging and the level mapping in the Jakarta Commons Logging JDK14Logger to the Java Logging levels.

| WsJDK14Logger | Java Logging in WebSphere Application Server | Java Logging | JDK14Logger |
|---|---|---|---|
| Fatal | Fatal | | |
| Error | Severe | Severe | Fatal, Error |
| Warning | Warning | Warning | Warning |
| | Audit | | |
| Info | Info | Info | Info |
| | Config | Config | |
| | Detail | | |
| Debug | Fine | Fine | Debug |
| | Finer | Finer | |
| Trace | Finest | Finest | Trace |

The WsJDK14Logger level is synchronized with the underlying Java Logging logger level. WebSphere Application Server administration controls the WsJDK14Logger level.

## Configurations for the WebSphere Application Server logger

This topic describes several ways to configure an application to use the WebSphere Application Server logger.

The type of configuration that best suits an application depends upon the following:

- Whether the class loader order setting for the application is `Classes loaded with parent class loader first (Parent First)` or `Classes loaded with application class loader first (Parent Last)`, you can set the class loader delegation mode on a console page. For more details about class load order and delegation, consult the class loading chapter in the *Developing and deploying applications* PDF book
- Whether Jakarta Commons Logging is bundled with the application configuration
- Whether Jakarta Commons Logging is provided within the application

The following tables describe the conditions required to enable an application to use the WebSphere Application Server logger.

**Class loader mode is `Parent First` and Jakarta Commons Logging is bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is not read** by the LogFactory because the parent class loader finds the WebSphere properties file first.<br><br>• The class name **is read** from the file<br>`META-INF/services/org.apache.commons.logging.LogFactory` | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration, unless the configuration is provided in a META-INF file of the application or module. | The log used is either of the following:<br><br>• The Log implementation specified in the WebSphere Application Server default configuration<br><br>• An application-specific Log implementation if an application-specific LogFactory that instantiates a different Log implementation is used. | The **application parent class loader** is the first class loader to load the Jakarta Commons Logging code. The **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**. |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere default configuration. | The log used is the Log implementation specified in the WebSphere default configuration. | The Jakarta Commons Logging bundled with the application is not used. |

**Class loader mode is `Parent First` and Jakarta Commons Logging is not bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is not read** by the LogFactory because the parent class loader finds the WebSphere Application Server properties file first.<br><br>• The class name **is read** from the file<br>`META-INF/services/org.apache.commons.logging.LogFactory` | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration, unless the configuration is provided in a META-INF file of the application or module. | The log used is either of the following:<br><br>• The Log implementation specified in the WebSphere Application Server default configuration<br><br>• An application-specific Log implementation if an application-specific LogFactory that instantiates a different Log implementation is used. | The **application parent class loader** is the first class loader to load the Jakarta Commons Logging code. The **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**. |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration. | The log used is the Log implementation specified in the WebSphere Application Server default configuration. | Same as in the previous row |

**Class loader mode is `Parent Last` and Jakarta Commons Logging is bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is read** by the LogFactory because the class loader finds the application properties file first.<br><br>• The class name **is read** from the file<br>`META-INF/services/org.apache.commons.logging.LogFactory` | The log factory used is either of the following:<br><br>• The default Jakarta Commons Logging LogFactory<br><br>• The LogFactory specified in the application configuration | The log used is the Log implementation specified in the application configuration.<br><br>If the log factory used is the default Jakarta Commons Logging LogFactory, the Log implementation must be on the classpath of the application class loader. | The **application class loader** is the first class loader to load the Jakarta Commons Logging code. The **application bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **application class loader**. |

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration. | The log used is the Log implementation specified in the WebSphere Application Server default configuration. | |

**Class loader mode is `Parent Last` and Jakarta Commons Logging is not bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br>• The properties file `commons-logging.properties` in the application classpath **is read** by the LogFactory because the class loader finds the application properties file first.<br>• The class name **is read** from the file `META-INF/services/org.apache.commons.logging.LogFactory` | The log factory used is either of the following:<br>• The default Jakarta Commons Logging LogFactory<br>• The LogFactory specified in the application configuration | The log used is the Log implementation specified in the application configuration.<br>If the log factory used is the default Jakarta Commons Logging LogFactory, the Log implementation must be on the classpath of the application class loader. | There is no Jakarta Commons Logging code at the application class loader. Thus, the **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**. |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration. | The log used is the Log implementation specified in the WebSphere Application Server default configuration. | |

# Programming with the JRas framework

Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server applications.

## Before you begin

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

The JRas extensions allow message logging and diagnostic trace to work with WebSphere Application Server applications. They are based on the stand-alone JRas logging toolkit.

1. Retrieve a reference to the JRas manager.
2. Retrieve message and trace loggers by using methods on the returned manager.
3. Call the appropriate methods on the returned message and trace loggers to create message and trace entries, as appropriate.

## JRas logging toolkit

The JRas logging toolkit provides diagnostic information to help the administrator diagnose problems or tune application performance.

**Deprecated:** The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Developing, deploying, and maintaining applications are complex tasks. For example, when a running application encounters an unexpected condition, it might not be able to complete a requested operation. In such a case, you might want the application to inform the administrator that the operation failed and provide information. This action enables the administrator to take the proper corrective action. Those who develop or maintain applications might need to gather detailed information relating to the path of a running application to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as *message logging* and *diagnostic trace*.

Message logging (messages) and diagnostic trace (trace) are conceptually quite similar, but do have important differences. It is important for application developers to understand these differences to use these tools properly. To start with, the following operational definitions of messages and trace are provided.

**Message**
: A message entry is an informational record that is intended for end users, systems administrators and support personnel to view. The text of the message must be clear, concise, and interpretable. Messages are typically localized, meaning that they display in the national language of the end user. Although the destination and lifetime of messages might be configurable, some level of message logging is always enabled in normal system operation. Message logging must be used judiciously due to both performance considerations and the size of the message repository.

**Trace**
: A trace entry is an information record that is intended for service engineers or developers to use. This trace record might be considerably more complex, verbose, and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries can be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but might be enabled as needed to gather diagnostic information.

WebSphere Application Server provides a message logging and diagnostic trace API that applications can use. This API is based on the stand-alone JRas logging toolkit, which was developed by IBM. The stand-alone JRas logging toolkit is a collection of interfaces and classes that provide message logging and diagnostic trace primitives. These primitives are not tied to any particular product or platform. The stand-alone JRas logging toolkit provides a limited amount of support, which is typically referred to as *systems management support*, including log file configuration support based on property files.

As designed, the stand-alone JRas logging toolkit does not contain the support that is required for integration into the WebSphere Application Server run time or for use in a Java 2 Platform, Enterprise Edition (J2EE) environment. To overcome these limitations, WebSphere Application Server provides a set of extension classes to address these shortcomings. This collection of extension classes is referred to as the JRas extensions. The JRas extensions do not modify the interfaces that are introduced by the

stand-alone JRas logging toolkit, but provide the appropriate implementation classes. The conceptual structure that is introduced by the stand-alone JRas logging toolkit is described in the following section. It is equally applicable to the JRas extensions.

## JRas concepts

The section contains a basic overview of important concepts and constructs that are introduced by the stand-alone JRas logging toolkit. This information is not an exhaustive overview of the capabilities of this logging toolkit, nor is it intended as a detailed discussion of usage or programming paradigms. More detailed information, including code examples, is available in JRas extensions and its subtopics, including in the API documentation for the various interfaces and classes that make up the logging toolkit.

**Event types**

The stand-alone JRas logging toolkit defines a set of event types for messages and a set of event types for trace. Examples of message types include informational, warning, and error. Examples of trace types include entry, exit, and trace.

**Event classes**

The stand-alone JRas logging toolkit defines both message and trace event classes.

**Loggers**

A logger is the primary object with which the user code interacts. Two types of loggers are defined: message loggers and trace loggers. The set of methods on message loggers and trace loggers are different because they provide different functionality. Message loggers create message records only and trace loggers create trace records only. Both types of loggers contain masks that indicate which categories of events the logger processes and which to ignore. Although every JRas logger is defined to contain both a message and trace mask, the message logger uses only the message mask and the trace logger uses the trace mask only. For example, by setting a message logger message mask to the appropriate state, it can be configured to process only error messages and ignore informational and warning messages. Changing the trace mask state of a message logger has no effect.

A logger contains one or more handlers to which it forwards events for further processing. When the user calls a method on the logger, the logger compares the event type that is specified by the caller to its current mask value. If the specified type passes the mask check, the logger creates an event object to capture the information relating to the event that passed to the logger method. This information can include information, such as the names of the class and method which logs the event, a message, and parameters to log, among others. When the logger creates the event object, it forwards the event to all handlers currently registered with the logger.

Methods that are used within the logging infrastructure do not make calls to the logger method. When an application uses an object that extends a thread class, implements the hashCode method, and makes a call to the logging infrastructure from that method, the result is a recursive loop.

**Handlers**

A handler provides an abstraction over an output device or event consumer. An example is a file handler, which knows how to write an event to a file. The handler also contains a mask that is used to further restrict the categories of events the handler processes. For example, a message logger might be configured to pass both warning and error events, but a handler attached to the message logger might be configured to pass error events only. Handlers also include formatters, which the handler invokes to format the data in the passed event before it is written to the output device.

**Formatters**

Handlers are configured with formatters, which know how to format events of certain types. A handler can contain multiple formatters, each of which knows how to format a specific class of event. The event object is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which then writes it to the output device.

# JRas Extensions

*JRas extensions* are the collection of implementation classes that support JRas integration into the WebSphere Application Server environment.

## JRas extensions

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The stand-alone JRas logging toolkit defines interfaces and provides a variety of concrete classes that implement these interfaces. Because the stand-alone JRas logging toolkit is developed as a general purpose toolkit, the implementation classes do not contain the configuration interfaces and methods that are necessary for use in the WebSphere Application Server product. In addition, many of the implementation classes are not written appropriately for use in a Java 2 Platform, Enterprise Edition (J2EE) environment. To overcome these shortcomings, WebSphere Application Server provides the appropriate implementation classes that support integration into the WebSphere Application Server environment. The collection of these implementation classes is referred to as the *JRas extensions*.

## Usage model

You can use the JRas extensions in three distinct operational modes:

**Integrated**
: In this mode, message and trace records are written only to logs that are defined and maintained by the WebSphere Application Server run time. This mode is the default mode of operation and is equivalent to the WebSphere Application Server V4.0 mode of operation.

**stand-alone**
: In this mode, message and trace records are written solely to stand-alone logs that are defined and maintained by the user. You control which categories of events are written to which logs, and the format in which entries are written. You are responsible for configuration and maintenance of the logs. Message and trace entries are not written to WebSphere Application Server runtime logs.

**Combined**
: In this mode, message and trace records are written to both WebSphere Application Server runtime logs and to stand-alone logs that you must define, control, and maintain. You can use filtering controls to determine which categories of messages and trace are written to which logs.

The JRas extensions are specifically targeted to an integrated mode of operation. The integrated mode of operation can be appropriate for some usage scenarios, but many scenarios are not adequately addressed by these extensions. Many usage scenarios require a stand-alone or combined mode of operation instead. A set of user extension points are defined that support JRas extensions in either a stand-alone or combined mode of operations.

## JRas extension classes

WebSphere Application Server provides a base set of implementation classes that are collectively referred to as the *JRas extensions*. Many of these classes provide the appropriate implementations of loggers, handlers, and formatters for use in a WebSphere Application Server environment.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The collection of JRas classes is targeted at an integrated mode of operation. If you choose to use the JRas extensions in either stand-alone or combined mode, you can reuse the logger and manager class that are provided by the extensions, but you must provide your own implementations of handlers and formatters.

## WebSphere Application Server message and trace loggers

The message and trace loggers that are provided by the stand-alone JRas logging toolkit cannot be directly used in the WebSphere Application Server environment. The JRas extensions provide the appropriate logger implementation classes. Instances of these message and trace logger classes are obtained directly and exclusively from the WebSphere Application Server Manager class. You cannot directly instantiate message and trace loggers. Obtaining loggers in any manner other than directly from the Manager class is not allowed and directly violates the programming model.

The message and trace logger instances that are obtained from the WebSphere Application Server Manager class are subclasses of the RASMessageLogger and RASTraceLogger classes that are provided by the stand-alone JRas logging toolkit. The RASMessageLogger and RASTraceLogger classes define the set of methods that are directly available. Public methods that are introduced by the JRas extensions logger subclasses cannot be called directly by user code because it is a violation of the programming model.

Loggers are named objects and are identified by name. When the Manager class is called to obtain a logger, the caller is required to specify a name for the logger. The Manager class maintains a name-to-logger instance mapping. Only one instance of a named logger is ever created within the lifetime of a process. The first call to the Manager class with a particular name results in the logger, which is configured by the Manager class. The Manager class caches a reference to the instance, then returns it to the caller. Subsequent calls to the Manager class that specify the same name result in a returned reference to the cached logger. Separate namespaces are maintained for message and trace loggers. You can use a single name obtain both a message logger and a trace logger from the Manager, without ambiguity, and without causing a namespace collision.

In general, loggers have no predefined granularity or scope. A single logger can be used to instrument an entire application. You might determine that having a logger per class is more effective, or the appropriate granularity might be somewhere in between. Partitioning an application into logging domains is determined by the application writer.

The WebSphere Application Server logger classes that are obtained from the Manager class are thread-safe. Although the loggers provided as part of the stand-alone JRas logging toolkit implement the serializable interface, loggers are not serializable. Loggers are stateful objects, tied to a Java virtual machine instance and are not serializable. Attempting to serialize a logger is a violation of the programming model.

Personal or individual logger subclasses are not supported in a WebSphere Application Server environment.

## WebSphere Application Server handlers

WebSphere Application Server provides the appropriate handler class that is used to write message and trace events to the WebSphere Application Server run time logs. You cannot configure the WebSphere Application Server handler to write to any other destination. The creation of a WebSphere Application Server handler is a restricted operation and is not available to user code. Every logger that is obtained from the Manager comes preconfigured with an instance of this handler already installed. You can remove the WebSphere Application Server handler from a logger when you want to run in stand-alone mode. When you remove it, you cannot add the WebSphere Application Server handler again to the logger from which it is removed or any other logger. Also, you cannot directly call any method on the WebSphere Application Server handler. Attempting to create an instance of the WebSphere Application Server handler, to call methods on the WebSphere Application Server handler or to add a WebSphere Application Server handler to a logger by user code is a violation of the programming model.

### WebSphere Application Server formatters

The WebSphere Application Server handler comes preconfigured with the appropriate formatter for data that is written to WebSphere Application Server logs. The creation of a WebSphere Application Server formatter is a restricted operation and not available to user code. No mechanism exists that allows the user to obtain a reference to a formatter installed in a WebSphere Application Server handler, or to change the formatter a WebSphere Application Server handler is configured to use.

### WebSphere Application Server manager

WebSphere Application Server provides a Manager class in the com.ibm.websphere.ras package. All message and trace loggers must be obtained from this Manager class. A reference to the Manager class is obtained by calling the static Manager.getManager method. Message loggers are obtained by calling the createRASMessageLogger method on the Manager class. Trace loggers are obtained by calling the createRASTraceLogger method on the Manager class.

The manager also supports a *group* abstraction that is useful when dealing with trace loggers. The group abstraction supports multiple, unrelated trace loggers to register as part of a named entity called a *group*. WebSphere Application Server provides the appropriate systems management facilities to manipulate the trace setting of a group, similar to the way the trace settings of an individual trace logger work.

For example, suppose component A consists of 10 classes. Suppose each class is configured to use a separate trace logger. All 10 trace loggers in the component are registered as members of the same group, for example, Component_A_Group. You can turn on trace for a single class, or you can turn on trace for all 10 classes in a single operation using the group name, if you want a component trace. Group names are maintained within the namespace for trace loggers.

## JRas framework (deprecated)

Because the JRas extensions classes do not provide the flexibility and behavior that are required for many scenarios, a variety of extension points are defined. You can write your own implementation classes to obtain the required behavior.

**Deprecated:** The JRas framework described in this topic is deprecated. However, you can achieve similar results using Java logging.

In general, the JRas extensions require you to call the Manager class to obtain a message logger or trace logger. No provision is made for you to provide your own message or trace logger subclasses. In general, user-provided extensions cannot be used to affect the integrated mode of operation. The behavior of the integrated mode of operation is solely determined by the WebSphere Application Server run time and the JRas extensions classes.

### Handlers

The stand-alone JRas logging toolkit defines the RASIHandler interface. All handlers must implement this interface. You can write your own handler classes that implement the RASIHandler interface. Directly create instances of user-defined handlers and add them to the loggers that are obtained from the Manager class.

The stand-alone JRas logging toolkit provides several handler implementation classes. These handler classes are inappropriate for use in the Java 2 Platform, Enterprise Edition (J2EE) environment. You cannot directly use or subclass any of the Handler classes that are provided by the stand-alone JRas logging toolkit. Doing so is a violation of the programming model.

### Formatters

The stand-alone JRas logging toolkit defines the RASIFormatter interface. All formatters must implement this interface. You can write your own formatter classes that implement the RASIFormatter interface. You can add these classes to a user-defined handler only. WebSphere Application Server handlers cannot be configured to use user-defined formatters. Instead, directly create instances of your formatters and add them to the your handlers appropriately.

As with handlers, the stand-alone JRas logging toolkit provides several formatter implementation classes. Direct use of these formatter classes is not supported.

### Message event types

The stand-alone JRas toolkit defines message event types in the RASIMessageEvent interface. In addition, the WebSphere Application Server reserves a range of message event types for future use. The RASIMessageEvent interface defines three types, with values of `0x01`, `0x02`, and `0x04`. The values `0x08` through `0x8000` are reserved for future use. You can provide your own message event types by extending this interface appropriately. User-defined message types must have a value of `0x1000` or greater.

Message loggers that are retrieved from the Manager class have their message masks set to `pass` or process all message event types defined in the RASIMessageEvent interface. To process user-defined message types, you must manually set the message logger mask to the appropriate state by user code after the message logger is obtained from the Manager class. WebSphere Application Server does not provide any built-in systems management support for managing message types.

### Message event objects

The stand-alone JRas toolkit provides a RASMessageEvent implementation class. When a message logging method is called on the message logger, and the message type is currently enabled, the logger creates and distributes an event of this class to all handlers that are currently registered with that logger.

You can provide your own message event classes, but they must implement the RASIEvent interface. You must directly create instances of such user-defined message event classes. When it is created, pass your message event to the message logger by calling the message logger's fireRASEvent method directly. WebSphere Application Server message loggers cannot directly create instances of user-defined types in response to calling a logging method (`msg.message`) on the logger. In addition, instances of user-defined message types are never processed by the WebSphere Application Server handler. You cannot create instances of the RASMessageEvent class directly.

### Trace event types

The stand-alone JRas toolkit defines trace event types in the RASITraceEvent interface. You can provide your own trace event types by extending this interface appropriately. In such a case, you must ensure that the values for the user-defined trace event types do not collide with the values of the types that are defined in the RASITraceEvent interface.

Trace loggers that are retrieved from the Manager class typically have their trace masks set to reject all types. A different starting state can be specified by using WebSphere Application Server systems management facilities. In addition, you can change the state of the trace mask for a logger at run-time, using WebSphere Application Server systems management facilities.

To process user-defined trace types, the trace logger mask must be manually set to the appropriate state by user code. WebSphere Application Server systems management facilities cannot be used to manage user-defined trace types, either at start time or run time.

## Trace event objects

The stand-alone JRas toolkit provides a RASTraceEvent implementation class. When a trace logging method is called on the WebSphere Application Server trace logger and the type is currently enabled, the logger creates and distributes an event of this class to all the handlers that are currently registered with that logger.

You can provide your own trace event classes. Such trace event classes must implement the RASIEvent interface. You must create instances of such user-defined event classes directly. When it is created, pass the trace event to the trace logger by calling the trace logger's fireRASEvent method directly. WebSphere Application Server trace loggers cannot directly create instances of user-defined types in response to calling a trace method (`entry`, `exit`, `trace`) on the trace logger. In addition, instances of user-defined trace types are never processed by the WebSphere Application Server handler. You cannot create instances of the RASTraceEvent class directly.

## User defined types, user defined events and WebSphere Application Server

By definition, the WebSphere Application Server handler processed user-defined message or trace types, or user-defined message or trace event classes. Message and trace entries of either a user-defined type or user-defined event class cannot be written to the WebSphere Application Server run-time logs.

***JRas programming interfaces for logging (deprecated):***

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### General considerations

You can configure the WebSphere Application Server to use Java 2 security to restrict access to protected resources such as the file system and sockets. Because user-written extensions typically access such protected resources, user-written extensions must contain the appropriate security checking calls, using AccessController doPrivileged calls. In addition, the user-written extensions must contain the appropriate policy file. In general, locating user-written extensions in a separate package is a good practice. It is your responsibility to restrict access to the user-written extensions appropriately.

### Writing a handler

User-written handlers must implement the RASIHandler interface. The RASIHandler interface extends the RASIMaskChangeGenerator interface, which extends the RASIObject interface. A short discussion of the methods that are introduced by each of these interfaces follows, along with implementation pointers. For more in-depth information on any of the particular interfaces or methods, see the corresponding product API documentation.

### RASIObject interface

The RASIObject interface is the base interface for stand-alone JRas logging toolkit classes that are stateful or configurable, such as loggers, handlers, and formatters.

- The stand-alone JRas logging tookit supports rudimentary properties-file based configuration. To implement this configuration support, the configuration state is stored as a set of key-value pairs in a properties file. The public Hashtable getConfig and public void setConfig(Hashtable ht) methods are used to get and set the configuration state. The JRas extensions do not support properties-based configuration. Implement these methods as no-operations. You can implement your own properties-based configuration using these methods.
- Loggers, handlers, and formatters can be named objects. For example, the JRas extensions require the user to provide a name for the loggers that are retrieved from the manager. You can name your handlers. The public String getName and public void setName(String name) methods are provided to

get or set the name field. The JRas extensions currently do not call these methods on user handlers. You can implement these methods as you want, including as no operations.
- Loggers, handlers, and formatters can also contain a description field. The public String getDescription and public void setDescription(String desc) methods can be used to get or set the description field. The JRas extensions currently do not use the description field. You can implement these methods as you want, including as no operations.
- The public String getGroup method is provided for use by the RASManager interface. Since the JRas extensions provide their own Manager class, this method is never called. Implement this as a no-operation.

**RASIMaskChangeGenerator interface**

The RASIMaskChangeGenerator interface is the interface that defines the implementation methods for filtering of events based on a mask state. It is currently implemented by both loggers and handlers. By definition, an object that implements this interface contains both a message mask and a trace mask, although both need not be used. For example, message loggers contain a trace mask, but the trace mask is never used because the message logger never generates trace events. Handlers, however, can actively use both mask values. For example, a single handler can handle both message and trace events.
- The public long getMessageMask and public void setMessageMask(long mask) methods are used to get or set the value of the message mask. The public long getTraceMask and public void setTraceMask(long mask) methods are used to get or set the value of the trace mask.

In addition, this interface introduces the concept of *calling back* to interested parties when a mask changes state. The callback object must implement the RASIMaskChangeListener interface.
- The public void addMaskChangeListener(RASIMaskChangeListener listener) and public void removeMaskChangeListener(RASIMaskChangeListener listener) methods are used to add or remove listeners to the handler. The public Enumeration getMaskChangeListeners method returns an enumeration over the list of currently registered listeners. The public void fireMaskChangedEvent(RASMaskChangeEvent mc) method is used to call back all the registered listeners to inform them of a mask change event.

For efficiency reasons, the JRas extensions message and trace loggers implement the RASIMaskChangeListener interface. The logger implementations maintain a composite mask in addition to the logger mask. The logger composite mask is formed by logically *or'ing* the appropriate masks of all handlers that are registered to that logger, then *and'ing* the result with the logger mask. For example, the message logger composite mask is formed by or'ing the message masks of all handlers that are registered with that logger, then and'ing the result with the logger message mask.

All handlers are required to properly implement these methods. In addition, when a user handler is instantiated, the logger that is added must be registered with the handler; use the addMaskChangeListener method. When either the message mask or trace mask of the handler is changed, the logger must be called back to inform it of the mask change. With this process, the logger can dynamically maintain the composite mask.

The RASMaskChangedEvent class is defined by the stand-alone JRas logging toolkit. Direct use of that class by user code is supported in this context.

In addition, the RASIMaskChangeGenerator interface introduces the concept of caching the names of all message and trace event classes that the implementing object process. The intent of these methods is to support a management program such as a graphical user interface to retrieve the list of names, introspect the classes to determine the event types that they might possibly process and display the results. The JRas extensions do not ever call these methods, so they can be implemented as no operations.
- The public void addMessageEventClass(String name) and public void removeMessageEventClass(String name) methodscan be called to add or remove a message event class name from the list. The method public Enumeration getMessageEventClasses returns an enumeration over the list of message event class names. Similarly, the public void

addTraceEventClass(String name) and public void removeTraceEventClass(String name) methods can be called to add or remove a trace event class name from the list. The public Enumeration getTraceEventClasses method returns an enumeration over the list of trace event class names.

**RASIHandler interface**

The RASIHandler interface introduces the methods that are specific to the behavior of a handler.

The RASIHandler interface, as provided by the stand-alone JRas logging toolkit, supports handlers that run in either a synchronous or asynchronous mode. In asynchronous mode, events are typically queued by the calling thread and then written by a worker thread. Because spawning of threads is not supported in the WebSphere Application Server environment, it is expected that handlers do not queue or batch events, although this activity is not expressly prohibited.

- The public int getMaximumQueueSize() and public void setMaximumQueueSize(int size) methods create `IllegalStateException` exceptions to manage the maximum queue size. The public int getQueueSize method is provided to query the actual queue size.
- The public int getRetryInterval and public void setRetryInterval(int interval) methods support the notion of error retry, which implies some type of queueing.
- The public void addFormatter(RASIFormatter formatter), public void removeFormatter(RASIFormatter formatter) and public Enumeration getFormatters methods are provided to manage the list of formatters that the handler can be configured with. Different formatters can be provided for different event classes, if appropriate.
- The public void openDevice, public void closeDevice and public void stop methods are provided to manage the underlying device that the handler abstracts.
- The public void logEvent(RASIEvent event) and public void writeEvent(RASIEvent event) methods are provided to pass events to the handler for processing.

**Writing a formatter**

User-written formatters must implement the RASIFormatter interface. The RASIFormatter interface extends the RASIObject interface. The implementation of the RASIObject interface is the same for both handlers and formatters. A short discussion of the methods that are introduced by the RASIFormatter interface follows. For more in-depth information on the methods introduced by this interface, see the corresponding product API documentation.

**RASIFormatter interface**

- The public void setDefault(boolean flag) and public boolean isDefault methods are used by the concrete RASHandler classes that are provided by the stand-alone JRas logging toolkit to determine if a particular formatter is the default formatter. Because these RASHandler classes must never be used in a WebSphere Application Server environment, the semantic significance of these methods can be determined by the user.
- The public void addEventClass(String name), public void removeEventClass(String name) and public Enumeration getEventClasses methods are provided to determine which event classes a formatter can use to format. You can provide the appropriate implementations.
- The public String format(RASIEvent event) method is called by handler objects and returns a formatted String representation of the event.

## Programming model summary

The programming model that is described in this section builds upon and summarizes some of the concepts already introduced. This section also formalizes usage requirements and restrictions. Use of the WebSphere Application Server JRas extensions in a manner that does not conform to the following programming guidelines is prohibited.

**Deprecated:** The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

You can use the WebSphere Application Server JRas extensions in three distinct operational modes. The programming models concepts and restrictions apply equally across all modes of operation.

- You must not use implementation classes that are provided by the stand-alone JRas logging toolkit directly, unless specifically noted otherwise. Direct usage of those classes is not supported. IBM Support provides no diagnostic aid or bug fixes relating to the direct use of classes that are provided by the stand-alone JRas logging toolkit.
- You must obtain message and trace loggers directly from the Manager class. You cannot directly instantiate loggers.
- You cannot replace the WebSphere Application Server message and trace logger classes.
- You must guarantee that the logger names that are passed to the Manager class are unique, and follow the documented naming constraints. When a logger is obtained from the Manager class, you must not attempt to change the name of the logger by calling the setName method.
- Named loggers can be used more than once. For any given name, the first call to the Manager class results in the Manager class creating a logger that is associated with that name. Subsequent calls to the Manager class that specify the same name result in a returned reference to the existing logger.
- The Manager class maintains a hierarchical namespace for loggers. Use a dot-separated, fully qualified class name to identify any logger. Other than dots or periods, logger names cannot contain any punctuation characters, such as an asterisk (*), a comma (.), an equals sign (=), a colon (:), or quotes.
- Group names must comply with the same naming restrictions as logger names.
- The loggers returned from the Manager class are subclasses of the RASMessageLogger and the RASTraceLogger classes that are provided by the stand-alone JRas logging toolkit. You can call any public method that is defined by the RASMessageLogger and RASTraceLogger classes. You cannot call any public method that is introduced by the provided subclasses.
- If you want to operate in either stand-alone or combined mode, you must provide your own Handler and Formatter subclasses. You cannot use the Handler and Formatter classes that are provided by the stand-alone JRas logging toolkit. User written handlers and formatters must conform to the documented guidelines.
- Loggers that are obtained from the Manager class come with a WebSphere Application Server handler installed. This handler writes message and trace records to logs that are defined by the WebSphere Application Server run time. Manage these logs using the provided systems management interfaces.
- You can programmatically add and remove user-defined handlers from a logger at any time. Multiple additions and removals of user defined handlers are supported. You are responsible for creating an instance of the handler to add, configuring the handler by setting the handler mask value and formatter appropriately, then adding the handler to the logger using the addHandler method. You are responsible for programmatically updating the masks of user-defined handlers, as appropriate.
- You might get a reference to the handler that is installed within a logger by calling the getHandlers method on the logger and processing the results. You must not call any methods on the handler that are obtained in this way. You can remove the WebSphere Application Server handler from the logger by calling the logger removeHandler method, passing in the reference to the WebSphere Application Server handler. When removed, the WebSphere Application Server handler cannot be added again to the logger.
- You can define your own message type. The behavior of user-defined message types and restrictions on their definitions is discussed in Extending the JRas framework.
- You can define your own message event classes. The use of user-defined message event classes is discussed in Extending the JRas framework.
- You can define your own trace types. The behavior of user-defined trace types and restrictions on your definitions is discussed in Extending the JRas framework.
- You can define your own trace event classes. The use of user-defined trace event classes is discussed in Extending the JRas framework.
- You must programmatically maintain the bits in the message and trace logger masks that correspond to any user-defined types. If WebSphere Application Server facilities are used to manage the predefined types, these updates must not modify the state of any of the bits that correspond to those types. If you are assuming ownership responsibility for the predefined types, then you can change all bits of the masks.

# JRas messages and trace event types

The basic JRas message and event types are not the same as those natively recognized by WebSphere Application Server, so the JRas types are mapped onto the types that are native to the runtime environment. You can control the way JRas message and trace events are processed using custom filters and message controls.

## Event types

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The base message and trace event types that are defined by the stand-alone JRas logging toolkit are not the same as the native types that are recognized by the WebSphere Application Server run-time. Instead, the basic JRas types are mapped onto the native types. This mapping can vary by platform or edition. The mapping is discussed in the following section.

## Platform message event types

The message event types that are recognized and processed by the WebSphere Application Server runtime are defined in the RASIMessageEvent interface that is provided by the stand-alone JRas logging toolkit. These message types are mapped onto the native message types, as follows.

| WebSphere Application Server native type | JRas RASIMessageEvent type |
|---|---|
| Audit | TYPE_INFO, TYPE_INFORMATION |
| Warning | TYPE_WARN, TYPE_WARNING |
| Error | TYPE_ERR, TYPE_ERROR |

## Platform trace event types

The trace event types that are recognized and processed by the WebSphere Application Server run time are defined in the RASITraceEvent interface that is provided by the stand-alone JRas logging toolkit. The RASITraceEvent interface provides a rich and complex set of types. This interface defines both a simple set of levels, as well as a set of enumerated types.

- For a user who prefers a simple set of levels, the RASITraceEvent interface provides `TYPE_LEVEL1`, `TYPE_LEVEL2`, and `TYPE_LEVEL3`. The implementations provide support for this set of levels. The levels are hierarchical, enabling level 2 also enables level 1, enabling level 3 also enables levels 1 and 2.
- For users who prefer a more complex set of values that can be *OR'd* together, the RASITraceEvent interface provides `TYPE_API`, `TYPE_CALLBACK`, `TYPE_ENTRY_EXIT`, `TYPE_ERROR_EXC`, `TYPE_MISC_DATA`, `TYPE_OBJ_CREATE`, `TYPE_OBJ_DELETE`, `TYPE_PRIVATE`, `TYPE_PUBLIC`, `TYPE_STATIC`, and `TYPE_SVC`.

The trace event types are mapped onto the native trace types as follows:

Mapping WebSphere Application Server trace types to the JRas RASITraceEvent level types.

| WebSphere Application Server native type | JRas RASITraceEvent level type |
|---|---|
| Event | TYPE_LEVEL1 |
| EntryExit | TYPE_LEVEL2 |
| Debug | TYPE_LEVEL3 |

Mapping WebSphere Application Server trace types to the JRas RASITraceEvent enumerated types.

| WebSphere Application Server native type | JRas RASITraceEvent enumerated types |
|---|---|

| | |
|---|---|
| Event | TYPE_ERROR_EXC, TYPE_SVC, TYPE_OBJ_CREATE, TYPE_OBJ_DELETE |
| EntryExit | TYPE_ENTRY_EXIT, TYPE_API, TYPE_CALLBACK, TYPE_PRIVATE, TYPE_PUBLIC, TYPE_STATIC |
| Debug | TYPE_MISC_DATA |

For simplicity, it is recommended that one or the other of the tracing type methodologies is used consistently throughout the application. If you decide to use the non-level types, choose one type from each category and use those types consistently throughout the application, to avoid confusion.

## Message and trace parameters

The various message logging and trace method signatures accept the `Object`, `Object[]` and `Throwable` parameter types. WebSphere Application Server processes and formats the various parameter types as follows:

**Primitives**

Primitives, such as int and long are not recognized as subclasses of Object type and cannot be directly passed to one of these methods. A primitive value must be transformed to a proper Object type (Integer, Long) before passing as a parameter.

**Object**

The toString method is called on the object and the resulting String is displayed. Implement the toString method appropriately for any object that is passed to a message logging or trace method. It is the responsibility of the caller to guarantee that the toString method does not display confidential data such as passwords in clear text, and does not cause infinite recursion.

**Object[]**

The Object[] type is provided for the case when more than one parameter is passed to a message logging or trace method. The toString method is called on each Object in the array. Nested arrays are not handled, that is none of the elements in the Object array belong in an array.

**Throwable**

The stack trace of the Throwable type is retrieved and displayed.

**Array of primitives**

An array of primitive, for example, `byte[]`, `int[]`, is recognized as an Object, but is treated somewhat as a second cousin of Object by Java code. In general, avoid arrays of primitives, if possible. If arrays of primitives are passed, the results are indeterminate and can change, depending on the type of array passed, the API used to pass the array, and the release of the product. For consistent results, user code needs to preprocess and format the primitive array into some type of String form before passing it to the method. If such preprocessing is not performed, the following problems can result:

- [B@924586a0b - This message is deciphered as a byte array at location X. This message is typically returned when an array is passed as a member of an Object[] type and results from calling the toString method on the byte[] type.
- Illegal trace argument : array of long. This response is typically returned when an array of primitives is passed to a method taking an Object.
- 01040703: The hex representation of an array of bytes. Typically this problem can occur when a byte array is passed to a method taking a single Object. This behavior is subject to change and cannot be relied on.
- ″1″ ″2″: The String representation of the members of an int[] type formed by converting each element to an integer and calling the toString method on the integers. This behavior is subject to change and cannot be relied on.
- [Ljava.lang.Object;@9136fa0b : An array of objects. Typically this response is seen when an array containing nested arrays is passed.

## Controlling message logging

Writing a message to a WebSphere Application Server log requires that the message type passes three levels of filtering or screening:
1. The message event type must be one of the message event types that is defined in the RASIMessageEvent interface.
2. Logging of that message event type must be enabled by the state of the message logger mask.
3. The message event type must pass any filtering criteria that is established by the WebSphere Application Server run-time.

When a WebSphere Application Server logger is obtained from the Manager class, the initial setting of the mask forwards all native message event types to the WebSphere Application Server handler. It is possible to control what messages get logged by programmatically setting the state of the message logger mask.

Some editions of the product support user specified message filter levels for a server process. When such a filter level is set, only messages at the specified severity levels are written to WebSphere Application Server. Message types that pass the mask check of the message logger can be filtered out by WebSphere Application Server.

## Control tracing

Each edition of the product provides a mechanism for enabling or disabling trace. The various editions can support static trace enablement (trace settings are specified before the server is started), dynamic trace enablement (trace settings for a running server process can be dynamically modified), or both.

Writing a trace record to a WebSphere Application Server requires that the trace type passes three levels of filtering or screening:
1. The trace event type must be one of the trace event types that is defined in the RASITraceEvent interface.
2. Logging of that trace event type must be enabled by the state of the trace logger mask.
3. The trace event type must pass any filtering criteria that is established by the WebSphere Application Server run-time.

When a logger is obtained from the Manager class, the initial setting of the mask is to suppress all trace types. The exception to this rule is the case where the WebSphere Application Server run time supports static trace enablement and a non-default startup trace state for that trace logger is specified. Unlike message loggers, the WebSphere Application Server can dynamically modify the trace mask state of a trace logger. WebSphere Application Server only modifies the portion of the trace logger mask that corresponds to the values that are defined in the RASITraceEvent interface. WebSphere Application Server does not modify undefined bits of the mask that might be in use for user-defined types.

When the dynamic trace enablement feature that is available on some platforms is used, the trace state change is reflected both in the application server run time and the trace mask of the trace logger. If user code programmatically changes the bits in the trace mask corresponding to the values that are defined by in the RASITraceEvent interface, the mask state of the trace logger and the run time state become unsynchronized and unexpected results occur. Therefore, programmatically changing the bits of the mask corresponding to the values that are defined in the RASITraceEvent interface is not supported.

**Related tasks**

“Programming with the JRas framework” on page 31
Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server applications.

# Instrumenting an application with JRas extensions

You can create an application using JRas extensions.

## Before you begin

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

## About this task

To create an application using the WebSphere Application Server JRas extensions, perform the following steps:

1. Determine the mode for the extensions: integrated, stand-alone, or combined.
2. If the extensions are used in either stand-alone or combined mode, create the necessary handler and formatter classes.
3. If localized messages are used by the application, create a resource bundle.
4. In the application code, get a reference to the Manager class and create the manager and logger instances.
5. Insert the appropriate message and trace logging statements in the application.

## Creating JRas resource bundles and message files

The WebSphere Application Server message logger provides the message and msg methods so the user can log localized messages. In addition, the message logger provides the textMessage method to log messages that are not localized. Applications can use either or both, as appropriate.

### Before you begin

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

The mechanism for providing localized messages is the resource bundle support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.ResourceBundle, java.util.ListResourceBundle and java.util.PropertyResourceBundle classes, as well as the java.text.MessageFormat class.

The PropertyResourceBundle class is the preferred mechanism to use. In addition, note that the JRas extensions do not support the extended formatting options such as {1, date} or {0, number, integer} that are provided by the MessageFormat class.

You can forward messages that are written to the internal WebSphere Application Server logs to other processes for display. For example, messages that are displayed on the administrative console, which can be running in a different location than the server process, can be localized using the *late binding* process. Late binding means that WebSphere Application Server does not localize messages when they are logged, but defers localization to the process that displays the message.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. You must package the resource bundle separately from the application, and install it in a location where the viewing process can access it. If you do not want to take these steps, you can use the early binding technique to localize messages as they are logged.

The two techniques are described as follows:

**Early binding**
: The application must localize the message before logging it. The application looks up the localized

text in the resource bundle and formats the message. When formatting is complete, the application logs the message using the textMessage method. Use this technique to package the application resource bundles with the application.

**Late binding**

The application can choose to have the WebSphere Application Server run time localize the message in the process where it displays. Using this technique, the resource bundles are packaged in a stand-alone `.jar` file, separately from the application. You must then install the resource bundle `.jar` file on every machine in the installation from which an administrative console or log viewing program might be run. You must install the `.jar` file in a directory that is part of the extensions class path. In addition, if you forward logs to IBM service, you must also forward the `.jar` file that contains the resource bundles.

To create a resource bundle, perform the following steps.

1. Create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:
   - Each property in the file is terminated with a line-termination character.
   - If a line contains only white space, or if the first non-white space character of the line is the number sign symbol (`#`) or exclamation mark (`!`), the line is ignored. The `#` and `!` characters can therefore be used to put comments into the file.
   - Each line in the file, unless it is a comment or consists only of white space, denotes a single property. A backslash (\) is treated as the line-continuation character.
   - The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (`=`), colon (`:`), and white space ( ).
   - The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but using this approach is not recommended because escaping characters is error prone and confusing. Instead, use a valid separator character that does not display in any keys in the properties file.
   - White space after the key and separator is ignored until the first non-white space character is encountered. All characters that remain before the line-termination character define the element.

   See the Java documentation for the java.util.Properties class for a full description of the syntax and construction of properties files.
2. Translate the file into localized versions of the file with language-specific file names for example, the DefaultMessages.properties file can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese.
3. When the translated resource bundles are available, write them to a system-managed persistent storage medium. Resource bundles are used to convert the messages into the requested national language and locale.
4. When a message logger is obtained from the JRas manager, configure the logger to use a particular resource bundle. Messages logged through the `message` API use this resource bundle when message localization is performed. At run time, the user's locale setting is used to determine the properties file from which to extract the message that is specified by a message key, ensuring that the message is delivered in the correct language.
5. If the message loggers msg method is called, explicitly identify a resource bundle name.

## What to do next

The application locates the resource bundle based on the file location relative to any directory in the class path. For instance, if the DefaultMessages.properties property resource bundle is in the `baseDir/subDir1/subDir2/resources` directory and `baseDir` is in the class path, the name *subdir1.subdir2*.resources.DefaultMessage is passed to the message logger to identify the resource bundle.

***JRas resource bundles:***

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a PropertiesResourceBundle resource bundle. This sample shows how to create such a properties file.

**Resource bundle sample**

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs are inserted into it. All the normal properties files conventions and rules apply to this file. In addition, the creator must be aware of other restrictions that are imposed on the values by the Java MessageFormat class. For example, apostrophes must be escaped or they cause a problem. Avoid the use of non-portable characters. WebSphere Application Server does not support the use of extended formatting conventions that the MessageFormat class supports, such as {1, date} or {0, number, integer}.

Assume that the base directory for the application that uses this resource bundle is `baseDir` and that this directory is in the class path. Assume that the properties file is stored in the subdirectory `baseDir` that is not in the class path (*baseDir/subDir1/subDir2/resources*). To allow the messages file to resolve, the subDir1.subDir2.resources.DefaultMessage name is used to identify the PropertyResourceBundle resource bundle and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`:

```
# Contents of the DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three substitution parameters: parm1={0}, parm2 = {1}, parm3={2}
```

When the `DefaultMessages.properties` file is created, the file can be sent to a translation center where the localized versions are generated.

## JRas manager and logger instances

You can use the JRas extensions in integrated, stand-alone, or combined mode. Configuration of the application varies depending on the mode of operation, but use of the loggers to log message or trace entries is identical in all modes of operation.

**Deprecated:** The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Integrated mode is the default mode of operation. In this mode, message and trace events are sent to the WebSphere Application Server logs.

In the combined mode, message and trace events are logged to both WebSphere Application Server and user-defined logs.

In the stand-alone mode, message and trace events are logged only to user-defined logs.

### Using the message and trace loggers

Regardless of the mode of operation, the use of message and trace loggers is the same.

### Using a message logger

The message logger is configured to use the DefaultMessages resource bundle. Message keys must be passed to the message loggers if the loggers are using the message API.

```
msgLogger.message(RASIMessageEvent.TYPE_WARNING, this,
     methodName, "MSG_KEY_00");
... msgLogger.message(RASIMessageEvent.TYPE_WARN, this,
     methodName, "MSG_KEY_01", "some string");
```

If message loggers use the msg API, you can specify a new resource bundle name.

```
msgLogger.msg(RASIMessageEvent.TYPE_ERR, this, methodName,
                  "ALT_MSG_KEY_00", "alternateMessageFile");
```

You can also log a text message. If you are using the textMessage API, no message formatting is done.

```
msgLogger.textMessage(RASIMessageEvent.TYPE_INFO, this, methodName,"String and Integer",
"A String", new Integer(5));
```

### Using a trace logger

Because trace is normally disabled, guard trace methods for performance reasons.

```
private void methodX(int x, String y, Foo z)
{
   // trace an entry point. Use the guard to make sure tracing is enabled.
Do this checking before you gather parameters to  trace.
   if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT) {
        // I want to trace three parameters, package them up in an Object[]
        Object[] parms = {new Integer(x), y, z};
        trcLogger.entry(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX", parms);
  }
... logic
  // a debug or verbose trace point
  if (trcLogger.isLoggable(RASITraceEvent.TYPE_MISC_DATA) {
        trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA, this, "methodX" "reached here");
  }
  ...
  // Another classification of trace event. An important state change is
 detected, so a different trace type is used.
  if (trcLogger.isLoggable(RASITraceEvent.TYPE_SVC) {
     trcLogger.trace(RASITraceEvent.TYPE_SVC, this, "methodX", "an important event");
  }
 ...
  // ready to exit method, trace. No return value to trace
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
        trcLogger.exit(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX");
   }
}
```

## Setting up for integrated JRas operation

Use JRas operations in integrated mode to send trace events and logging messages to only WebSphere Application Server logs.

### Before you begin

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

In the integrated mode of operation, message and trace events are sent to WebSphere Application Server logs. This approach is the default mode of operation.

1. Import the requisite JRas extensions classes:

   ```
   import com.ibm.ras.*;
   import com.ibm.websphere.ras.*;
   ```

2. Declare logger references:

```
private RASMessageLogger msgLogger = null;
private RASTraceLogger trcLogger = null;
```

3. Obtain a reference to the Manager class and create the loggers. Because loggers are named singletons, you can do this activity in a variety of places. One logical candidate for enterprise beans is the ejbCreate method. For example, for the myTestBean enterprise bean, place the following code in the ejbCreate method:

```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
         myTestBean.class.getName());



// Configure the message logger to use the message file that is created
// for this application.
msgLogger.setMessageFile("acme.widgets.DefaultMessages");
trcLogger = mgr.createRASTraceLogger("Acme", "Widgets", "RasTest",
         myTestBean.class.getName());
mgr.addLoggerToGroup(trcLogger, groupName);
```

## Setting up for combined JRas operation

Use JRas operation in combined mode to output trace data and logging messages to both WebSphere Application Server and user-defined logs.

### Before you begin

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

In combined mode, messages and trace are logged to both WebSphere Application Server logs and user-defined logs. The following sample assumes that:

- You wrote a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter.
- You are not using user-defined types or events.

1. Import the requisite JRas extensions classes:

```
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;
```

2. Import the user handler and formatter:

```
import com.ibm.ws.ras.test.user.*;
```

3. Declare the logger references:

```
private RASMessageLogger msgLogger = null;
  private RASTraceLogger trcLogger = null;
```

4. Obtain a reference to the Manager class, create the loggers, and add the user handlers. Because loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for enterprise beans is the ejbCreate method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must support this approach. The following sample is a message logger sample. The procedure for a trace logger is similar.

```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
    msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
           myTestBean.class.getName());
    // Configure the message logger to use the message file defined
    // in the ResourceBundle sample.
    msgLogger.setMessageFile("acme.widgets.DefaultMessages");

   // Create the user handler and formatter. Configure the formatter,
```

```
// then add it to the handler.
RASIHandler handler = new SimpleFileHandler("myHandler", "FileName");
RASIFormatter formatter = new SimpleFormatter("simple formatter");
 formatter.addEventClass("com.ibm.ras.RASMessageEvent");
 handler.addFormatter(formatter);

// Add the Handler to the logger. Add the logger to the list of the
//handlers listeners, then set the handlers
// mask, which updates the loggers composite mask appropriately.
// WARNING - there is an order dependency here that must be followed.
msgLogger.addHandler(handler);
handler.addMaskChangeListener(msgLogger);
handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

## Setting up for stand-alone JRas operation

You can configure JRas operations to output trace data and logging messages to only user-defined locations.

### Before you begin

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

In stand-alone mode, messages and traces are logged only to user-defined logs. The following sample assumes that:

- You have a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter.
- You are not using user-defined types of events.

1. Import the requisite JRas extensions classes:

   ```
   import com.ibm.ras.*;
   import com.ibm.websphere.ras.*;
   ```

2. Import the user handler and formatter:

   ```
   import com.ibm.ws.ras.test.user.*;
   ```

3. Declare the logger references:

   ```
   private RASMessageLogger msgLogger = null;
     private RASTraceLogger trcLogger = null;
   ```

4. Obtain a reference to the Manager class, create the loggers, and add the user handlers. Because loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for enterprise beans is the ejbCreate method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must support this approach. The following sample is a message logger sample. The procedure for a trace logger is similar.

   ```
   com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
       msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
                myTestBean.class.getName());
       // Configure the message logger to use the message file that is defined in
       //the ResourceBundle sample.
       msgLogger.setMessageFile("acme.widgets.DefaultMessages");

       // Get a reference to the Handler and remove it from the logger.
       RASIHandler aHandler = null;
       Enumeration enum = msgLogger.getHandlers();
       while (enum.hasMoreElements()) {
            aHandler = (RASIHandler)enum.nextElement();
            if (aHandler instanceof WsHandler)
                 msgLogger.removeHandler(wsHandler);
       }
   ```

```
 // Create the user handler and formatter. Configure the formatter,
 // then add it to the handler.
 RASIHandler handler = new SimpleFileHandler("myHandler", "FileName");
 RASIFormatter formatter = new SimpleFormatter("simple formatter");
 formatter.addEventClass("com.ibm.ras.RASMessageEvent");
 handler.addFormatter(formatter);

// Add the Handler to the logger. Add the logger to the list of the
// handlers listeners, then set the handlers
// mask, which will update the loggers composite mask appropriately.
// WARNING - there is an order dependency here that must be followed.
msgLogger.addHandler(handler);
handler.addMaskChangeListener(msgLogger);
handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

# Logging Common Base Events in WebSphere Application Server

WebSphere Application Server uses Common Base Events within its logging framework. Common Base Events can be created explicitly and then logged through the Java logging API, or can be created implicitly by using the Java logging API directly.

## About this task

An *event* is a notification from an application or the application server that reports information that is related to a specific problem or situation. Common Base Events provide you with a standard structure for these event notifications, which allow you to correlate events that are received from different applications. Log Common Base Events to capture events from different sources to help you fix a problem within an application environment or to tune system performance.

For Common Base Event creation, the application server environment provides a Common Base Event factory with a content handler that provides both runtime data and template data for Common Base Events.

1. Optional: Read about the Common Base Event types and how they are implemented within an application server. Refer to “The Common Base Event in WebSphere Application Server.”
2. Read “Logging Common Base Events in WebSphere Application Server” on page 75.
3. Configure the Common Base Event framework for your application server using one of the following methods:
   - “Logging with Common Base Event API and the Java logging API” on page 64
   - “Generate Common Base Event content with the default event factory” on page 65.

## Results

Common Base Events will now be logged according to your configuration. Use these event logs to determine the source of application problems.

# The Common Base Event in WebSphere Application Server

The Common Base Event is an XML document that defines a common representation of events that is intended for use by enterprise management and business applications. The Common Base Event defines common fields, the values they can take, and the exact meanings of these values.

An application creates an event object whenever something happens that either needs to be recorded for later analysis or which might require the trigger of additional work. An *event* is a structured notification that reports information that is related to a situation. An event reports three kinds of information:

- The situation: What happened
- The identity of the affected component: For example, the server that shut down

- The identity of the component that is reporting the situation, which might be the same as the affected component

The application that creates the event object is called the *event source*. Event sources can use a common structure for the event. The accepted standard for such a structure is called the *Common Base Event*. The Common Base Event is an XML document that is defined as part of the autonomic computing initiative.

The Common Base Event model is a standard that defines a common representation of events that is intended for use by enterprise management and business applications. This standard, which is developed by the IBM Autonomic Computing Architecture Board, supports encoding of logging, tracing, management, and business events using a common XML-based format. This format makes it possible to correlate different types of events that originate from different applications. For more information about the Common Base Event model, see the Common Base Event specification (*Canonical Situation Data Format: The Common Base Event V1.0.1*). The common event infrastructure currently supports Version 1.0.1 of the specification.

The basic concept behind the Common Base Event model is the *situation*. A situation can be anything that happens anywhere in the computing infrastructure, such as a server shutdown, a disk-drive failure, or a failed user login. The Common Base Event model defines a set of standard situation types that accommodate most of the situations that might arise (for example, StartSituation and CreateSituation).

The Common Base Event contains all of the information that is needed by the consumers to understand the event. This information includes data about the runtime environment, the business environment, and the instance of the application object that created the event.

For complete details on the Common Base Event format, see the XML schema that is included in the Common Base Event specification document, at http://www.ibm.com/developerworks/autonomic/books/fpy0mst.htm#HDRCBEDESC .

## Types of problem determination events

Problem determination involves multiple types of data, including at least two different classes of event data, log events, and diagnostic events.

Log events, which are also referred to as *message events*, are typically emitted by components of a business application during normal deployment and operations. Log events might identify problems, but these events are also normally available and emitted while an application and its components are in production mode. The target audience for log and message events is users and administrators of the application and the components that make up the application. Log events are normally the only events available when a problem is first detected, and are typically used during both problem recovery and problem resolution.

Diagnostic events, which are commonly referred to as *trace events*, are used to capture internal diagnostic information about a component, and are usually not emitted or available during normal deployment and operation. The target audience for diagnostic events is the developers of the components that make up the business application. Diagnostic events are typically used when trying to resolve problems within a component, such as a software failure, but are sometimes used to diagnose other problems, especially when the information provided by the log events is not sufficient to resolve the problem. Diagnostic events are typically used when trying to resolve a problem.

A *Common Base Event* is a common structure for an event. It defines common fields, the values that these fields can take, and the exact meanings of these values for an event. Common Base Events are primarily used to represent log events.

## Common Base Event structure

A *Common Base Event* is a common structure for an event. It defines common fields, the values that these fields can take, and the exact meanings of these values for an event.

The Common Base Event contains several structural elements. These elements include:

- Common header information
- Component identification, both source and reporter
- Situation information
- Message data
- Extended data
- Context data
- Associated events and association engine

Each of these structural elements has its own embedded elements and attributes.

The following table presents a summary of all the fields in the Common Base Event and their usage requirements for problem determination events. This table shows whether a particular element or attribute is required, recommended, optional, prohibited, or discouraged for log events, and the base specification.

| Field name | Log events | Base specification |
|---|---|---|
| Version | Required | Required |
| creationTime | Required | Required |
| severity | Required | Optional |
| Msg | Required | Optional |
| sourceComponentId* | Required | Required |
| sourceComponentId.location | Required | Required |
| sourceComponentId.locationType | Required | Required |
| sourceComponentId.component | Required | Required |
| sourceComponentId.subComponent | Required | Required |
| sourceComponentId.componentIdType | Required | Required |
| sourceComponentId.componentType | Required | Required |
| sourceComponentId.application | Recommended | Optional |
| sourceComponentId.instanceId | Recommended | Optional |
| sourceComponentId.processId | Recommended | Optional |
| sourceComponentId.threadId | Recommended | Optional |
| sourceComponentId.executionEnvironment | Optional | Optional |
| situation* | Required | Required |
| situation.categoryName | Required | Required |
| situation.situationType* | Required | Required |
| situation.situationType.reasoningScope | Required | Required |
| situation.situationType.(specific Situation Type elements) | Required | Required |
| msgDataElement* | Recommended | Optional |
| msgDataElement .msgId | Recommended | Optional |
| msgDataElement .msgIdType | Recommended | Optional |
| msgDataElement .msgCatalogId | Recommended | Optional |
| msgDataElement .msgCatalogTokens | Recommended | Optional |
| msgDataElement .msgCatalog | Recommended | Optional |
| msgDataElement .msgCatalogType | Recommended | Optional |
| msgDataElement .msgLocale | Recommended | Optional |

| | | |
|---|---|---|
| extensionName | Recommended | Optional |
| localInstanceId | Optional | Optional |
| globalInstanceId | Optional | Optional |
| priority | Discouraged | Optional |
| repeatCount | Optional | Optional |
| elapsedTime | Optional | Optional |
| sequenceNumber | Optional | Optional |
| reporterComponentId* | Optional | Optional |
| reporterComponentId.location | Required (2) | Required (2) |
| reporterComponentId.locationType | Required (2) | Required (2) |
| reporterComponentId.component | Required (2) | Required (2) |
| reporterComponentId.subComponent | Required (2) | Required (2) |
| reporterComponentId.componentIdType | Required (2) | Required (2) |
| reporterComponentId.componentType | Required (2) | Required (2) |
| reporterComponentId.instanceId | Optional | Optional |
| reporterComponentId.processId | Optional | Optional |
| reporterComponentId.threadId | Optional | Optional |
| reporterComponentId.application | Optional | Optional |
| reporterComponentId.executionEnvironment | Optional | Optional |
| extendedDataElements* | Note 3 | Optional |
| contextDataElements* | Note 4 | Optional |
| associatedEvents* | Note 5 | Optional |

**Notes:**

- Items followed by an asterisk (*) are elements that consist of sub elements and attributes. The fields in those elements are listed in the table directly following the parent element name.
- Some of the elements are optional, but when included, they include sub elements and attributes that are required. For example, the reporterComponentId element has a ComponentIdentification type. The component attribute in ComponentIdentification is required. Therefore, the reporterComponentId.component attribute is required, but only when the reporterComponentId parent element is included.
- The extendedDataElements element can be included multiple times to supply extended data information. See the Extended data section for more information on required and recommended extended data element values.
- The contextDataElements element can be included multiple times to supply context data information.
- The associatedEvents element can be included multiple times to supply correlation data. No recommended uses of this element exist for the producers of problem determination data, and the use of this element is discouraged.

***Common header information:***

This topic provides additional information about how to format and use these fields for problem determination events, which can be used to clarify and extend the information provided in the other documents.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

The common header information in the Common Base Event includes the following information about an event:

- Version: The version of this Common Base Event
- creationTime: The date and time when the event generated
- Severity and priority: The severity of the condition (situation) that is identified by the event
- extensionName: The type of event that was captured
- localInstanceId and globalInstanceId: Identifiers that can be used to quickly identify a specific event within a set of events
- repeatCount and elapsedTime: Information that supports a system to efficiently report multiple events of the same type, by consolidating those events into a single event
- sequenceNumber: Sequence information that supports a system to order a set of events in other ways than time of capture

**severity**

All problem determination events must provide an indication as to the relative severity of the condition (situation) being reported by providing appropriate values for the severity field in the Common Base Event. The severity field is required for problem determination events. This field is more restrictive than the base specification for the Common Base Event, which lists this field as optional because effective and efficient problem determination requires the ability to quickly identify the information that is needed to resolve a problem as well as prioritize the problems that need addressing. Typically, the following values are used for problem determination events:

| | | |
|---|---|---|
| 10 | Information | Log information events, normal conditions, and events that are supplied to clarify operations, for example, state transitions, operational changes. These events typically do not require administrator action or intervention. |
| 20 | Harmless | Similar to information events, but are used to capture audit items, such as state transitions or operational changes. These events typically do not require administrator action or intervention. |
| 30 | Warning | Warnings typically represent recoverable errors, for example a failure that the system can correct. These events can require administrator action or intervention. |
| 40 | Minor | Minor errors describe events that represent an unrecoverable error within a component. The failure affects the component ability to service some requests. The business application can continue to perform its normal functions, but its overall operation might be degraded. These events require administrator action or intervention to address the condition. |

| | | |
|---|---|---|
| 50 | Critical | Critical errors describe events that represent an unrecoverable error within a component. The failure significantly affects the component ability to service most requests. The business application can continue most, but not all of its normal functions and its overall operation might be degraded. These events require administrator action or intervention to address the condition. |
| 60 | Fatal | Fatal errors describe events that represent an unrecoverable error within a component. The failure usually results in the complete failure of the component. The business application can continue some normal functions, but its overall operation might be degraded. These events require administrator action or intervention to address the condition. |

**msg**
Refer to “Message data” on page 60 for information on this attribute.

**priority**
The use of the priority field is discouraged for problem determination events. The severity field is typically used to communicate and evaluate the importance of problem determination events. Use the priority field to enhance the information that is provided in the severity field, that is. prioritize events of the same severity.

**extensionName**
The extensionName field is used to communicate the type of event that is reported, for example, what general class of events is being reported. In many cases this field provides an indication of what additional data you can expect with the event, for example, optional data values.

**repeatCount**
The repeatCount field is valid for problem determination events, but is not typically used or supplied by the event producers. This field is used for data reduction and consolidation by event management and analysis systems.

**elapsedTime**
The elapsedTime field is valid for problem determination events, but is not typically used or supplied by the event producers. This field is used for data reduction and consolidation by event management and analysis systems.

**sequenceNumber**
The sequenceNumber field is valid for problem determination events. It is typically used only by event producers when the granularity of the event time stamp (the creationTime field) is not sufficient in ordering events. The sequenceNumber field is typically used to sequence events that have the same time stamp value.

Event management and analysis systems can use the sequenceNumber field for a number of reasons, including providing alternative sequencing, not necessarily based on a time stamp. The recommendations here are provided primarily for event producers.

***Component identification for source and reporter:***

The component identification fields in the Common Base Event are used to indicate which component in the system is experiencing the condition that is described by the event (the sourceComponentID) and which component emitted the event (the reporterComponentID).

Typically, these components are the same, in which case only the sourceComponentID is supplied. Some notes and scenarios on when to use these two elements in the Common Base Event:

- The sourceComponentID is always used to identify the component experiencing the condition that is described by the event.
- The reporterComponentID is used to identify the component that actually produced and emitted the event. This element is typically used only within events that are emitted by a component that is monitoring another component and providing operational information regarding that component. The monitoring component (for example, a Tivoli® agent or hardware device driver) is identified by the reporterComponentID and the component being monitored (for example, a monitored server or hardware device) is identified by the sourceComponentID.

  A potential misuse of the reporterComponentID is to identify a component that provides event conversion or management services for a component, for example, identifying an adapter that transforms the events that are captured by a component into Common Base Event format. The event conversion function is considered an extension of the component and not identified separately.

The information that is used to identify a component in the system is the same, regardless of whether it is the source component or reporter component:

| | | |
|---|---|---|
| location locationType | Component location | Identifies the location of the component. |
| component componentIdType | Component name | Identifies the asset name of the component, as well as the type of component. |
| subcomponent | Subcomponent name | Identifies a specific part or subcomponent of a component, for example a software module or hardware part. |
| application | Business application name | Identifies the business application or process the component is a part of and provides services for. |
| instanceId | Operational instance | Identifies the operational instance of a component, that is the actual running instance of the component. |
| processId threadId | Operational instance | Identifies the operational instance of a component within the context of a software operating system, that is he operating system process and thread running when the event was produced. |
| executionEnvironment | Operational instance Component location | Provides additional information about the operational instance of a component or its location by identifying the name of the environment hosting the operational instance of the component, for example the operating system name for a software application, the application server name for a Java 2 Platform, Enterprise Edition (J2EE) application, or the hardware server type for a hardware part. |

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines. This section provides additional information about how to format and use some of these fields for problem determination events, which can be used to clarify and extend the information that is provided in the other documents.

**Component**

The Component field in a problem determination event is used to identify the manageable asset that is associated with the event. A manageable asset is open for interpretation, but a good working definition is a manageable asset represents a hardware or software component that can be separately obtained or developed, deployed, managed, and serviced. Examples of typical component names are:

- IBM eServer™ xSeries® model x330
- IBM WebSphere Application Server version 5.1 (5.1 is the version number)
- Microsoft® Windows® 2000
- The name of an internally developed software application for a component

**subComponent**

The Subcomponent field in a problem determination event identifies the specific part of a component that is associated with the event. The subcomponent name is typically not a manageable asset, but provides internal diagnostic information when diagnosing an internal defect within a component, that is What part failed? Examples of typical subcomponents and their names are:

- Intel® Pentium® processor within a server system (Intel Pentium IV Processor)
- the enterprise bean container within a Web application server (enterprise bean container)
- the task manager within an operating system (Linux® Kernel Task Manager)
- the name of a Java class and method (myclass.mycompany.com or myclass.mycompany.com.methodname).

The format of a subcomponent name is determined by the component, but use the convention shown previously for naming a Java class or the combination of a Java class and method is followed. The subcomponent field is required in the Common Base Event.

**componentIdType**

The componentIdType field is required by the Common Base Event specification, but provides minimal value for problem determination events. For most problem determination events, it is encouraged to use the value provided in the application field instead of the componentIdType. The componentIdType field identifies the type of component; the application is identified by the application field.

**application**

The application field is listed as an optional value within the Common Base Event specification, but provide it within problem determination events whenever it this value is available. The only reason this field is not required for problem determination events is that instances exist where the issuing component might not be aware of the overall business application.

**instanceId**

The instanceId field is listed as an optional value within the Common Base Event specification, but provide this value within problem determination events whenever it is available.

Always provide the instanceID when a software component is identified and identify the operational instance of the component (for example, which operation instance of an installed software image is actually associated with the event). Provide this value for hardware components when these components support the concept of operational instances.

The format of the supplied value is defined by the component, but must be a value that an analysis system can use (either human or programmatic) to identify the specific running instance of the identified component. Examples include:

- **cell, node, server** name for the IBM WebSphere Application Server
- **deployed EAR file name** for a Java enterprise bean

- **serial number** for a hardware processor

**processId**
The processId field is listed as an optional value within the Common Base Event specification, but provide this value for problem determination events whenever it is available and applicable. Always provide this value for software-generated events, and identify the operating system process that is associated with the component that is identified in the event. Match the format of the thread ID with the format of the operating system (or other running environment, such as a Java virtual machine). This field is typically not applicable or used for events that are emitted by hardware (for example, firmware).

**threadId**
The threadId field is listed as an optional value within the Common Base Event specification, but provide this value for problem determination events whenever it is available and applicable. Always provide for software-generated events, and identify the active operating system thread when the event was detected or issued. A notable exception to this recommendation is some operating systems or running environments do not support threads. Match the format of the thread ID with the format of the operating system (or other running environment, such as a Java virtual machine). This field is typically not applicable or used for events that are emitted by hardware (for example, firmware).

**executionEnvironment**

The executionEnvironment field, when used, identifies the immediate running environment that is used by the component being identified. Some examples are:

- the operating system name when the component is a native software application.
- the operating system/Java virtual machine name when the component is a Java 2 Platform, Standard Edition (J2SE) application.
- the Web server name when the component is a servlet.
- the portal server name when the component is a portlet.
- the application server name when the component is an enterprise bean.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

***Situation information:***

The situation information is used to classify the condition that is reported by an event into a common set of situations.

The Common Base Event specification [CBE101] provides information on the set of situations defined for the Common Base Event, with the values and formats that are used to describe these situations. The Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

Consider the following points regarding situation information for problem determination events:

- Whenever possible, use the situation categorizations and qualifiers that are described in the base Common Base Event specification. Avoid using your own situation definitions as much as possible.
- Not all messages and logs can be classified using the situation definitions that are supplied in the base Common Base Event specification. You can use the OtherSituation categorization to provide your own situation information, but the recommended course of action for problem determination events is to use the ReportSituation categorization, with reportCategory=Log.
- Warning events can be confusing. A warning event (that is an event with severity=warning) typically indicates a recoverable failure, but the situation settings can be interpreted as unrecoverable failures (for example ConnectSituation, successDisposition=UNSUCCESSFUL). Use the appropriate situation categorization so the severity setting indicates the severity of the situation, that is whether the component recovered from the failure.
- The recommended setting for the reasoningScope value is EXTERNAL for all message events.

***Message data:***

All problem determination Common Base Events must provide human readable text that describes the specific reported event within the msg field of the Common Base Event.

The text that is associated with events representing actual messages or log entries is expected to be translated and localized. Include the msgDataElement element in the Common Base Event whenever internationalized text is provided in the event. This element provides information about how the message text is created and how to interpret it. This information is particularly invaluable when trying to interpret the event programmatically or when trying to interpret the message independent of the locale or language that is used to format the message text.

**Prerequisite:** Understand the concepts that are associated with creating internationalized messages. A good source of education on these concepts is provided by the documentation that is associated with internationalization of Java information and the usage of resource bundles within the Java language.

The msgDataElement element in the Common Base Event includes the following information about the value of the msg field that is provided with an event:

- The locale of the supplied message text, which identifies how the locale-independent fields within the message are formatted, as well as the language of the message (msgLocale).
- A locale-independent identifier that is associated with the message that can be used to interpret the message independent of the message language, message locale, and message format (msgId and msgIdType).
- Information on how a translated message is created, including:
  - The identifier that is used to retrieve the message template (msgCatalogId).
  - The name and type of message catalog that are used to retrieve the message template (msgCatalog and msgCatalogType).
  - Any locale-independent information that is inserted into the message template to create the final message (msgCatalogTokens).

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines. This section provides additional information about how to format and use these fields for problem determination events.

**msg**

All message, log, and trace events must provide a human-readable message in the msg field of the Common Base Event. The msg field is required for problem determination events, both log events and diagnostic events. This field is more restrictive than the base specification for the Common Base Event, which lists this field as optional; effective and efficient problem determination requires the ability to quickly identify the reported condition. The format and usage of this message is component-specific, but use the following general guidelines:

- Expect the message text that is supplied with messages and log events to be internationalized.
- Provide the locale of the supplied message text with the msgLocale field in the msgDataElement element of the Common Base Event.
- Provide additional information regarding the format and construction of internationalized messages whenever possible, using the msgDataElement element of the Common Base Event.

**msgLocale**

Provide the message locale whenever message text is provided within the Common Base Event, as is the case with all problem determination events. The msgLocale field is listed as an optional value within the Common Base Event specification, but provide this information within problem determination events whenever possible. The reason this field is not required for problem determination events is that instances exist where the locale information is not provided or available when formatting the Common Base Event.

**msgId and msgIdType**

Several companies include a locale-independent identifier within internationalized message text that you can use to interpret the described condition by the message text, independent of the message. For example, most messages issued by IBM software look like `IEE890I WTO Buffers in console backup storage = 1024`, where a unique, locale-independent identifier `IEE890I` precedes the translated message text. This identifier provides a way to uniquely detect and identify a message independent of location and language. This detection is invaluable for locale-independent and programmatic analysis.

The msgId field is listed as an optional value within the Common Base Event specification, but it must be provided within problem determination events whenever this identifier is included in the message text. Likewise, the msgIdType field is listed as an optional value within the Common Base Event specification, but it must be provided within problem determination events whenever a value is supplied for msgId. Do not supply these fields when the message text is not translated or localized, for example, for trace events.

**msgCatalogId**

The msgCatalogId field is listed as an optional value within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. This field is not required for problem determination events because not all problem determination events include translated message text Some cases exist where the value is not provided or available when formatting the Common Base Event. Do not supply this field when the message text is not translated or localized, for example, for trace events.

**msgCatalogTokens**

The msgCatalogTokens field is listed as an optional value within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. This field is not required for problem determination events because not all problem determination events include translated message text, and cases exist where the value is not provided or available when formatting the Common Base Event. This value contains the list of locale-independent values or message tokens that are inserted into the localized message text when creating a translated message.

These values are difficult to extract from a translated message without knowing the translated message template that is used to create the message. Do not supply this field when the message text is not translated or localized

The Common Base Event provides several mechanisms for providing additional data about an event, including this field, extended data elements, and extensions to the schema. Always use the msgCatalogTokens field to supply the list of message tokens that is included in the message text associated with an event. These values can also be supplied in other parts of the Common Base Event, but they must be included in this field.

**msgCatalog and msgCatalogType**

The msgCatalog and msgCatalogType fields are listed as optional values within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. These fields are not required for problem determination events because not all problem determination events include translated message text, and cases exist where the values are not provided or available when formatting the Common Base Event. Do not complete these fields when the message text has is not translated or localized, for example, for trace events.

***Extended data:***

The Common Base Event provides several methods for including this additional data, including extending the Common Base Event schema or supplying one or more ExtendedDataElement elements within the Common Base Event, which is the preferred approach.

The base information that is included in a Common Base Event might not be sufficient to represent all of the information captured by a component when creating a problem determination event.

Use an ExtendedDataElement element to represent a single data item. A Common Base Event can contain more than one of these elements, essentially one for each additional data item. A hint to the number and type of ExtendedDataElement elements is supplied by the extensionName value, but this information is only a hint. The usage of the attributes in the ExtendedDataElement element for problem determination events is the same as those for any other Common Base Event.

## Sample Common Base Event instance

This XML document is an example of a Common Base Event instance that is generated by a WebSphere Application Server application.

Use the following example for reference:

```
<CommonBaseEvent  creationTime="2004-09-18T04:03:28.484Z"
   globalInstanceId="myhost:1095479647062:1899"
   msg="WSVR0024I: Server server1 stopped"
   severity="10"
   version="1.0.1">

 ... several extendedDataElements for WebSphere Application Server internal use only ...

<sourceComponentId  component="com.ibm.ws.runtime.component.ServerCollaborator"
    componentIdType="Unknown"
    executionEnvironment="Windows 2000[x86]#5.0"
    instanceId="myhost\myhost\server1"
    location="myhost"
    locationType="Hostname"
    processId="1095479647062"
    subComponent="Unknown"
    threadId="Alarm : 0"
    componentType="http://www.ibm.com/namespaces/autonomic/WebSphereApplicationServer"/>

 <msgDataElement msgLocale="en_US">
  <msgCatalogTokens value="server1"/>
  <msgId>WSVR0024I< /msgId>
  <msgCatalogId>WSVR0024I< /msgCatalogId>
  <msgCatalog>com.ibm.ws.runtime.runtime< /msgCatalog>
 </msgDataElement>

 <situation categoryName="ReportSituation">
  <situationType xsi:type="ReportSituation" reasoningScope="EXTERNAL" reportCategory="LOG"/>
 </situation>

</CommonBaseEvent>
```

A number of extendedDataElement elements in the XML are used by WebSphere Application Server, but are not for application use because these elements might change.

The CommonBaseEvent element defines the Common Base Event instance. This element has a set of attributes that are common for all Common Base Events. This set includes the extensionName attribute, which defines the type or class of the Common Base Event instance, the creation time, severity, and priority.

Nested within the CommonBaseEvent element are elements giving more detail about the situation. The first of these elements is the situation element. This classification is standardized.

The CommonBaseEvent element also includes the sourceComponentId and the (optional) reporterComponentId elements. The sourceComponentId element describes where the situation occurred; the reporterComponentId describes where the situation is detected. If the sourceComponentId and the reporterComponentId elements are the same, the reporterComponentId element is omitted.

The attributes of both the sourceComponentId and the reporterComponentId elements are the same. They identify the component type, name, operating system, and network location. The content of these attributes provides vertical correlation of the stack of IT resources that are active when the Common Base Event is created.

Also included in the CommonBaseEvent element are contextDataElements elements that describe the context in which the situation occurred. This context correlates Common Base Event instances that are part of the same work. This correlation is called *horizontal correlation* because an instance of a particular context type correlates events at the same level of abstraction, for example at the business level, the application level, or at the middleware level.

FExtended data elements contain additional data that is used to describe a situation. In this example, an extended data element is added by WebSphere Application Server to describe the Java 2 Platform, Enterprise Edition (J2EE) component that generated the Common Base Event instance and some application data.

## Sample Common Base Event template

The content handler uses template information to fill in blanks in the Common Base Event when the Common Base Event complete method is called.

Components that use the WebSphere Application Server event factory home can include a Common Base Event template XML file to provide data to populate Common Base Events. Information that is already supplied in the event is not overridden if the same field is supplied in the template.

The following example illustrates a Common Base Event template:

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

<TemplateEvent
  version="1.0.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

 <CommonBaseEvent
  <sourceComponentId application="My Application" component="com.ibm.componentX"/>
  <extendedDataElements name="Sample ExtendedDataElement name" type="string">
   <values>Sample ExtendedDataElement value</values>
  </extendedDataElements>
 </CommonBaseEvent>

</TemplateEvent>
```

## Component identification for problem determination

This topic describes types of problem determination events.

A business application is made up of multiple components. A component can be made up of several internal subcomponents. Consistent application of these concepts is critical for effective problem determination of a business application; all of the parts of the application must use the same concepts and assumptions when creating and formatting events. Use the following definitions and examples when creating Common Base Events for problem determination.

**Business application**

A business application is the business logic and business data that is used to address a set of specific business requirements. A business application consists of several components of multiple types, combined in a unique manner by an enterprise, to provide the functions and resources that are needed to address those requirements. The primary creator and manager of a business application is the enterprise, and each enterprise or company creates unique business applications. Examples of business applications are the Payroll Application for the ACME Corporation and the Inventory Application for Spacely Sprockets.

**Components**

A business application is created and managed by the enterprise as a set of components. Components are deployable assets, which are developed either by the enterprise or a vendor, and managed by the enterprise. A component might be created by the enterprise, typically for use within a specific business application. For example, the ACME Corporation might create a set of enterprise beans to represent the business logic that is required by their Payroll Application. A component might also be an asset that is produced by a vendor and acquired by an enterprise. Examples of these components are hardware products, such as IBM eServers or Sun Solaris systems, or software products, such as IBM WebSphere Application Server, Oracle Database Servers.

**Subcomponents**

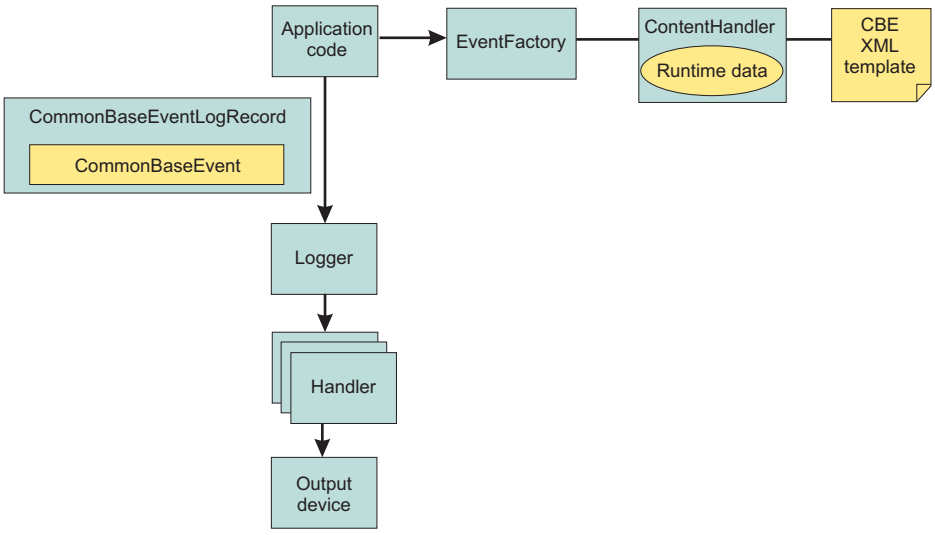
A specific component, depending on its complexity, might consist of several subcomponents. For example, the IBM WebSphere Application Server consists of many subcomponents, such as the enterprise bean container and the servlet engine. Subcomponent information is typically used only by the creator of the component to service the component, and as such are not separately deployable or manageable resources in the enterprise. The enterprise might deploy a change or update to a subcomponent, but only upon guidance from the component vendor and as part of the vendor's component. For example, a software fix for the enterprise bean container of the IBM WebSphere Application Server is packaged and deployed as a software update to the IBM WebSphere Application Server. Replacement of the processor in an IBM eServer is deployed as a physical part, but only as a part of the original deployed component, the IBM eServer.

# Logging with Common Base Event API and the Java logging API

In cases where the events that are generated by the Java logging API are insufficient to describe the event that needs capturing, you can create Common Base Events with the Common Base Event factory APIs.

## Before you begin

When you create a Common Base Event, you can add data to the Common Base Event before it is logged. The following diagram illustrates how application code can create and log Common Base Events:




## About this task

WebSphere Application Server is configured to use an event factory that automatically populates WebSphere Application Server-specific information into the Common Base Events that it generates. In general, it is good practice to create events using the WebSphere Application Server default Common

Base Event factory because this approach ensures consistency of Common Base Event content across events. However, you can create and use other Common Base Event factories.

Common Base Events are initiated and logged in the following sequence:

1. Application code invokes the createCommonBaseEvent method on the EventFactory class to create a CommonBaseEvent.
2. Application code wraps CommonBaseEvent event in a CommonBaseEventLogRecord record, and adds event-specific data.
3. Application code calls the CommonBaseEvent event complete method.
4. The CommonBaseEvent event invokes the ContentHandler completeEvent method.
5. The ContentHandler handler adds XML template data to the CommonBaseEvent event. Not all ContentHandler handlers support templates.
6. The ContentHandler handler adds runtime data to the CommonBaseEvent event.
7. Application code passes the CommonBaseEventLogRecord record to the logger using the Logger.log method.
8. Logger passes CommonBaseEventLogRecord record to Handlers.
9. Handlers format data and write to the output device.

- You can use the default Common Base Event factory to generate content. Read “Generate Common Base Event content with the default event factory” for more information.
- If you do not wish to use the default event factory, you can create custom content handlers and event factories.
  1. Create a custom factory home. Read “Creating custom Common Base Event factory homes” on page 70.
  2. Create a custom content handler. Read “Creating custom Common Base Event content handlers” on page 68.

## Results

After completing all the above steps you will have a Common Base event based on your configuration settings.

## Generate Common Base Event content with the default event factory

A default Common Base Event content handler populates Common Base Events with WebSphere Application Server runtime information. This content handler can also use a Common Base Event template to populate Common Base Events.

The default content handler is used when the server creates CommonBaseEventLogRecords as would be the case in the following example:

```
// Get a named logger
Logger logger = Logger.getLogger("com.ibm.someLogger");
// Log to the logger -- implicitly the default content handler
// will be associated with the CommonBaseEvent contained in the
// CommonBaseEventLogRecord.
logger.warning("MSG_KEY_001");
```

To specify a Common Base Event template in the above case, a Logger.properties file would need to be provided with an eventfactory entry for *com.ibm.someLogger*. If a valid template is found on the classpath, then the Logger’s event factory will use the specified template’s content in addition to the WebSphere Application Server runtime information when populating Common Base Events. If the template is not found on the classpath, or is invalid, then the Logger’s event factory will only use the WebSphere Application Server runtime information when populating Common Base Events.

The default content handler is also associated with the event factory home supplied in the global event factory context. This is convenient for creating Common Base Events that need to be populated with content similar to that generated from the WebSphere Application Server:

```
// Request the event factory from the global event factory home
EventFactory eventFactory = EventFactoryContext.getInstance().getEventFactoryHome().getEventFactory(templateName);

// Create a Common Base Event
CommonBaseEvent commonBaseEvent = eventFactory.createCommonBaseEvent();

// Complete the Common Base Event using content from the template (if specified above)
// and the server runtime information.
eventFactory.getContentHandler().completeEvent(commonBaseEvent);
```

In the above example, if the template referenced by *templateName* is found on the classpath, and the template is valid, then the event factory home will return an event factory which uses a content handler that combines the template's content with the WebSphere Application Server runtime information when populating Common Base Events. If the template is not found on the classpath, or is invalid, then the event factory home will return an event factory which uses a content handler that uses only the WebSphere Application Server runtime information when populating Common Base Events.

The default content handler populates Common Base Events in the server environment with the following runtime information:

**CommonBaseEvent.globalInstanceId**
**Value:** The *unique_record_id*

Set this value only if the CommonBaseEvent.globalInstanceId value is null before the completeEvent method is called.

**CommonBaseEvent.msg**
**Value:** A localized message that is based on the MsgDataElement element.

Set this value only if the CommonBaseEvent.msg message is null before the completeEvent method is called.

**CommonBaseEvent.severity**
**Value:** Set based on the value of level set on the CommonBaseEventLogRecord record, if level >= Level.SEVERE, set to `50`; if level >= Level.WARNING, set to `30`; the default is set to `10`.

Set this value only if the CommonBaseEvent.severity value is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.component**
**Value:**Set based on the LoggerName value that is set on the CommonBaseEventLogRecord record.

Set this value only if the CommonBaseEvent.ComponentIdentification.component is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.componentIdType**
**Value:** ″Unknown″

Set this value only if the CommonBaseEvent.ComponentIdentification.componentIdType value is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.executionEnvironment**
**Value:** `OSname[OSarch]#OSversion`

Set this value only if the CommonBaseEvent.ComponentIdentification.executionEnvironment value is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.instanceId**
**Value:** `cellName\nodeName\serverName`

Set this value only if the CommonBaseEvent.ComponentIdentification.instanceId value is null before the completeEvent method is called. Set only in a server environment because this value is ignored in a client application.

**CommonBaseEvent.ComponentIdentification.location**
**Value:**The host name

Set this value only if both the CommonBaseEvent.ComponentIdentification.location and the CommonBaseEvent.ComponentIdentification.locationType values are null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.locationType**
**Value:** The host name

Set this value only if both the CommonBaseEvent.ComponentIdentification.location and the CommonBaseEvent.ComponentIdentification.locationType values are null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.processId**
**Value:** An internally generated representation of the process number.

Set this value only if the CommonBaseEvent.ComponentIdentification.processId value is null before the completeEvent method is called

**CommonBaseEvent.ComponentIdentification.subComponent**
**Value:** Set based on values of the sourceClassName and the sourceMethodName names that are set on the sourceClassName.sourceMethodName name of the CommonBaseEventLogRecord record.

Set this value only if the CommonBaseEvent.ComponentIdentification.subComponent values is null before the completeEvent method is called and both the sourceClassName and the sourceMethodName names are set.

**CommonBaseEvent.ComponentIdentification.threadId**
**Value:** Set to the value of the Java Virtual Machine (JVM) thread name.

Set this value only if the CommonBaseEvent.ComponentIdentification.threadId values is null before the completeEvent value is called.

**CommonBaseEvent.ComponentIdentification.componentType**
**Value:** http://www.ibm.com/namespaces/autonomic/WebSphereApplicationServer

Set this value only if the CommonBaseEvent.ComponentIdentification.componentType values is null before the completeEvent method is called.

**CommonBaseEvent.MsgDataElement.msgLocale**
**Value:** Set based on the default locale of the JVM.

Set this value only if the CommonBaseEvent.msg value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.categoryName**
**Value:** `ReportSituation`

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.situationType.type**
**Value:** `ReportSituation`

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.situationType.reasoningScope**
**Value:** EXTERNAL

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.situationType.reportCategory**

**Value:** LOG

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

The sourceComponentIdentification value is populated if no reporterComponentIdentification ID exists when the completeEvent method is invoked on the content handler. Otherwise, the reporterComponentIdentification ID is populated instead.

## Common Base Event content handler

Content handlers populate data into Common Base Events when the Common Base Event complete method is invoked. You can associate content handlers with Common Base Event templates, which provide default information to transfer into each Common Base Event.

Content handlers might also provide any other information that is relevant to completing the population of the Common Base Event, such as appropriate runtime defaults. The use of content handlers ensures consistency of field use in the Common Base Event within a component or within a set of components that share the same runtime. For example, some content handlers support the specification of a template. If used consistently across a component, this template ensures that all events for that component have the same template information filled in. Similarly, some content handlers can also supply runtime information to their associated Common Base Events. If consistently used throughout the entire runtime, runtime information ensures that all events use runtime data in a similar way.

The event factory home that is used in the WebSphere Application Server runtime is associated with a content handler that both reads from a template, and supplies runtime data. Have components use Event Factories that are obtained from this event factory home with their own templates, to produce consistency between application events and server events.

More details can be found in“Creating custom Common Base Event content handlers” or the API documentation for org.eclipse.hyades.logging.events.cbe.ContentHandler at www.eclipse.org/hyades.

## Creating custom Common Base Event content handlers

Create a custom Common Base Event content handler or template to automate configuration or values for specific events.

### Before you begin

A *content handler* is an object that automatically sets the property values of each event based on any arbitrary policies that you want to use.

The following content handler classes were added to WebSphere Application Server to facilitate the use of the Common Base Event infrastructure:

| Class Name | Description |
|---|---|
| WsContentHandlerImpl | This provides an implementation of org.eclipse.hyades.logging.events.cbe.ContentHandler specifically for use in the WebSphere Application Server environment. This content handler completes Common Base Events using information from the WebSphere Application Server runtime, and it uses the same content handler as is used internally by the WebSphere Application Server when completing Common Base Events for logging. |

| | |
|---|---|
| WsTemplateContentHandlerImpl | This provides the same function as WsContentHandlerImpl, but it extends the org.eclipse.hyades.logging.events.cbe.impl.TemplateContentHandlerImpl class to enable the use of a Common Base Event template. Template content takes precedence in cases where the template data specifies values for the same Common Base Event fields as does the WsContentHandlerImpl. |

### About this task

In some situations, you might want some event property data set automatically for every event that you create. This automation is a way to fill in certain standard values that do not change, such as the application name, or to set some properties based on information that is available from the runtime environment, like creation time or thread information. You can set property data automatically by creating a content handler.

- Use the following code sample to implement the CustomContentHandler class:

```
public class CustomContentHandler extends WsContentHandlerImpl {

 public CustomContentHandler() {
  super();
  // TODO Custom initialization code goes here
 }

 public void completeEvent(CommonBaseEvent cbe) throws CompletionException {
  // following code will add WAS content to the Content Base Event
  super.completeEvent(cbe);
  // TODO Custom content can be added to the Content Base Event here
 }
}
```

- The following shows how to implement the CustomTemplateContentHandler class:

```
public class CustomTemplateContentHandler extends WsTemplateContentHandlerImpl {

 public CustomTemplateContentHandler() {
  super();
  // TODO Custom initialization code goes here
 }

 public void completeEvent(CommonBaseEvent cbe) throws CompletionException {
  // following code will add WAS content to the Content Base Event
  super.completeEvent(cbe);
  // TODO Custom content can be added to the Content Base Event here
 }
}
```

### Results

You now have a content handler or a custom content handler template based on the settings that you specified.

## Common Base Event factory home

Event Factory homes provide Event Factory instantiation that is based on a unique factory name.

Event factory home implementations are tightly coupled with content handlers that are used to populate Common Base Events with template or default data. Event factory instances are maintained by the associated event factory home, based on their unique name. For example, when application code requests a named event factory, the newly created Event Factory instance is returned and persisted for future requests for that named event factory. An abstract event factory home class provides the implementation

for the APIs in the event factory home interface. Implementers extend the abstract event factory home class and implement the createContentHandler API to create a typed content handler that is based on the type of event factory home implementation.

In WebSphere Application Server, the default event factory home that is obtained with a call to EventFactoryContext.getInstance.getEventFactoryHome method is associated with a ContentHandler handler capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactoryHome at www.eclipse.org/hyades.

## Creating custom Common Base Event factory homes

Use custom Common Base Event factory homes to control configuration and implementation of unique event factories.

### Before you begin

Event factory homes create and provide homes for Event Factory instances. Each event factory home has a content handler. This content handler is assigned to every event factory the event factory home creates. In turn, when a Common Base Event is created, the content handler from the event factory is assigned to it. Event factory instances are maintained by the associated event factory home, based on their unique name. For example, when application code requests a named event factory, the newly created event factory instance is returned and persisted for future requests for that named event factory.

The following classes were added to facilitate the use of event eactory homes for logging Common Base Events:

| Class Name | Description |
|---|---|
| WsEventFactoryHomeImpl | This class extends the org.eclipse.hyades.logging.events.cbe.impl.AbstractEventFactoryHome class. This event factory home returns event factory instances associated with the WsContentHandlerImpl content handler. The WsContentHandlerImpl is the content handler used by the WebSphere Application Server by default when no event factory template is in use. |
| WsTemplateEventFactory HomeImpl | This class extends the org.eclipse.hyades.logging.events.cbe.impl.EventXMLFileEventFactoryHomeImpl class. This event factory home returns event factory instances associated with the WsTemplateContentHandlerImpl Content Handler. The WsTemplateContentHandlerImpl is the content handler used by the WebSphere Application Server when an Event Factory template is required. |

### About this task

Custom event factory homes support the use of Common Base Event for logging in WebSphere Application Server and make logging easy and consistent between the WebSphere Application Server runtime and the exploiters of this API. The CustomEventFactoryHome and CustomTemplateEventFactoryHome classes will be used to obtain an event factory. These classes are there to make sure the correct content handler is being used with a particular event factory. The CustomEventFactoryHelper class is an example of how the infrastructure provider can hide the factory selection details from infrastructure users, using their own set of parameters to decide which the appropriate event factory is.

- The following code samples provide examples of how to implement and use the CustomEventFactoryHome class.
  1. Implementation of the CustomEventFactoryHome class is as follows:

```
public class CustomEventFactoryHome extends AbstractEventFactoryHome {

 public CustomEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomContentHandler();
 }
}
```

2. The following is an example of how to use the CustomEventFactoryHome class:

```
// get the event factory
  EventFactory eventFactory=(new CustomEventFactoryHome()).getEventFactory("XYZ");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

- For the CustomTemplateEventFactoryHome class you can use the following code for implementation and use:

1. Implement the CustomTemplateEventFactoryHome class by using this code:

```
public class CustomTemplateEventFactoryHome extends
  EventXMLFileEventFactoryHomeImpl {

 public CustomTemplateEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomTemplateContentHandler();
 }
}
```

2. Use the CustomTemplateEventFactoryHome class by following this sample code:

```
// get the event factory
  EventFactory eventFactory=(new
    CustomTemplateEventFactoryHome()).getEventFactory("XYZ");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

- The CustomEventFactoryHelper class can be implemented and used by following the code below:

1. Implement the custom CustomEventFactoryHelper class using this code:

```
public class CustomTemplateEventFactoryHome extends
  EventXMLFileEventFactoryHomeImpl {

 public CustomTemplateEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
```

```
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomTemplateContentHandler();
 }
}
```

Figure 4 CustomTemplateEventFactoryHome class

```
public class CustomEventFactoryHelper {
 // name of the event factory to use
 public static final String FACTORY_NAME="XYZ";

 public static EventFactory getEventFactory(String param1, String param2) {
  EventFactory factory=null;
  switch (resolveFactory(param1,param2)) {
  case 1:
   factory=(new CustomEventFactoryHome()).getEventFactory(FACTORY_NAME);
   break;
  case 2:
   factory=(new
    CustomTemplateEventFactoryHome()).getEventFactory(FACTORY_NAME);
   break;

  default:
   // Add default for event factory
   break;
  }
  return factory;
 }

 private static int resolveFactory(String param1, String param2) {
  int factory=0;
  // Add code here to resolve which factory to use
  return factory;
 }
}
```

2. To use the CustomEventFactoryHelper class, use the following code:

```
// get the event factory
  EventFactory eventFactory=
   CustomEventFactoryHelper.getEventFactory("param1","param2","param3");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

### Results

Use the information provided here to implement a custom content factory home and the associated classes based on the settings that you specify.

## Common Base Event factory context

The event factory context provides a service to look up event factory homes. Retrieve the event factory context using a call to the EventFactoryContext.getInstance method.

Using this class, you can look up the event factory homes by name, and avoid the need to include the typed home in code. The EventFactoryHome name must be located on the class path to be found. The EventFactoryContext context also stores an EventFactoryHome name as a default, which can be obtained with a call to the EventFactoryContext.getInstance.getEventFactoryHome method.

In WebSphere Application Server, the EventFactoryContext context is configured with a default EventFactoryHome name which is associated to a ContentHandler handler that is capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactory at www.eclipse.org/hyades.

### Common Base Event factory

Use event factories to create Common Base Events and complete event properties with associated content handlers.

Content handlers populate data into Common Base Events when the Common Base Event invokes the complete method. All event properties set by the application code have priority over all properties that are specified by the content handler. Event factory implementations are tightly coupled with the content handler instance, which is associated with the event factory when the event factory is instantiated. Factory instances can be retrieved only from their associated event factory home. Event factory instances are retrieved and maintained based on unique names. Event factory names are hierarchical; they are represented using the standard Java dot-delimited, name-space naming conventions.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactory at www.eclipse.org/hyades.

## java.util.logging -- Java logging programming interface

The java.util.logging.Logger class provides a variety of methods with which data can be logged.

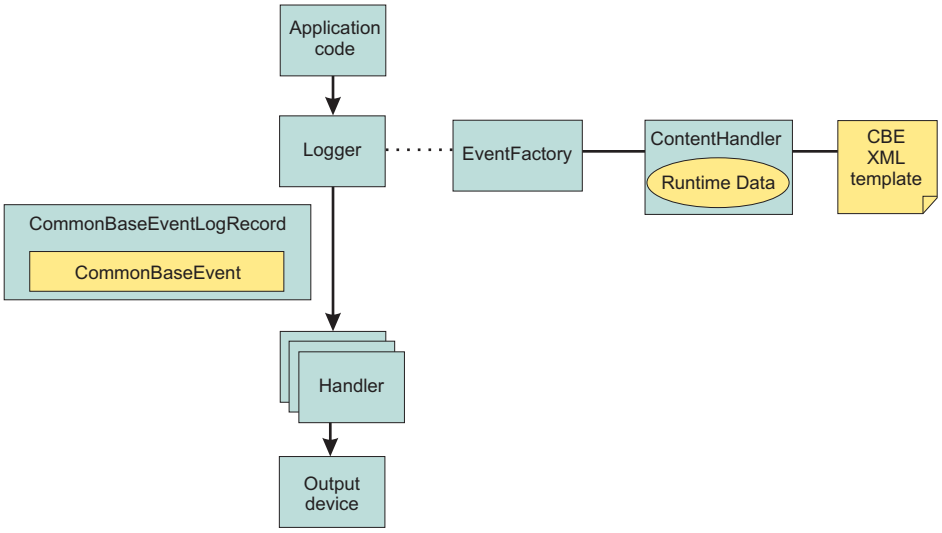
In the WebSphere Application Server, the Java logging API (java.util.logging) automatically creates Common Base Events for events that are logged at the WsLevel.DETAIL level or above (including WsLevel.DETAIL, Level.CONFIG, Level.INFO, WsLevel.AUDIT, Level.WARNING, Level.SEVERE, and WsLevel.FATAL). These Common Base Events are created using the event factory that is associated with the logger to which the message is logged. If no event factory is specified, WebSphere Application Server uses a default event factory which automatically fills in WebSphere Application Server-specific information.

The WebSphere Application Server uses a special implementation of the java.util.logging.Logger class that automatically creates Common Base Events for the following methods:

- config
- info
- warning
- severe
- log: All variants except log(LogRecord) when used with the WsLevel.DETAIL level or more severe levels
- logp: When used with the WsLevel.DETAIL level or more severe levels
- logrb: When used with the WsLevel.DETAIL level or more severe levels

The WebSphere Application Server logger implementation is used only for named loggers for example, loggers that are instantiated with calls, such as Logger.getLogger(″com.xyz.SomeLoggerName″). Loggers instantiated with calls to the Logger.getAnonymousLogger and Logger.getLogger, or Logger.global methods do not use the WebSphere Application Server implementation, and do not automatically create Common Base Events for logging requests made to them. Log records that are logged directly with the Logger.log(LogRecord) method are not automatically converted by WebSphere Application Server loggers into Common Base Events.

The following diagram illustrates how application code can log Common Base Events:




The Java logging API processing of named loggers and message-level events proceeds as follows:

1. Application code invokes the named logger (WsLevel.DETAIL or above) with event-specific data.
2. The logger creates a Common Base Event using the createCommonBaseEvent method on the event factory that is associated with the logger.
3. The logger creates a Common Base Event using the event factory associated to the logger.
4. The logger wraps the common base event in a CommonBaseEventLogRecord record, and adds event-specific data.
5. The logger calls the Common Base Event complete method.
6. The Common Base Event invokes the ContentHandler completeEvent method.
7. The content handler adds XML template data to the Common Base Event (including for example, the component name). Not all content handlers support templates.
8. The content handler adds runtime data to the Common Base Event (including for example, the current thread name).
9. The logger passes the CommonBaseEventLogRecord record to the handlers.
10. The handlers format data and write to the output device.

## Logger.properties file

Use the Logger.properties file to set logger attributes for your component.

The properties file is loaded the first time the Logger.getLogger(loggername) method is called within an application. The Logger.properties file must be either on the WebSphere Application Server class path, or the context class path.

The logging subsystem uses Common Base Events to represent all the messages in the WebSphere Application Server activity.log file. You can specify your own event factory template to be used with your loggers. Use the eventfactory property in your Logger.properties file. See “Sample Common Base Event template” on page 63 for details on the Common Base Event template.

By convention, the name of the event factory template file should be the fully qualified package name of the package using the template. The name of the file must end with the `.event.xml` extension. For example, a valid event factory template file name for the com.abc.somepackage package is:

```
com.abc.somepackage.event.xml
```

When you specify the property value for the eventfactory property in the Logger.properties file, include the full path name with no leading slash relative to the root of your class path entry. Do not include the `.event.xml` extension.

For example, if the template files from the example above are located in the `com/abc/templates` directory, the valid value for the eventfactory property is:

```
com/abc/templates/com.abc.somepackage
```

Finally, if this event factory template file is used by the com.abc.somepackage.SomeClass logger, then the following entry will appear in the Logger.properties file:

```
com.abc.somepackage.SomeClass.eventfactory=com/abc/templates/com.abc.somepackage
```

## Logging Common Base Events in WebSphere Application Server

The following practices ensure consistent use of Common Base Events within your components, and between your components and WebSphere Application Server components.

Follow these guidelines:

- Use a different logger for each component. Sharing loggers across components gets in the way of associating loggers with component-specific information.
- Associate loggers with event templates that specify source component identification. This association ensures that the source of all events created with the logger is properly identified.
- Use the same template for directly created Common Base Events (events created using the Common Base Event factories) and indirectly created Common Base Events (events created using the Java logging API) within the same component.
- Avoid calling the complete method on Common Base Events until you are finished adding data to the Common Base Event and are ready to log it. This approach ensures that any decisions made by the content handler based on data already in the event are made using the final data.

The following sample `Logger.properties` file entry demonstrates how to associate the com.ibm.componentX logger with the com.ibm.componentX event factory:

```
com.ibm.componentX.eventfactory=com.ibm.componentX
```

The following sample code demonstrates the use of the same event factory setting for direct (Part 1) and indirect (Part 2) Common Base Event logging:

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

<TemplateEvent>
  version="1.0.1"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

 <CommonBaseEvent>
   <sourceComponentId application="My application" component="com.ibm.componentX"/>
   <extendedDataElements CommonBaseEventname="Sample ExtendedDataElement name" type="string">
   <values>Sample ExtendedDataElement value</values>
  </extendedDataElements>
 < /CommonBaseEvent>

< /TemplateEvent>
```

# Chapter 3. Diagnosing problems (using diagnosis tools)

Various diagnosis tools are provided to help you determine the source and impact of problems occurring in your application serving environment.

### About this task

The purpose of this section is to aid you in understanding why your enterprise application, application server, or WebSphere Application Server is not working and to help you resolve the problem. Unlike performance tuning, which focuses on solving problems associated with slow processes and non-optimized performance, problem determination focuses on finding solutions to functional problems.

1. If deploying or running an application results in exceptions such as ClassNotFoundException, use the Class Loader Viewer to diagnose problems with class loaders.
2. If you already have an error message and want to quickly look up its explanation and recommended response, look up the message by expanding the Messages section of the Information Center under **Reference** > **Messages**.
3. For help in knowing where to find error and warning messages, interpreting messages, and configuring log files, see Working with message logs.
4. Difficult problems can require the use of tracing, which exposes the low-level flow of control and interactions between components. For help in understanding and using traces, see Working with trace.
5. For help in adding log and trace capability to your own application, see “Configuring Java logging using the administrative console” on page 4.
6. For help in using settings or tools to help you diagnose the problem, see Working with troubleshooting tools. Some of these tools are bundled with the product, and others are freely downloadable.
7. To learn how to work with Diagnostic Providers, see Working with Diagnostic Providers..
8. To find out how to look up documented problems, common mistakes, WebSphere Application Server prerequisites, and other problem-determination information on the WebSphere Application Server public Web site, or to obtain technical support from IBM, see Obtaining help from IBM.
9. The Troubleshoot IBM Developer Kit for Java describes debugging techniques and the diagnostic tools that are available to help you solve problems with Java. It also gives guidance on how to submit problems to IBM.
10. For current information available from IBM Support on known problems and their resolution, see the WebSphere Application Server Product support page. For last minute updates, limitations, and known problems, refer to the Release notes section.
11. IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the Must gather documents page for information to gather to send to IBM Support.

## Troubleshooting class loaders

Class loaders find and load class files. For a deployed application to run properly, the class loaders that affect the application and its modules must be configured so that the application can find the files and resources that it needs. Diagnosing problems with class loaders can be complicated and time-consuming. To diagnose and fix the problems more quickly, use the administrative console class loader viewer to examine class loaders and the classes loaded by each class loader.

## Before you begin

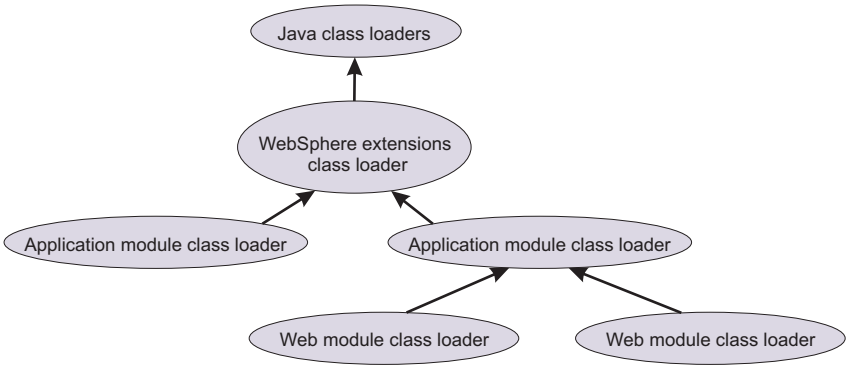
This topic assumes that you have installed an application on a server supported by the product and you want to examine class loaders used by the application or its modules. The modules can be Web modules (.war files) or enterprise bean (EJB) modules (.jar files). The class loader viewer enables you to examine class loaders in a runtime environment.

This topic also assumes that you have enabled the class loader viewer service. Click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Class loader viewer service**, enable the service and restart the server.

## About this task

The runtime environment of WebSphere Application Server uses the following class loaders to find and load new classes for an application in the following order:

1. The bootstrap, extensions, and CLASSPATH class loaders created by the Java virtual machine
2. A WebSphere extensions class loader
3. One or more application module class loaders that load elements of enterprise applications running in the server
4. Zero or more Web module class loaders




Each class loader is a child of the previous class loader. That is, the application module class loaders are children of the WebSphere extensions class loader, which is a child of the CLASSPATH Java class loader. Whenever a class needs to be loaded, the class loader usually delegates the request to its parent class loader. If none of the parent class loaders can find the class, the original class loader attempts to load the class. Requests can only go to a parent class loader; they cannot go to a child class loader. After a class is loaded by a class loader, any new classes that it tries to load reuse the same class loader or go up the precedence list until the class is found.

If the class loaders that load the artifacts of an application are not configured properly, the Java virtual machine (JVM) might throw a class loading exception when starting or running that application. “Class loading exceptions” on page 80 describes the types of exceptions caused by improperly configured class loaders and suggests ways to use the class loader viewer to correct configurations of class loaders. The types of exceptions include:

- ClassCastException
- ClassNotFoundException
- NoClassDefFoundException
- UnsatisfiedLinkError

Use the class loader viewer to examine class loaders and correct problems with application or class loader configurations.

- Examine a tree view that lists all installed applications and their modules. The modules can be Web modules (.war files) or EJB modules (.jar files).

  Click **Troubleshooting** → **Class loader viewer** to access the Enterprise applications topology page.
- Examine the class loader delegation hierarchy.

  On the Enterprise applications topology page, select a module to access the Class loader viewer page. The page lists the class loaders visible to Web and EJB modules in an installed enterprise application. This page helps you to determine which class loaders loaded files of a module and to diagnose problems with class loaders.

  The delegation hierarchy is determined by the class loader delegation mode, or *class loader order*, specified for an application or Web module. The value can be either `Classes loaded with parent class loader first` or `Classes loaded with local class loader first (parent last)`. Refer to the Configure class loaders step for more information.
- Export information on class loaders.
  1. On the Class loader viewer page, click **Export**.
  2. Select to open a browser or editor on the class loader information or to save the information to disk in XML format.
  3. Click **OK**, and specify any additional information requested by the system.
- Display information about class loaders visible to the module in an HTML table format.

  On the Class loader viewer page, click **Table View**. The Table View page displays the following information:

| Class loader attribute | Description |
|---|---|
| **Delegation** | Indicates whether the class loader delegates the loading of the module to its parent class loader. A value of `true` implies that the class loader of the parent application is being used (`Classes loaded with parent class loader first`). A value of `false` implies that the module class loader is being used (`Classes loaded with local class loader first (parent last)`). Refer to the Configure class loaders step for more information. |
| **Classpath** | Lists the paths over which the class loader searches for classes and resources. |
| **Classes** | Lists the names of classes loaded in the JVM by this class loader. |

  The **Table View** option does not return a value when out-of-memory errors are generated. The out-of-memory errors might be related to a memory leak. To examine information about class loaders in a table, resolve the out-of-memory problem, and then click **Table View** again.
- Search class loaders.

  On the Class loader viewer page, click **Search** to access the Search page, on which you can search class loaders for the following:
  – Specific strings
  – Specific .jar files
  – The names of files in a specific directory
  – The names of files loaded by a specific class loader

  The search is case-sensitive. “Class loading exceptions” on page 80 describes several uses of the Search page.
- Configure class loaders. You can configure class loaders for the following:
  – All applications installed on a specific server.
  – A specific application
  – A specific Web module

  **Note:** For detailed information about server, application, and Web class loaders, see the chapter on class loading in the *Developing and deploying applications* PDF book.

Class loader configuration determines which class loader loads the classes and resource files for an application or Web module. Application and WAR module class loader configuration settings include **Class loader order** and **WAR class loader policy**.

A **Class loader order** value can be either `Classes loaded with parent class loader first` or `Classes loaded with local class loader first (parent last)`. The default is `Classes loaded with parent class loader first`. A class loader with the `Classes loaded with parent class loader first` mode delegates loading a class or resource to its immediate parent class loader before searching its classpath.

When troubleshooting class loading problems, you might need to override classes visible to a parent class loader. To override such classes with those specific to an application, set the **Class loader order** to `Classes loaded with local class loader first (parent last)` on the class loader that contains the application classes on its classpath. An application can override classes visible to a parent class loader, but doing so can result in a ClassCastException or UnsatisfiedLinkError if there is a mixed use of overridden classes and non-overridden classes.

For example, under default class loader policies, a Web module has its own Web module (WAR) class loader to load its artifacts, which are typically in the `WEB-INF/classes` and `WEB-INF/lib` directories. An application module class loader is the immediate parent of this WAR class loader. To ensure that the Web module class loader searches these paths for a particular class or resource first, before delegating the load operation to the application module class loader, set the **Class loader order** of the Web module to `Classes loaded with local class loader first (parent last)`.

Class loader policies determine the structure of the application and WAR module class loaders. Under the default policies, every running application EAR has its own application module class loader, and every Web module has its own WAR module class loader. The default policies ensure Java EE compliance regarding visibility and isolation among application artifacts. Changing the default policies is not suggested when troubleshooting class loading problems.

### What to do next

If you continue to have class loader problems, refer to “Class loading exceptions” and to the class loading chapter of the *Developing and deploying applications* PDF book.

## Class loading exceptions

What kind of class-loading error do you see when you develop an application or start an installed application?



### ClassCastException

A class cast exception results when the following conditions exist and can be corrected by the following actions:

- The type of the source object is not an instance of the target class (type).
- The class loader that loaded the source object (class) is different from the class loader that loaded the target class.
- The application fails to perform or improperly performs a narrow operation.

**The type of the source object is not an instance of the target class (type).**

This is the typical class cast exception. You can diagnose whether the source object of a cast statement is not an instance of the target class (type) by examining the class signature of the source object class, then verifying that it does not contain the target class in its ancestry and the source object class is different than the target class. You can obtain class information by inserting a simple print statement in your code. For example:

```
System.out.println( source.getClass().getName() + ":" + target.getClass().getName() );
```

Or use a javap command. For example:

```
javap java.util.HashMap
Compiled from "HashMap.java"
public class java.util.HashMap extends java.util.AbstractMap
             implements java.util.Map,java.lang.Cloneable,java.io.Serializable {
```

**The class loader that loaded the source object (class) is different from the class loader that loaded the target class.**

Assuming that the type of the source object is an instance of the target class, a class cast exception occurs when the class loader that loaded the source object's class is different that the class loader that loaded the target class. This condition might occur when the target class is visible on the classpaths of more than one class loader in the WebSphere Application Server runtime environment. To correct this problem, use the Search and Search by class name console pages used to diagnose problems with class loaders:

1. Click **Troubleshooting** → **Class loader viewer** → ***module_name*** → **Search** to access the Search page.
2. For **Search type**, select **Class/Package**.
3. For **Search terms**, type the name of the class that is loaded by two class loaders.
4. Click **OK**. The Search by class name page is displayed, listing all class loaders that load the class.

   If there is more than one class loader listed, then the target class was loaded by more than one class loader. Because the source object is an instance of the target class, the class loader that loaded the source object class is different from the class loader that loaded the target class.
5. Return to the Class loader viewer page and examine the classpath to determine why two different class loaders load the class.
6. Correct your code so that the class is visible only to the appropriate class loader.

**The application fails to perform or improperly performs a narrow operation.**

A class cast exception can occur because, when the application is resolving a remote enterprise bean (EJB) object, the application code does not perform a narrow operation as required. The application must perform a narrow operation after looking up a remote object. Examine the application and determine whether it looks up a remote object and, if so, the result of the lookup is submitted to a narrow method.

The narrow method must be invoked according to the EJB 2.0 programming model. In particular, the target class submitted to the narrow method must be the exact, most derived interface of the EJB. This also causes a class cast exception in the WebSphere Application Server runtime environment. Examine the application and determine whether the target class submitted to the narrow method is a super-interface of the EJB that is specified, not the exact EJB type; if so, modify the application to invoke narrow with the exact EJB interface.

Lastly, if a class cast exception occurs during a narrow operation, verify that the narrow method is being applied to the result of a remote EJB lookup, not to a local enterprise bean. A narrow is not used for local lookups. Examine the application or module deployment descriptor to ensure that the object being narrowed is not a local object.

## ClassNotFoundException

A class not found exception results when the following conditions exist and can be corrected by the following actions:

- The class is not visible on the logical classpath of the context class loader.
- The application incorrectly uses a class loader API.
- A dependent class is not visible.

**The class is not visible on the logical classpath of the context class loader.**

The class not found is not in the logical class path of the class loader associated with the current

thread. The logical classpath is the accumulation of all classpaths searched when a load operation is invoked on a class loader. To correct this problem, use the Search page to search by class name and by Java archive (JAR) name:

1. Click **Troubleshooting** → **Class loader viewer** → ***module_name*** → **Search** to access the Search page.
2. For **Search type**, select **Class/Package**.
3. For **Search terms**, type the name of the class that is not found.
4. Click **OK**. The Search by class name page is displayed, listing all class loaders that load the class.
5. Examine the page to see if the class exists in the list.
6. If the class is not in the list, return to the Search page. For **Search terms**, type the name of the .jar file for the class; for **Search type**, select **JAR/Directory**.
7. Click **OK**. The Search by Path page is displayed, listing all directories that hold the JAR file.

If the JAR file is not in the list, the class likely is not in the logical class path, not readable or an alternate class is already loaded. Move the class to a location that enables it to be loaded.

**The application incorrectly uses a class loader API.**

An application can obtain an instance of a class loader and call either the loadClass method on that class loader, or it can call Class.forName(*class_name*, *initialize*, *class_loader*) with that class loader. The application may be incorrectly using the class loader application programming interface (API). For example, the class name is incorrect, the class is not visible on the logical classpath of that class loader, or the wrong class loader was engaged.

To correct this problem, determine whether the class exists and whether the application is properly using the class loader API. Follow the steps in The class is not visible on the logical classpath of the context class loader to determine whether the class is loaded. If the class has not been loaded, attempt to correct the application and see if the class loads. If the class is in the class path with proper permission and is not being overridden by another factory class, examine the API used to load the class.

1. Click **Troubleshooting** → **Class loader viewer** → ***module_name*** → **Search** to access the class loader Search page.
2. For **Search type**, select **Class/Package**.
3. For **Search terms**, type the name of the class.
4. Click **OK**. The Search by class name page is displayed, listing all class loaders that load the class.
5. Examine the page to see if the class exists in the list.
6. If the class is in the list and a ClassNotFound exception was thrown, then the .jar file or class is not in the correct context or a wrong API call in the current context was used.

   If the class is not in the list, return to the Search page and do the following:
   a. Search for the class that generated the exception; that is, the class calling Class.forName.
   b. See which class loader loads the class.
   c. Determine whether the class loader has access or can load the class not found by evaluating the class path of the class loader.

**A dependent class is not visible.**

When a class loader *clsldr* loads a class *cls*, the Java virtual machine (JVM) invokes *clsldr* to load the classes on which *cls* depends. Dependent classes must be visible on the logical classpath of *clsldr*, otherwise an exception occurs. This condition typically occurs when users make WebSphere Application Server classes visible to the JVM, or make application classes visible to the JVM or to the WebSphere extensions class loader. For example:

- Class A depends on Class B.
- Class A is visible to the WebSphere extensions class loader.

- Class B is visible on the local classpath of a WAR module class loader, not the WebSphere extensions class loader classpath.

When the JVM loads class A using the WebSphere extensions class loader, it then attempts to load Class B using the same class loader and ultimately creates a class not found exception.

To correct this problem:
1. Make the application-specific classes visible to the appropriate application class loader.
2. Search for the class not found (Class B).
3. If Class B is in the proper location, search for the class that loads the dependent class (Class A) in the Class loader viewer.
4. If the class is loaded and a ClassNotFound exception was thrown, then the `.jar` file or class is not in proper context or the wrong API call in the current context was used.

   If no class was found, do the following:
   a. Search for the class that generated the exception; that is, the class calling Class.forName.
   b. See which class loader loads the class.
   c. Determine whether the class loader has access or can load the class not found by evaluating the class path of the class loader.
5. Ensure that the caller class (Class B) is visible to the JVM or WebSphere extensions class loader.

## NoClassDefFoundException

A no class definition found exception results when the following conditions exist and can be corrected by the following actions:

**The class is not in the logical class path.**
Refer to “ClassNotFoundException” on page 81 for information.

**The class cannot load.**
There are various reasons for a class not loading. The reasons include: failure to load the dependent class, the dependent class has a bad format, or the version number of a class.

## UnsatisfiedLinkError

A linkage error results when the following conditions exist and can be corrected by the following actions:
- A user action caused the error.
- System.mapLibraryName returns the wrong library file.
- The native library is already loaded.
- A dependent native library was used.

**A user action caused the error.**

Several user actions can result in a linkage error:

**A library extension name is incorrect for the platform.**

**System.loadLibrary is passed an incorrect parameter.**

**The library is not visible.**
As a best practice, use the JVM class loader to find or load native libraries. WebSphere Application Server prints the Java library path (java.library.path) when starting up. If the JVM class loader is intended to load the library, verify that the path containing the native library file is in the Java library path. If not, append the path to the platform-specific native library environment variable or to the java.library.path system property of the server process definition.

In general, the Java virtual machine invokes findLibrary() on the class loader *xxx* that loads the class that calls System.loadLibrary(). If *xxx*.findLibrary() fails, the Java virtual

machine attempts to find the library using the JVM class loader, which searches the JVM library path. If the library cannot be found, the Java virtual machine creates an UnsatisfiedLinkError exception.

Thus, if a WebSphere class loader is intended to find a native library myNativeLib, the library must be visible on the `nativelibpath` of the class loader that loads the class that calls System.loadLibrary(myNativeLib). This practice is necessary or desirable in the following situation:

- Shared libraries have a **Native library path** in their configuration. Because shared libraries enable the versioning of application-specific libraries, consider specifying the paths to any native libraries used by the shared library code in the shared library configuration.

Ensure that the correct WebSphere class loader loads the class that calls System.loadLibrary() and that the native library is visible on the **Native library path** setting.

**The native library is already loaded.**

This condition can result from either of the following errors:

**User error**

Check for multiple calls to System.loadLibrary and remove any extraneous calls.

**Error when an application restarts**

The JVM has a restriction that only one class loader can load a native library at a time. An error results when an application restarts before the garbage collector cleans up the class loader from the stopped application. When the class that loads the native library moves, all of the classes that depend on that native library and their dependencies also must move.

To correct this condition, move the loading of the native library to a class loader that does not reload:

1. Locate all application classes that load native libraries or have native methods.
2. Identify any dependent classes for the classes in step 1, such as logging packages.
3. Create a server-associated shared library or an isolated shared library.
4. Move the JAR files loaded for classes in steps 1 and 2 from the application to the shared library created in step 3.
5. Save your changes.
6. Redeploy the application and rerun the scenario.

For more information about invoking, creating, and managing shared libraries, read “Managing shared libraries” in the *Administering applications and their environment* PDF book.

Classes within server-scoped libraries are loaded once for each server lifecycle, ensuring that the native library required by the application is loaded once for each Java virtual machine, regardless of the application’s life cycle.

**A dependent native library was used.**

Dependent native libraries must be found or loaded by the JVM class loader. That is, if a native library *NL* is dependent on another native library, *DNL*, the JVM class loader must find DNL on the Java library path. This is because the JVM runs native code when loading NL; when it encounters the dependency on DNL, the JVM native code can call only to the JVM class loader to resolve the dependency. A WebSphere class loader cannot load a dependent native library.

Modify the platform-specific environment variable defining the Java library path (LIBPATH) to include the path containing the unresolved native library.

## Class loader viewer service settings

Use this page to configure the server to start the class loader viewer service when the server starts. The Class Loader Viewer helps you diagnose problems with class loaders.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Class loader viewer service**.

Class loaders find and load class files. For a deployed application to run properly, the class loaders that affect the application and its modules must be configured so that the application can find the files and resources that it needs. Diagnosing problems with class loaders can be complicated and time-consuming. To diagnose and fix the problems more quickly, enable the class loader viewer service on this page and then use the console Class loader viewer to examine class loaders and the classes loaded by each class loader. Click **Troubleshooting** → **Class loader viewer** to access the Class loader viewer in the console.

### Enable service at server startup

Specifies whether or not the server attempts to start the class loader viewer service when the server starts.

The default is not to start the class loader viewer service.

## Enterprise application topology

Use this page to see where modules reside in a topology of enterprise applications. Knowing where a module resides helps you to determine which class loader loaded a module and to diagnose problems with class loaders.

To view this administrative console page, click **Troubleshooting** → **Class loader viewer**. This page lists all installed applications and their modules in a tree view. The modules can be Web modules (.war files) or enterprise bean (EJB) modules (.jar files).

When deploying an application to a server or starting an application, you might encounter problems related to class loaders. Use the console pages accessed from this page to troubleshoot errors such as the following:

- ClassCastException
- ClassNotFoundException
- NoClassDefFoundException
- UnsatisfiedLinkError

You can use the Class loader viewer console pages without having to restart or manipulate the application.

### Enterprise applications topology

Displays a tree hierarchy of applications installed on a server and lists the module files in the class paths of the applications.

Expand the hierarchy for an application to see what Web modules (.war files) and EJB modules (.jar files) are in the application class path.

Click on a module name to examine the class loaders of the module.

## Class loader viewer settings

Use this page to examine the class loaders visible to a Web module (.war file) or enterprise bean (.ejb file) in an installed enterprise application. This page helps you to determine which class loaders loaded files of a module and to diagnose problems with class loaders.

To view this administrative console page, click **Troubleshooting** → **Class loader viewer** → ***module_name***.

To learn more about classes used by the module and their class loaders, click a button:

| Button | Resulting action |
|---|---|
| **Export** | Opens a dialog that enables you to view or save the class loader information on this page in an XML file. |
| **Table View** | Displays the Table view page, which provides information about class loaders visible to the module in an HTML table format for each class loader. Such information includes:<br>**Delegation**<br>Whether the class loader delegates a load operation to its immediate parent before searching its local classpath for a class or resource<br>**Classpath**<br>The local classpath, which includes the paths over which the class loader searches for classes and resources, excluding the classpaths of any parent class loaders.<br>**Classes**<br>The names of classes loaded by the class loader |
| **Search** | Displays the Search page, on which you can search class loaders for the following:<br>• Specific strings<br>• Specific .jar files<br>• The names of files in a specific directory<br>• The names of files loaded by a specific class loader |

## Class Loader

Displays a hierarchy of class loaders that affect the loading of classes used by the Web or EJB module. The **Hierarchy** tab displays the class loaders in a tree hierarchy. The **Search Order** tabs lists the class loaders in the order in which the runtime environment uses them to find and load classes.

Expand a hierarchy of class loaders to view the following:

- Class loader names
- Arrows that point upwards beside class loader names, indicating that requests can go to a parent class loader only and not go to a child class loader
- The names of classes that are loaded by a class loader
- The paths of property files and .jar files used by the classes

The following class loaders might be in a hierarchy:

| Class loader name | Description |
|---|---|
| **JDK Extension Loader** | The JDK extensions class loader is a composite class loader that is comprised of the Java virtual machine (JVM) bootstrap class loader, the JVM extensions class loader and the JVM system class loader, which load the core SDK classes and resources as well as classes and resources visible on the JVM classpath. |
| **WAS Extension Class Loader** | The WAS Extension Class Loader loads the WebSphere Application Server classes, standalone resource classes, custom service classes, and custom registry classes. At bootstrap, this class loader uses the `ws.ext.dirs` system property to determine the path that is used to load classes. Each directory in the `ws.ext.dirs` class path and every .jar file or .zip file in these directories is added to the class path used by this class loader. |
| **WAS Compound Class Loader** | The WAS Compound Class Loaders load classes and resources of enterprise archive (EAR) modules, Web (WAR) modules, and server-associated shared libraries. Under default class loader policies, an instance of a WAS Compound Class Loader exists for each running EAR and WAR module and for each class loader defined in the server configuration. |

Click on **Classes** to view a list of classes loaded by a class loader.

The class loader viewer service must be enabled to view the list of classes.

## Search settings

Use this page to search for information about class loaders visible to a Web module (.war file) or enterprise bean (.ejb file) in an installed enterprise application. This page helps you diagnose problems with class loaders.

To view this administrative console page, click **Troubleshooting** → **Class loader viewer** → ***module_name*** → **Search**.

On the Search page, you can search class loaders for the following:
- Specific strings
- Specific .jar files
- The names of files in a specific directory
- The names of files loaded by a specific class loader

### Search type

Specifies the type of items in which to search for the string.

| Search type | Instructions and resulting action |
|---|---|
| **Class/Package** | In the **Search terms** field, type a class name or package name. After you select this search type and click **Go**, the program searches class loaders for a class or package name. The program displays a list of classes and packages that have the string in their name. |
| **JAR/Directory** | In the **Search terms** field, type a `.jar` file name or directory name. After you select this search type and click **Go**, the program searches class loaders for a .jar file or directory name. The program displays a list of .jar files that have the string in their name and of all files in directories that have the string in their name. |

### Search terms

Specifies the string to be found in the items searched.

The search is case-sensitive. If the search string is `classname`, the string *ClassName* is not found.

The search matches the entire string. If the search type is **JAR/Directory** and the search string is `C:/WebSphere/AppServerd0603.185/java/jre/lib/ext/CmpCrmf.jar`, the entire path of the JAR file is matched. If the search type is **JAR/Directory** and the search string is `Cmp`, the string `Cmp` is not found.

The search supports limited regular expressions. It supports the wildcard characters asterisk (`*`), question mark (`?`), and percent sign (`%`). The wildcard characters `*` and `%` match zero or more characters; `?` matches exactly one character.

| Search string | Resulting matches |
|---|---|
| `*Cmp*` | Items that have `Cmp` in their name |
| `*Cmp*.jar` | Items that have `Cmp` in their name and that end in `.jar` |
| `%Cmp%` | Items that have `Cmp` in their name |
| `%Cmp%.jar` | Items that have `Cmp` in their name and that end in `.jar` |
| `*Cmp?rmf.jar` | Items that have a name with any characters before `Cmp`, then any one character, and then `rmf.jar` |

The search supports full regular expressions if the value for the search string starts and ends with a forward slash (/).

| Search string | Resulting matches |
|---|---|
| /.*Cmp.*/ | Items that contain any character before and after Cmp in their name |
| /.*Cmp.*\.jar/ | Items that have Cmp in their name and that end in .jar |
| /.*Cmp?rmf\.jar/ | Items that have a name with any characters before Cmp, then any one character, and then rmf.jar |
| /.*\d\.jar/ | Items with a name that ends in a number followed by .jar |

# Diagnosing problems with message logs

WebSphere Application Server can write system messages to several general purpose logs, including JVM, process, and IBM service logs, which can be examined for problem determination.

## Before you begin

The JVM logs are created by redirecting the System.out and System.err streams of the JVM to independent log files. WebSphere Application Server writes formatted messages to the System.out stream. In addition, applications and other code can write to these streams using the print() and println() methods defined by the streams. Some Developer Kit built-ins such as the printStackTrace() method on the Throwable class can also write to these streams. Typically, the System.out log is used to monitor the health of the running application server. The System.out log can be used for problem determination, but it is recommended to use the IBM Service log and the advanced capabilities of the Log Analyzer instead. The System.err log contains exception stack trace information that is useful when performing problem analysis.

Because each application server represents a JVM, there is one set of JVM logs for each application server and all of its applications located by default in the following directory:

- *profile_root*/logs/*server_name*

The process logs are created by redirecting the STDOUT and STDERR streams of the process to independent log files. Native code, including the Java virtual machine (JVM) itself, writes to these files. As a general rule, WebSphere Application Server does not write to these files. However, these logs can contain information relating to problems in native code or diagnostic information written by the JVM.

As with JVM logs, there is a set of process logs for each application server, since each JVM is an operating system process.

The IBM service log contains both the WebSphere Application Server messages that are written to the System.out stream and some special messages that contain extended service information that is normally not of interest, but can be important when analyzing problems. There is one service log for all WebSphere Application Server JVMs on a node, including all application servers. The IBM Service log is maintained in a binary format and requires a special tool to view. This viewer, the Log and Trace Analyzer, provides additional diagnostic capabilities. In addition, the binary format provides capabilities that are utilized by IBM support organizations.

In addition to these general purpose logs, WebSphere Application Server contains other specialized logs that are specific to a particular component or activity. For example, the HTTP server plug-in maintains a special log. Normally, these logs are not of interest, but you might be instructed to examine one or more of these logs while performing specific problem determination procedures. For details on how and when to view the plug-in log, see the Accessing a Web resource through the application server and bypassing the HTTP server subsection of the A Web resource does not display topic.

### About this task

Sometimes server and application problems can be diagnosed by examining log output from the WebSphere Application Server.

Determine which type of logs you would like to implement:

- JVM logs
- Process logs
- IBM service logs

## Viewing JVM logs

The Java virtual machine (JVM) logs are written as plain text files.

### About this task

Use either of two techniques to view the JVM logs for an application server:

- Use the administrative console, which also supports viewing the JVM logs from a remote machine.
- Use a text editor on the machine where the logs are stored.

1. View the JVM logs from the administrative console.
   a. Start the administrative console.
   b. Click **Troubleshooting > Logs and Trace** in the console navigation tree. To view the logs for a particular server, click on the server name to select it, then click **JVM Logs**.
   c. Select the runtime tab.
   d. Click **View** corresponding to the log you want to view.
2. View the JVM logs from the machine where they are stored.
   a. Go to the machine where the logs are stored.
   b. Navigate to the *profile_root*/logs/*server_name* directory and select SystemOut.log or SystemErr.log.
   c. Open the file in a text editor or drag and drop the file into an editing and viewing program.

## JVM log interpretation

View the JVM log files to determine problems within application environments.

The JVM logs contain print data written by applications. The application can write this data directly in the form of `System.out.print()`, `System.err.print()`, or other method calls. The application can also write data indirectly by calling a JVM function, such as an `Exception.printStackTrace()`. In addition, the `System.out` JVM log contains system messages written by the WebSphere Application Server.

You can format application data to look like WebSphere Application Server system messages by using the Installed Application Output field of the JVM Logs properties panel, or as plain text with no additional formatting. WebSphere Application Server system messages are always formatted. Depending on how the JVM log is configured, formatted messages can be written to the JVM logs in either basic or advanced format.

### Message formats

Formatted messages are written to the JVM logs in one of two formats:

**Basic Format**
: The format used in earlier versions of WebSphere Application Server.

**Advanced Format**
: Extends the basic format by adding information about an event, when possible.

## Basic and advanced format fields

Basic and Advanced Formats use many of the same fields and formatting techniques. The various fields that may be found in these formats follow:

**TimeStamp**
The timestamp is formatted using the locale of the process where it is formatted. It includes a fully qualified date (for example YYMMDD), 24 hour time with millisecond precision and a time zone.

**ThreadId**
An 8 character hexadecimal value generated from the hash code of the thread that issued the message.

**ThreadName**
The name of the Java thread that issued the message or trace event.

**ShortName**
The abbreviated name of the logging component that issued the message or trace event. This is typically the class name for WebSphere Application Server internal components, but can be some other identifier for user applications.

**LongName**
The full name of the logging component that issued the message or trace event. This is typically the fully qualified class name for WebSphere Application Server internal components, but can be some other identifier for user applications.

**EventType**
A one character field that indicates the type of the message or trace event. Message types are in upper case. Possible values include:

- **F** A Fatal message.
- **E** An Error message.
- **W** A Warning message.
- **A** An Audit message.
- **I** An Informational message.
- **C** An Configuration message.
- **D** A Detail message.
- **O** A message that was written directly to `System.out` by the user application or internal components.
- **R** A message that was written directly to `System.err` by the user application or internal components.
- **Z** A placeholder to indicate the type was not recognized.

**ClassName**
The class that issued the message or trace event.

**MethodName**
The method that issued the message or trace event.

**Organization**
The organization that owns the application that issued the message or trace event.

**Product**
The product that issued the message or trace event.

**Component**
The component within the product that issued the message or trace event.

## Basic format

Message events displayed in basic format use the following format. The notation <name> indicates mandatory fields that will always appear in the basic format message. The notation [name] indicates optional or conditional fields that will be included if they can be determined.

```
<timestamp><threadId><shortName><eventType>[className][methodName]<message>
```

### Advanced format

Message events displayed in advanced format use the following format. The notation <name> is used to indicate mandatory fields that will always appear in the advanced format for message entries. The notation [name] is used to indicate optional or conditional fields that will be included if they can be determined.

```
<timestamp><threadId><eventType><UOW><source=longName>[className]
[methodName]<Organization><Product><Component>
[thread=threadName]<message>
```

## Configuring the JVM logs

Use the administrative console to configure the JVM logs for an application server.

### About this task

To log events or information from a running JVM, you can use the administrative console to configure the settings you need for each server. Configuration changes for the JVM logs that are made to a running application server are not applied until the application server is restarted.

1. Start the administrative console
2. Click **Troubleshooting > Logs and Trace**, then click ***server* > JVM Logs**.
3. Select the Configuration tab.
4. Scroll through the panel to display the attributes for the stream to configure.
5. Change the appropriate configuration attributes and click **Apply**.
6. Save your configuration changes.

### Java virtual machine (JVM) log settings

Use this page to view and modify the settings for the Java virtual machine (JVM) `System.out` and `System.err` logs.

To view this administrative console page, click **Troubleshooting > Logs and Trace >***server name* **> JVM Logs**.

View and modify the settings for the Java Virtual Machine (JVM) System.out and System.err logs for this managed process. The JVM logs are created by redirecting the System.out and System.err streams of the JVM to independent log files. The System.out log is used to monitor the health of the running application server. The System.err log contains exception stack trace information that is useful when performing problem analysis. There is one set of JVM logs for each application server and all of its applications. JVM logs are also created for the deployment manager and each node manager. Changes on the Configuration panel will apply when the server is restarted. Changes on the Runtime panel will apply immediately.

***File Name:***

Specifies the name of one of the log file described on this page.

The first file name field specifies the name of the `System.out` log. The second file name field specifies the name of the `System.err` file.

Press the **View** button on the Runtime tab to view the contents of a selected log file.

The file name specified for the `System.out` log or the `System.err` log must have one of the following values:

*filename*

The name of a file in the file system. It is recommended that you use a fully qualified file name. If the file name is not fully qualified, it is considered to be relative to the current working directory for

the server. Each stream must be configured with a dedicated file. For example, you cannot redirect both `System.out` and `System.err` to the same physical file.

If the directory containing the file already exists, the user ID under which the server is running requires read/write access to the directory. If the directory does not exist, it will be created with the proper permissions. The user id under which the server is running must have authority to create the directory.

**console**
This is a special file name used to redirect the stream to the corresponding process stream. If this value is specified for `System.out`, the file is redirected to `stdout`. If this value is specified for `System.err`, the file is redirected to `stderr`.

**none** Discards all data written to the stream. Specifying **none** is equivalent to redirecting the stream to dev/null on an operating system such as AIX® of Linux.

The default path for *filename* is the value of the variable SERVER_LOG_ROOT. To see the value of the SERVER_LOG_ROOT variable:

1. On the administrative console, select **Environment > WebSphere Variables**
2. Click on the **Server** radio button, and then click **Apply**. The value of the *SERVER_LOG_ROOT* variable appears in the resulting list.

To change the value of *SERVER_LOG_ROOT*:

1. Select SERVER_LOG_ROOT
2. Enter a new path in the Value field
3. Click Apply
4. Save the configuration. You will have to restart the server for the change to take effect.

You can also change the location and name of the ${SERVER_LOG_ROOT}/SystemOut.log and ${SERVER_LOG_ROOT}/SystemErr.log files to any other absolute path and filename (for example, /tmp/myLogfile.log).

***File formatting:***

Specifies the format to use in saving the `System.out` file.

***Log file rotation:*** Use this set of configuration attributes to configure the `System.out` or `System.err` log file to be self-managing.

A self-managing log file writes messages to a file until reaching either the time or size criterion. At the specified time or when the file reaches the specified size, logging temporarily suspends while the log file rolls over, which involves closing and renaming the saved file. The new saved file name is based on the original name of the file plus a timestamp qualifier that indicates when the renaming occurs. Once the renaming completes, a new, empty log file with the original name reopens and logging resumes. All messages remain after the log file rollover, although a single message can split across the saved and the current file.

You can only configure a log to be self-managing if the corresponding stream is redirected to a file.

**File Size**
Click this attribute for the log file to manage itself based on its file size. Automatic roll over occurs when the file reaches the specified size you specify in the maximum size field.

**Maximum Size**
Specify the maximum size of the file in megabytes. When the file reaches this size, it rolls over.

This attribute is only valid if you click File size.

**Time** Click this attribute for the log file to manage itself based on the time of day. At the time specified in the start time field, the file rolls over.

**Start Time**
Specify the hour of the day, from 1 to 24, when the periodic rollover algorithm starts for the first time after an Application Server restart. The algorithm loads at Application Server startup. Once started at the (start time field) hour, the rollover algorithm rolls the file every (repeat time field) hours. This rollover pattern continues without adjustment until the Application Server stops.

**Note:** The rollover always occurs at the beginning of the specified hour of the day. The first hour of the day, which starts at 00:00:00 (midnight), is hour 1 and the last hour of the day, which starts at 23:00:00, is hour 24. Therefore, if you want log files to roll over at midnight, set the start time to 1.

**Repeat time**
Specifies the number of hours after which the log file rolls over. Valid values range from 1 to 24.

Configure a log file to roll over by time, by size, or by time and size. Click **File Size** and **Time** to roll the file at the first matching criterion. For example, if the repeat time field is 5 hours and the maximum file size is 2 MB, the file rolls every 5 hours, unless it reaches 2 MB before the interval elapses. After the size rollover, the file continues to roll at each interval.

***Maximum Number of Historical Log Files:*** Specifies the number of historical (rolled) files to keep. The stream writes to the current file until it rolls. At rollover, the current file closes and is saved as a new name consisting of the current name plus the rollover timestamp. The stream then reopens a new file with the original name to continue writing. The number of historical files grows from zero to the value of the maximum number of historical files field. The next rollover deletes the oldest historical file.

***Installed Application Output:*** Specifies whether `System.out` or `System.err` print statements issued from application code are logged and formatted.

**Show application print statements**
Click this field to show messages that applications write to the stream using **print** and **println** stream methods. WebSphere Application Server system messages always appear.

**Format print statements**
Click this field to format application print statement like WebSphere Application Server system messages.

## Process logs

WebSphere Application Server processes contain two output streams that are accessible to native code running in the process. These streams are the stdout and stderr streams. Native code, including Java virtual machines (JVM), might write data to these process streams. In addition, JVM provided System.out and System.err streams can be configured to write their data to these streams also.

By default, the stdout and stderr streams are redirected to log files at application server startup, which contain text written to the stdout and stderr streams by native modules (*SRVPGMs, .dlls, .exes, UNIX® libraries, and other modules). By default, these files are stored as *profile_root*/logs/*server_name*/native_stderr.log and *profile_root*/logs/native_stdout.log.

## Configuring the service log

The settings for service logs are typically shared for all servers, but you can configure a separate service log for each server process by overriding the configuration values at the server level.

### About this task

The configuration values for the service log are inherited by each server process from the node configuration, but under certain circumstances you might wish to configure the service logs differently for individual servers. You can use the administrative console to change the service log settings from the server level configuration panels.

1. Start the administrative console.

2. Click **Troubleshooting > Logs and Trace > *server_name* > IBM Service Logs**.
3. Select the **Enable** box to enable the service log, clear the check box to disable the log.
4. Set the name for the service log.

   The default name is *profile_root*/logs/activity.log. If the name is changed, the run time requires write access to the new file, and the file must use the .log extension.
5. Set the maximum file size. Specifies the number of megabytes to which the file can grow. When the file reaches this size, it wraps, replacing the oldest data with the newest data.
6. Save the configuration.
7. Restart the server to apply the configuration changes.

### IBM service log settings

To view this administrative console page, click **Troubleshooting > Logs and Trace** > *server name* > **IBM Service Logs**.

Use this panel to configure the IBM service log, also known as the activity log. The IBM service log contains both the WebSphere Application Server messages that are written to the System.out stream and some special messages that contain extended service information that can be important when analyzing problems. There is one service log for all WebSphere Application Server Java virtual machines (JVMs) on a node, including all application servers. The IBM Service log is maintained in a binary format. Use the Log and Trace Analyzer or Showlog tool to view the IBM service log.

***Enable service log:***

Specifies creation of a log file by the IBM Service log.

***File Name:***

Specifies the name of the file used by the IBM Service log.

***Maximum File Size:***

Specifies the maximum size in megabytes of the service log file. The default value is 2 megabytes.

When this size is reached, the service log wraps in place. Note that the service log does not roll over to a new log file like the JVM logs.

***Enable Correlation ID:***

Specifies the generation of a correlation ID that is logged with each message.

You can use the correlation ID to correlate activity to a particular client request.

You can also use it to correlate activities on multiple application servers, if applicable.

## Viewing the service log

Service logs are logs written in a binary format. You cannot view a service log directly using a text editor. You should never directly edit the service log, as doing so will corrupt the log.

### Before you begin

To move a service log from one machine to another, you must use a mechanism like FTP, which supports binary file transfer. You can view a service log in two ways:

- It is recommended that you use the Log and Trace Analyzer tool to view the service log. This tool provides interactive viewing and analysis capability that is helpful in identifying problems.

- If you are unable to use the Log and Trace Analyzer tool, use the Showlog tool to convert the contents of the service log to a text format that you can then write to a file or dump to the command shell window.

### About this task

Run the showlog script to view the contents of the service log as described in the following procedure.

1. Open a shell window on the machine where the service log resides.
2. Change the directory to *app_server_root*/bin where *app_server_root* is the fully qualified path where the WebSphere Application Server product is installed.
3. Run the showlog script.

   ```
   showlog
   ```

4. Run the following showlog script with no parameters to display usage instructions.

   ```
   showlog
   ```

5. Format and write the service log contents to a file.

   ```
   showlog service_log_filename output_filename
   ```

   If the service log is not in the default location, you must fully qualify the *service_log_filename*

## CORBA minor codes

Applications that use CORBA services generate minor codes to indicate the underlying cause of a failure. These codes are written to the exception stack. Look for ″minor code″ in the exception stack to locate these exceptions.

### Overview

Common Object Request Broker Architecture (CORBA) is an industry-wide standard for object-oriented communication between processes, which is supported in several programming languages. Several subcomponents of the product use CORBA to communicate across processes.

When a CORBA process fails, that is a request from one process to another cannot be sent, completed, or returned, a high-level exception is created, such as TransactionRolledBackException: CORBA TRANSACTION_ROLLEDBACK.

### Minor codes that are used by product components

| Range | Related subcomponent | Where to find details |
|---|---|---|
| 0x49424300-0x494243FF | Security | Security components troubleshooting tips |
| 0x49421050-0x4942105F, 0x49421070-0x4942107F | ORB services | Object request broker troubleshooting tips |
| 0x4f4d and above | Standard CORBA exceptions | http://www.omg.org |
| 0x49421080-0x4942108F | Naming services | |

## Configuring the hang detection policy

The hang detection option for WebSphere Application Server is turned on by default. You can configure a hang detection policy to accommodate your applications and environment so that potential hangs can be reported, providing earlier detection of failing servers. When a hung thread is detected, WebSphere Application Server notifies you so that you can troubleshoot the problem.

## Before you begin

A common error in Java Platform, Enterprise Edition (Java EE) applications is a hung thread. A hung thread can result from a simple software defect (such as an infinite loop) or a more complex cause (for example, a resource deadlock). System resources, such as CPU time, might be consumed by this hung transaction when threads run unbounded code paths, such as when the code is running in an infinite loop. Alternately, a system can become unresponsive even though all resources are idle, as in a deadlock scenario. Unless an end user or a monitoring tool reports the problem, the system may remain in this degraded state indefinitely.

Using the hang detection policy, you can specify a time that is too long for a unit of work to complete. The thread monitor checks all managed threads in the system (for example, Web container threads and object request broker (ORB) threads) . Unmanaged threads, which are threads created by applications, are not monitored. For more information read “Hung threads in Java Platform, Enterprise Edition applications” on page 97.

## About this task

The thread hang detection option is enabled by default. To adjust the hang detection policy values, or to disable hang detection completely:

1. From the administrative console, click **Servers** > **Application Servers** > *server_name*
2. Under Server Infrastructure, click **Administration** > **Custom Properties**
3. Click **New**.
4. Add the following properties:

```
Name: com.ibm.websphere.threadmonitor.interval
Value:  The frequency (in seconds) at which managed threads in the selected
        application server will be interrogated.
Default: 180 seconds (three minutes).

Name: com.ibm.websphere.threadmonitor.threshold
Value:   The length of time (in seconds) in which a thread can be active
         before it is considered hung.  Any thread that is detected as
         active for longer than this length of time is reported as hung.
Default: The default value is 600 seconds (ten minutes).

Name:com.ibm.websphere.threadmonitor.false.alarm.threshold
Value: The number of times (T) that false alarms can occur
       before automatically increasing the threshold. It is possible that a
       thread that is reported as hung eventually completes its work,
       resulting in a false alarm.  A large number of these events indicates
       that the threshhold value is too small. The hang detection facility can
       automatically respond to this situation: For every T false alarms, the
       threshold T is increased by a factor of 1.5. Set the value to
       zero (or less) to disable the automatic adjustment.
Default: 100

Name: com.ibm.websphere.threadmonitor.dump.java
Value: Set to true to cause a javacore to be created when a hung thread
           is detected and a WSVR0605W message is printed.  The threads
           section of the javacore can be analyzed to determine what
           the reported thread and other related threads are doing.
Default: False
```

   To disable the hang detection option, set the **com.ibm.websphere.threadmonitor.interval** property to less than or equal to zero.
5. Click **Apply**.
6. Click **OK**.
7. Save the changes. Make sure a file synchronization is performed before restarting the servers.

8. Restart the Application Server for the changes to take effect.

## Hung threads in Java Platform, Enterprise Edition applications

WebSphere Application Server monitors thread activity and performs diagnostic actions if one has become inactive.

When WebSphere detects that a thread has been active longer than the time defined by the thread monitor threshold, the application server takes the following actions:

- Logs a warning in the WebSphere Application Server log that indicates the name of the thread that is hung and how long it has already been active. The following message is written to the log:

  ```
  WSVR0605W: Thread threadname has been active for
  hangtime and may be hung.  There are totalthreads
  threads in total in the server that may be hung.
  ```

  where: *threadname* is the name that appears in a JVM thread dump, *hangtime* gives an approximation of how long the thread has been active and *totalthreads* gives an overall assessment of the system threads.
- Issues a Java Management Extensions (JMX) notification. This notification enables third-party tools to catch the event and take appropriate action, such as triggering a JVM thread dump of the server, or issuing an electronic page or e-mail. The following JMX notification events are defined in the com.ibm.websphere.management.NotificationConstants class:
  - TYPE_THREAD_MONITOR_THREAD_HUNG This event is triggered by the detection of a (potentially) hung thread.
  - TYPE_THREAD_MONITOR_THREAD_CLEAR This event is triggered if a thread that was previously reported as hung completes its work. Consult the section on false alarms for more information.
- Triggers changes in the performance monitoring infrastructure (PMI) data counters. These PMI data counters are used by various tools, such as the Tivoli Performance Viewer, to provide a performance analysis.
- Triggers changes in the performance monitoring infrastructure (PMI) data counters. These PMI data counters are used by various tools, such as the Tivoli Performance Viewer (TPV), to provide a performance analysis.

For additional information about performance monitoring and Tivoli Performance Viewer, see the chapter Monitoring performance with Tivoli Performance Viewer (TPV) in the *Tuning guide* PDF book

### False Alarms

If the work actually completes, a second set of messages, notifications and PMI events is produced to identify the false alarm. The following message is written to the log:

```
WSVR0606W: Thread threadname was previously reported to be
hung but has completed. It was active for approximately hangtime.
There are totalthreads threads in total in the server that still
may be hung.
```

where *threadname* is the name that appears in a JVM thread dump, *hangtime* gives an approximation of how long the thread has been active and *totalthreads* gives an overall assessment of the system threads.

### Automatic adjustment of the hang time threshold

If the thread monitor determines that too many false alarms are issued (determined by the number of pairs of hang and clear messages), it can automatically adjust the threshold. When this adjustment occurs, the following message is written to the log:

```
WSVR0607W: Too many thread hangs have been falsely reported.  The hang
threshold is now being set to thresholdtime.
```

where: *thresholdtime* is the time (in seconds) in which a thread can be active before it is considered hung.

You can prevent WebSphere Application Server from automatically adjusting the hang time threshold. See "Configuring the hang detection policy" on page 95

## Example: Adjusting the thread monitor to affect server hang detection

The hang detection policy affects how the application server responds to a thread that is not being processed correctly.

You can adjust the thread monitor settings by using the wsadmin scripting interface. These changes take effect immediately, but do not persist to the server configuration, and are lost when the server is restarted. The following script provides an example of how to adjust the properties for the thread monitor using the wsadmin tool:

```
# Read in the interval, threshold, false alarm from the command line
set interval [lindex $argv 0]
set threshold [lindex $argv 1]
set adjustment [lindex $argv 2]

# Get the object name of the server you want to change the values on
set server [$AdminControl completeObjectName "type=Server,*"]

# Read in the interval and print to the console
set i [$AdminControl getAttribute $server threadMonitorInterval]

# Read in the threshold and print to the console
set t [$AdminControl getAttribute $server threadMonitorThreshold]

# Read in the false alarm adjustment threshold and print to the console
set a [$AdminControl getAttribute $server threadMonitorAdjustmentThreshold]

# Set the new values using the command line parameters
$AdminControl setAttribute $server threadMonitorInterval  ${interval}

$AdminControl setAttribute $server threadMonitorThreshold ${threshold}

$AdminControl setAttribute $server threadMonitorAdjustmentThreshold ${adjustment}
```

# Working with trace

Use trace to obtain detailed information about running the WebSphere Application Server components, including application servers, clients, and other processes in the environment.

## About this task

Trace files show the time and sequence of methods called by WebSphere Application Server base classes, and you can use these files to pinpoint the failure. Collecting a trace is often requested by IBM technical support personnel. If you are not familiar with the internal structure of WebSphere Application Server, the trace output might not be meaningful to you.

1. Configure an output destination to which trace data is sent.
2. Enable trace for the appropriate WebSphere Application Server or application components.
3. Run the application or operation to generate the trace data.
4. Analyze the trace data or forward it to the appropriate organization for analysis.

## Results

For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

## Enabling trace on client and stand-alone applications

When stand-alone client applications (such as Java applications which access enterprise beans hosted in WebSphere Application Server) have problems interacting with WebSphere Application Server, it might be useful to enable tracing for the application. Enabling trace for client programs will cause the WebSphere Application Server classes used by those applications, such as naming-service client classes, to generate trace information.

### About this task

A common troubleshooting technique is to enable tracing on both the application server and client applications, and match records according to timestamp to try to understand where a problem is occurring.

1. To enable trace for the WebSphere Application Server classes in a client application, add the system properties shown in the following example to the startup script or command of the client application. The location of the output and the classes and detail included in the trace follow the same rules as for adding trace to WebSphere Application Servers. For example, trace the stand-alone client application program named com.ibm.sample.MyClientProgram, enter the following command:

   ```
   java -DtraceSettingsFile=MyTraceSettings.properties
   -Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager
   -Djava.util.logging.configureByServer=true com.ibm.samples.MyClientProgram
   ```

   The file identified by *file name* must be a properties file placed in the class path of the application client or stand-alone process. You must create a trace properties file by copying the app_server_root/properties/TraceSettings.properties file to the same directory as your client application Java archive (JAR) file

   You cannot use the **-DtraceSettingsFile=TraceSettings.properties** property to enable tracing of the ORB component for thin clients. ORB tracing output for thin clients can be directed by setting **com.ibm.CORBA.Debug.Output = debugOutputFilename** parameter in the command line.

   The java.util.logging.manager and java.util.logging.configureByServer system properties configure Java logging to use a WebSphere Application Server-specific LogManager class and to use the configuration from the file specified by the traceSettingsFile property. The default Java Logging properties file, located in the Java SE Runtime Environment 6 (JRE6), will not be applied.
2. You can configure the MyTraceSettings.properties file to send trace output to a file using the traceFileName property. Specify one of two options:
   - The fully qualified name of an output file. For example, traceFileName=c:\\MyTraceFile.log. You must specify this property to generate visible output.
   - stdout. When specified, output is written to System.out.
3. You can also specify a trace string for writing messages with the Trace String property, Specify a startup trace specification similar to that available on the server. For your convenience, you can enter multiple individual trace strings into the trace settings file, one trace string per line.

### Results

Here are the results of using each optional property setting:

- Specify a valid setting for the traceFileName property without a trace string to write messages to the specified file or System.out only.
- Specify a trace string without a traceFileName property value to generate no output.
- Specify both a valid traceFileName property and a trace string to write both message and trace entries to the location specified in the traceFileName property.

## Tracing and logging configuration

Configure tracing and logging settings to help diagnose problems or evaluate system performance.

You can configure the application server to start in a trace-enabled state by setting the appropriate configuration properties. You can only enable trace for an application client or standalone process at process startup.

In WebSphere Application Server, V6 and later, a logging infrastructure, extending Java Logging, is used. This results in the following changes to the configuration of the logging infrastructure in WebSphere Application Server:

- Loggers defined in Java logging are equivalent to, and configured in the same way as, trace components introduced in previous versions of WebSphere Application Server. Both are referred to as ″components.″
- Both Java logging levels and WebSphere Application Server levels can be used. The following is a complete list of valid levels, ordered in ascending order of severity:

| **Trace option** | **Output file** |
|---|---|
| all | trace.log |
| finest or debug | trace.log |
| finer or entryExit | trace.log |
| fine or event | trace.log |
| detail | SystemOut.log |
| config | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |
| info | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |
| audit | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |
| warning | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |
| severe or error | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |
| fatal | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |
| off | trace.log and SystemOut.log (If tracing is not enabled, the output file is SystemOut.log) |

- Setting the logging and tracing level for a component to all will enable all the logging for that component. Setting the logging and tracing level for a component to off will disable all the logging for that component.
- You can only configure a component to one level. However, configuring a component to a certain level enables it to perform logging on the configured level and any higher severity level.
- Several levels have equivalent names: finest is equivalent to debug; finer is equivalent to entryExit; fine is equivalent to event; severe is equivalent to error.

Java Logging does not distinguish between tracing and message logging. However, previous versions of WebSphere Application Server have made a clear distinction between those kind of messages. In WebSphere Application Server, V6 and later, the differences between tracing and message logging are as follows:

- Tracing messages are messages with lower severity (for example, tracing messages are logged on levels fine, finer, finest, debug, entryExit, or event).
- Tracing messages are generally not localized.
- When tracing is enabled, a much higher volume of messages will be produced, and the trace output will be in the trace file, not the SystemOut/Err log files. The trace file will only appear if tracing is enabled.

- Tracing messages provide information for problem determination.

## Trace and logging strings

In WebSphere Application Server, V5.1.1 and earlier, trace strings were used for configuring tracing only. Starting in WebSphere Application Server, Version 6 and later, the ″trace string″ becomes a ″logging string″; it is used to configure both tracing and message logging.

In WebSphere Application Server, V5.1.1 and earlier, the trace service for all WebSphere Application Server components is disabled by default. To request a change to the current state of the trace service, a trace string is passed to the trace service. This trace string encodes the information detailing which level of trace to enable or disable and for which components.

In all versions of WebSphere Application Server, the tracing for all components is disabled by default. To change to the current state of the tracing and message logging, a logging string must be constructed and passed to the server. This logging string specifies what level of trace or logging to enable or disable for specific components.

You can type in trace strings (or logging strings), or construct them using the administrative console. Trace and logging strings must conform to a specific grammar.

For WebSphere Application Server, V5.1.1 and earlier, the specification of this grammar is as follows:

```
TRACESTRING=COMPONENT_TRACE_STRING[:COMPONENT_TRACE_STRING]*

 COMPONENT_TRACE_STRING=COMPONENT_NAME=LEVEL=STATE[,LEVEL=STATE]*

LEVEL = all | entryExit | debug | event

STATE = enabled | disabled

COMPONENT_NAME = COMPONENT | GROUP
```

For WebSphere Application Server, V6 and later, the previous grammar is supported. However a new grammar has been added to better represent the underlying infrastructure:

```
LOGGINGSTRING=COMPONENT_LOGGING_STRING[:COMPONENT_LOGGING_STRING]*

 COMPONENT_TRACE_STRING=COMPONENT_NAME=LEVEL

LEVEL = all | (finest | debug) | (finer | entryExit) | (fine | event )
| detail | config | info | audit | warning | (severe | error) | fatal | off

COMPONENT_NAME = COMPONENT | GROUP
```

The COMPONENT_NAME is the name of a component or group registered with the trace service logging infrastructure. Typically, WebSphere Application Server components register using a fully qualified Java class name, for example com.ibm.servlet.engine.ServletEngine. In addition, you can use a wildcard character of asterisk (*) to terminate a component name and indicate multiple classes or packages. For example, use a component name of com.ibm.servlet.* to specify all components whose names begin with com.ibm.servlet. Use a wildcard character of asterisk (*) at the end of the component or group name to make the logging string applicable to all components or groups whose names start with specified string. For example, a logging string specifying ″com.ibm.servlet.*″ as a component name will be applied to all components whose names begin with com.ibm.servlet. When an asterisk (*) is used by itself in place of the component name, the level the string specifies, will be applied to all components.

The following are examples of using an asterisk (*) in logging strings. Note that the asterisk (*) in the logging string does not need to have a period (.) in front of it. The period (.) can be used anywhere in the logging string.

- `com.ibm.ejs.ras.*=all` - enables tracing for all loggers with names starting with ″com.ibm.ejs.ras.″. If there is a logger named ″com.ibm.ejs.ras″ it will not have trace enabled.
- `com.ibm.ejs.ras*=all` - enables tracing for all loggers with names starting with ″com.ibm.ejs.ras″, such as com.ibm.ejs.ras, com.ibm.ejs.raslogger, com.ibm.ejs.ras.ManagerAdmin

**Note:**

- In WebSphere Application Server, V5.1.1 and earlier, you could set the level to ″all=disabled″ to disable tracing. This syntax, beginning with Version 6.0, will result in LEVEL=info; tracing will be disabled, but logging will be enabled.
- In WebSphere Application Server, V6 and later, ″info″ is the default level. If the specified component is not present (*=xxx is not found), *=info is always implied. Any component that is not matched by the trace string will have its level set to info.
- If the logging string does not start with a component logging string specifying a level for all components, using the ″*″ in place of component name, one will be added, setting the default level for all components.
- STATE = enabled | disabled is not needed in Version 6 and later. However, if used, it has the following effect:
  - ″enabled″ sets the logging for the component specified to the level specified
  - ″disabled″ sets the logging for the component specified to one level above the level specified. The following examples illustrate the effect that disabling has on the logging level:

| Logging string | Resulting logging level | Notes® |
|---|---|---|
| com.ibm.ejs.ras=debug=disabled | com.ibm.ejs.ras=finer | debug (version 5) = finest (version 6) |
| com.ibm.ejs.ras=all=disabled | com.ibm.ejs.ras=info | ″all=disabled″ will disable tracing; logging is still enabled. |
| com.ibm.ejs.ras=fatal=disabled | com.ibm.ejs.ras=off | |
| com.ibm.ejs.ras=off=disabled | com.ibm.ejs.ras=off | off is the highest severity |

## Proceed from broad to specific trace specifications in the trace string

**Note:** Start the trace string from the most broad component groups and then select more specific traces. The advantage to this approach is that the trace settings for classes or packages that are contained in a larger group are specified correctly by including them later in the trace string.

The logging string is processed from left to right. During the processing, part of the logging string might be modified or removed if the levels they configure are overridden by another part of the logging string.

Groups that contain packages that disable traces disable any packages that are enabled previously on the same line. For example:

```
*=off : MyGroup1=info : MyGroup2=finest : com.mycompany.mypackage.*=info  : com.mycompany.mypackage.MyClass=finest
```

This trace string indicates that the only tracing should come from the MyGroup1 group, the MyGroup2 group, and the com.mycompany.mypackage.* package with more specific tracing for MyClass class. If you reverse this string, all tracing is disabled.

## Examples

Examples of legal trace strings include:

| Version 5 syntax | Version 6 syntax |
|---|---|
| `com.ibm.ejs.ras.ManagerAdmin=debug=enabled` | `com.ibm.ejs.ras.ManagerAdmin=finest` |

| Version 5 syntax | Version 6 syntax |
| --- | --- |
| com.ibm.ejs.ras.ManagerAdmin=all=enabled,event=disabled | com.ibm.ejs.ras.ManagerAdmin=detail |
| com.ibm.ejs.ras.*=all=enabled | com.ibm.ejs.ras.*=all |
| com.ibm.ejs.ras.*=all=enabled:com.ibm.ws.ras=debug=enabled,entryexit=enabled | com.ibm.ejs.ras.*=all:com.ibm.ws.ras=finer |

# Enabling trace at server startup

Use the administrative console to enable tracing at a server's startup. You can use trace to assist you in monitoring system performance and diagnosing problems.

## About this task

The diagnostic trace configuration settings for a server process determines the initial trace state for a server process. The configuration settings are read at server startup and used to configure the trace service. You can also change many of the trace service properties or settings while the server process is running.

1. Start the administrative console.
2. Click **Troubleshooting > Logs and trace** in the console navigation tree, then click **Server > Diagnostic Trace**.
3. Click **Configuration**.
4. Do not select the **None** check box. If this option is selected, the trace data is not logged or recorded anywhere. All other handlers (including handlers registered by applications) still have an opportunity to process these traces.
5. Select whether to direct trace output to either a file or an in-memory circular buffer.

   **Note:** Different components can produce different amounts of trace output per entry. Naming and security tracing, for example, produces a much higher trace output than Web container tracing. Consider the type of data being collected when you configure your memory allocation and output settings.
6. If the in-memory circular buffer is selected for the trace output set the size of the buffer, specified in thousands of entries. This is the maximum number of entries that will be retained in the buffer at any given time.
7. If a file is selected for trace output, set the maximum size in megabytes to which the file should be allowed to grow. When the file reaches this size, the existing file will be closed, renamed, and a new file with the original name reopened. The new name of the file will be based upon the original name with a timestamp qualifier added to the name. In addition, specify the number of history files to keep.
8. Select the desired format for the generated trace.
9. Save the changed configuration.
10. To enter a trace string to set the trace specification to the desired state:
    a. Click **Troubleshooting > Logs and trace** in the console navigation tree.
    b. Select a server name.
    c. Click **Change Log Level Details**.
    d. If **All Components** has been enabled, you might want to turn it off, and then enable specific components.
    e. Click a component or group name. For more information see "Log level settings" on page 5 If the selected server is not running, you will not be able to see individual component in graphic mode.
    f. Enter a trace string in the trace string box.
    g. Select **Apply**, then **OK**.
11. Allow enough time for the nodes to synchronize, and then start the server.

## Enabling trace on a running server

Use the administrative console to enable tracing on a running server. You can use trace to assist you in monitoring system performance and diagnosing problems.

### About this task

You can modify the trace service state that determines which components are being actively traced for a running server by using the following procedure.

1. Start the administrative console.
2. Click **Troubleshooting > Logs and Trace** in the console navigation tree, then click **server > Diagnostic Trace**.
3. Select the **Runtime** tab.
4. Select the **Save runtime changes to configuration as well** check box if you want to write your changes back to the server configuration.
5. Change the existing trace state by changing the trace specification to the desired state.
6. Configure the trace output if a change from the existing one is desired.
7. Click **Apply**.

## Managing the application server trace service

You can manage the trace service for a server process while the server is stopped and while it is running. You can specify which components to trace, where to send trace output, the characteristics of the trace output device, and which format to generate trace output in.

### About this task

Modify the trace settings to help with diagnosing problems or tuning performance within certain applications. To manage the trace service for a server process:

1. Start the administrative console.
2. Click **Troubleshooting > Logs and Trace >** *server_name*.
3. Click the **Diagnostic Trace** link.
4. On the Configuration tab, do not select the **None** option. If this option is selected, the trace data is not logged or recorded anywhere. All other handlers (including handlers registered by applications) still have an opportunity to process these traces.
5. On the Runtime tab, select either the **Memory Buffer** or **File Trace Output** option.
6. Specify the appropriate values for your configuration for either the Memory Buffer or File Trace Output option.
7. Click **Apply**.
8. Click **Troubleshooting > Logs and Trace >** *server_name*.
9. Click the **Change Log Detail Levels** link.
10. Under Additional properties, click **Change Log Detail Levels**.
11. On the Runtime tab, change the existing trace state by changing the trace specification to the desired state.
12. Click **Apply**.

## Trace output

Trace output allows administrators to examine processes in the application server and diagnose various issues.

On an application server, trace output can be directed either to a file or to an in-memory circular buffer. If trace output is directed to the in-memory circular buffer, it must be dumped to a file before it can be viewed.

On an application client or stand-alone process, trace output can be directed either to a file or to the process console window.

In all cases, trace output is generated as plain text in either basic, advanced or log analyzer format as specified by the user. The basic and advanced formats for trace output are similar to the basic and advanced formats that are available for the JVM message logs.

## Basic and advanced format fields

Basic and Advanced Formats use many of the same fields and formatting techniques. The fields that can be used in these formats include:

**TimeStamp**
The timestamp is formatted using the locale of the process where it is formatted. It includes a fully qualified date (YYMMDD), 24 hour time with millisecond precision and the time zone.

**ThreadId**
An 8 character hexadecimal value generated from the hash code of the thread that issued the trace event.

**ThreadName**
The name of the Java thread that issued the message or trace event.

**ShortName**
The abbreviated name of the logging component that issued the trace event. This is typically the class name for WebSphere Application Server internal components, but may be some other identifier for user applications.

**LongName**
The full name of the logging component that issued the trace event. This is typically the fully qualified class name for WebSphere Application Server internal components, but may be some other identifier for user applications.

**EventType**
A one character field that indicates the type of the trace event. Trace types are in lower case. Possible values include:

- **>** a trace entry of type method entry.
- **<** a trace entry of type method exit.
- **1** a trace entry of type fine or event.
- **2** a trace entry of type finer.
- **3** a trace entry of type finest, debug or dump.
- **Z** a placeholder to indicate that the trace type was not recognized.

**ClassName**
The class that issued the message or trace event.

**MethodName**
The method that issued the message or trace event.

**Organization**
The organization that owns the application that issued the message or trace event.

**Product**
The product that issued the message or trace event.

**Component**
The component within the product that issued the message or trace event.

## Basic format

Trace events displayed in basic format use the following format:

```
<timestamp><threadId><shortName><eventType>[className][methodName]<textmessage>
               [parameter 1]
               [parameter 2]
```

### Advanced formats

Trace events displayed in advanced format use the following format:

```
<timestamp><threadId><eventType><UOW><source=longName>[className][methodName]
<Organization><Product><Component>[thread=threadName]
<textMessage>[parameter 1=parameterValue][parameter 2=parameterValue]
```

### Log analyzer trace format

Preserves trace information in the same format as produced by Showlog tool.

## Diagnostic trace service settings

To view this page, click the following path:

- **Troubleshooting > Logs and Trace > *server* > Diagnostic trace**

### Trace Output

Specifies where trace output should be written. The trace output can be written directly to an output file, or stored in memory and written to a file on demand using the Dump button found on the run-time page.

Different components can produce different amounts of trace output per entry. Naming and security tracing, for example, produces a much higher trace output than Web container tracing. Consider the type of data being collected when you configure your memory allocation and output settings.

#### None

If this option is selected, the trace data is not logged or recorded anywhere. All other handlers (including handlers registered by applications) still have an opportunity to process these traces.

**Memory Buffer** Specifies that the trace output should be written to an in-memory circular buffer. If you select this option you must specify the following parameters:

- **Maximum Buffer Size**
  - Specifies the number of entries, in thousands, that can be cached in the buffer. When this number is exceeded, older entries are overwritten by new entries.

#### File

Specifies to write the trace output to a self-managing log file. The self-managing log file writes messages to the file until the specified maximum file size is reached. When the file reaches the specified size, logging is temporarily suspended and the log file is closed and renamed. The new name is based on the original name of the file, plus a timestamp qualifier that indicates when the renaming occurred. Once the renaming is complete, a new, empty log file with the original name is reopened, and logging resumes. No messages are lost as a result of the rollover, although a single message may be split across the two files. If you select this option you must specify the following parameters:

- **Maximum File Size**
  - Specifies the maximum size, in megabytes, to which the output file is allowed to grow. This attribute is only valid if the File Size attribute is selected. When the file reaches this size, it is rolled over as described above.
- **Maximum Number of Historical Files**
  - Specifies the maximum number of rolled over files to keep.
- **File Name**

– Specifies the name of the file to which the trace output is written.

## Trace Output Format

Specifies the format of the trace output.

You can specify one of three levels for trace output:

- **Basic (Compatible)**
  – Preserves only basic trace information. Select this option to minimize the amount of space taken up by the trace output.
- **Advanced**
  – Preserves more specific trace information. Select this option to see detailed trace information for use in troubleshooting and problem determination.
- **Log analyzer trace format**
  – Preserves trace information in the same format as produced by Showlog tool.

## Runtime tab

### Save runtime changes to configuration

Save runtime changes made on the runtime tab to the trace configuration as well. Select this box to copy run-time trace changes to the trace configuration settings as well. Saving these changes to the trace configuration will cause the changes to persist even if the application is restarted.

## Trace Output

Specifies where trace output should be written. The trace output can be written directly to an output file, or stored in memory and written to a file on demand using the Dump button found on the run-time page.

### None

If this option is selected, the trace data is not logged or recorded anywhere. All other handlers (including handlers registered by applications) still have an opportunity to process these traces.

### Memory Buffer

Specifies that the trace output should be written to an in-memory circular buffer. If you select this option you must specify the following parameters:

- **Maximum Buffer Size**
  – Specifies the number of entries, in thousands, that can be cached in the buffer. When this number is exceeded, older entries are overwritten by new entries.
- **Dump File Name**
  – The name of the file to which the memory buffer will be written when it is dumped. This option is only available from the Runtime tab.

### File

Specifies to write the trace output to a self-managing log file. The self-managing log file writes messages to the file until the specified maximum file size is reached. When the file reaches the specified size, logging is temporarily suspended and the log file is closed and renamed. The new name is based on the original name of the file, plus a timestamp qualifier that indicates when the renaming occurred. Once the renaming is complete, a new, empty log file with the original name is reopened, and logging resumes. No messages are lost as a result of the rollover, although a single message may be split across the two files. If you select this option you must specify the following parameters:

- **Maximum File Size**

  - Specifies the maximum size, in megabytes, to which the output file is allowed to grow. This attribute is only valid if the File Size attribute is selected. When the file reaches this size, it is rolled over as described above.
- **Maximum Number of Historical Files**
  - Specifies the maximum number of rolled over files to keep.
- **File Name**
  - View the file that is specified by the **File Name** parameter. This does not apply your configuration.

# Select a server to configure logging and tracing

Use this page to select the server for which you want to configure logging and trace settings.

**Application Servers**

This page lists application servers in the cell and the nodes holding the application servers. The status indicates whether a server is running, stopped, or encountering problems.

When you select an application server, a panel is displayed that will allow you to choose which log or trace task to configure for that application server.

To view this administrative console page, click **Troubleshooting** > **Logs and Trace**

## Server

Specifies the logical name of the server.

## Host name

Specifies the name of the host for the application server.

## Version

Specifies the version for the application server.

## Type

Specifies the type of application server.

## Status

Indicates whether the application server is started or stopped. (Network Deployment only)

Note that if the status is *Unavailable*, the node agent is not running in that node, and you must restart the node agent before you can start the server.

# Log and trace settings

Use this page to view and configure logging and trace settings for the server.

To view this administrative console page, click:

- **Troubleshooting > Logs and Trace >** *server_name*

## Diagnostic Trace

The diagnostic trace configuration settings for a server process determine the initial trace state for a server process. The configuration settings are read at server startup and used to configure the trace service. You can also change many of the trace service properties or settings while the server process is running.

### Java virtual machine (JVM) Logs

The JVM logs are created by redirecting the System.out and System.err streams of the JVM to independent log files. WebSphere Application Server writes formatted messages to the System.out stream. In addition, applications and other code can write to these streams using the print() and println() methods defined by the streams.

### Process Logs

WebSphere Application Server processes contain two output streams that are accessible to native code running in the process. These streams are the stdout and stderr streams. Native code, including Java virtual machines (JVM), might write data to these process streams. In addition, JVM provided System.out and System.err streams can be configured to write their data to these streams also.

### IBM Service Logs

The IBM service log contains both the WebSphere Application Server messages that are written to the System.out stream and some special messages that contain extended service information that is normally not of interest, but can be important when analyzing problems. There is one service log for all WebSphere Application Server JVMs on a node, including all application servers. The IBM Service log is maintained in a binary format and requires a special tool to view. This viewer, the Log and Trace Analyzer, provides additional diagnostic capabilities. In addition, the binary format provides capabilities that are utilized by IBM support organizations.

### Change Log Level Details

Enter a log detail level that specifies the components, packages, or groups to trace. The log detail level string must conform to the specific grammar described in this topic. You can enter the log detail level string directly, or generate it using the graphical trace interface.

## Working with troubleshooting tools

WebSphere Application Server includes a number of troubleshooting tools that are designed to help you isolate the source of problems. Many of these tools are designed to generate information to be used by IBM Support, and their output might not be understandable by the customer.

### About this task

This section only discusses tools that are bundled with the WebSphere Application Server product. A wide range of tools which address a variety of problems is available from the WebSphere Application Server Technical Support Web site.

1. Select the appropriate tool for the task. For more information on the capacities of the supplied troubleshooting tools, see the relevant articles in this section.
2. Run the tool as described in the relevant article.
3. Contact IBM Support for assistance in deciphering the output of the tool. For current information available from IBM Support on known problems and their resolution, see the IBM Support page. IBM Support has documents that can save you time gathering information needed to resolve this problem. For the last minute updates, limitations, and known problems, see the Release notes. Before opening a PMR, see the Must gather page.
4. Use the IBM Support Assistant to help find and use various IBM Support resources, such as updated documentation and problem determination tools.

## Gathering information with the collector tool

The collector tool gathers information about your WebSphere Application Server installation and packages it in a Java archive (JAR) file that you can send to IBM Customer Support to assist in determining and analyzing your problem. Information in the JAR file includes logs, property files, configuration files, operating system and Java data, and the presence and level of each software prerequisite.

## Before you begin

The sort of information that you gather is not something that most people use. In fact, the collector tool packages its output into a JAR file. IBM includes the collector tool in the product code, along with other tools that help capture the information that you must provide when reporting a problem. The collector tool is part of a strategy of making problem reporting as easy and complete as possible.

There are two phases of using the collector tool. The first phase runs the collector tool on your WebSphere Application Server product and produces a Java archive (JAR) file. The IBM Support team performs the second phase, which is analyzing the Java archive (JAR) file that the collector program produces. The collector program runs to completion as it creates the JAR file, despite any errors that it might find like missing files or invalid commands. The collector tool collects as much data in the JAR file as possible.

The collector tool is a Java application that requires a Java SE Runtime Environment 6 (JRE6) to run.

## About this task

The tool is within the installation root directory for WebSphere Application Server. But you run the tool from a working directory that you create outside of the installation root directory. This procedure describes both of those steps and all of the other steps for using the tool and reporting the results from running the tool.

There are two ways to run the collector tool. Run the collector tool to collect summary data or to traverse the system to gather relevant files and command results. The collector tool produces a Java archive (JAR) file of information needed to determine and solve a problem. The collector summary option produces a lightweight collection of version and other information that is useful when first reporting the problem to IBM Support. Run the collector tool from the root user or from the administrator user to access system files that contain information about kernel settings, installed packages, and other vital data.

The tool collects information about the default profile if you do not use the optional parameter to identify another profile.

Run the collector tool.

1. Log on to the system with a user profile that has all object (*ALLOBJ) special authority.
2. Make a working directory where you can start the collector program.
3. Run the **STRQSH** command from the CL command line to prepare to run the collector program.
4. Make the working directory the current directory.
5. Run the following command from Qshell:

   `cd workingDirectory`

   The collector program writes its output JAR file to the current directory. The program also creates and deletes a number of temporary files in the current directory. Creating a work directory to run the collector program avoids naming collisions and makes cleanup easier. You cannot run the collector tool in a directory under the installation root directory for WebSphere Application Server.
6. Run the collector program by entering the fully qualified command from the command line of the working directory.
   - Run the following command from Qshell:

   `app_server_root/bin/collector`

   - Use the following command to gather data from a specific profile that might not be the default profile.
   - Run the following command from Qshell:

   `app_server_root/bin/collector -profileName profile_name`
7. Optional: You can also run the collector tool from the profile's root directory instead of the *app_server_root*/bin/ directory.

Run the following command from Qshell:

*profile_root*/bin/collector

You should get the same output if you run the collector tool from the bin directory of *profile_root* as you would running it from *app_server_root*.

Issuing the command from the profile also runs the setupCmdLine file in the profile's bin directory. This file sets an environment parameter that the collector uses to determine which profile's data to collect.

## Results

The collector program creates the Collector.log log file and an output JAR file in the current directory.

The name of the JAR file is composed of the host name, cell name, node name, and profile name:
*host_name-cell_name-node_name-profile_name*.JAR

The Collector.log log file is one of the files collected in the *host_name-cell_name-node_name-profile_name*.JAR file.

## What to do next

Send the *host_name-cell_name-node_name-profile_name*.JAR file to IBM Support for analysis.

## Collector tool output

Use the collector tool to gather and analyze output from WebSphere Application Server.

The first step in using the collector tool on your WebSphere Application Server product is to run the tool to produce a Java archive (JAR) file as output. The second step in using the collector tool is to analyze its output. The preferred method of performing this analysis is to send the JAR file to IBM Support for analysis. However, you can use this topic to understand the content of the JAR file if you perform your own analysis.

You can view the files contained in the JAR file without extracting the files from the JAR file. However, it is easier to extract all files and view the contents of each file individually. To extract the files, use one of the following commands:

- `jar -xvf WASenv.jar`
- `unzip WASenv.jar`

*Wasenv.jar* stands for the name of the JAR file that the collector tool creates.

The JAR file contains:

- A collector tool log file, `collector.log`
- Copies of stored WebSphere Application Server files and their full paths that are located under directory root in the JAR file
- Java information in a directory named `Java`
- A JAR file manifest

### Tips and suggestions

- Unzip the JAR file to an empty directory for easy access to the gathered files and for simplified cleanup.
- Check the `collector.log` file for errors:
  - Some errors might be normal or expected. For example, when the collector attempts to gather files or directories that do not exist for your specific installation, it logs an error about the missing files.
  - A non-zero return code means that a command that the collector tool attempted to run does not exist. This might be expected in some cases. If this type of error occurs repeatedly, there might actually be a problem.

### collector command - summary option

WebSphere Application Server products include an enhancement to the collector tool beginning with Version 5.0.2, known as the *collector summary option*.

The collector summary option helps you communicate with WebSphere Application Server technical staff at IBM Support. Run the collector tool with the -Summary option to produce a lightweight text file and console version of some of the information in the Java archive (JAR) file that the tool produces without the -Summary parameter. You can use the collector summary option to retrieve basic configuration and prerequisite software level information when starting a conversation with IBM Support.

The collector summary option produces version information for the WebSphere Application Server product and the operating system as well as other information. It stores the information in the Collector_Summary.txt file and writes it to the console. You can use the information to answer initial questions from IBM Support or you can send the Collector_Summary.txt file directly to IBM Support.

When running the collector tool on WebSphere Application Server for i5/OS®, an additional file named WAS_Collection_timestamp.html is created in the logs directory of the profile where the collector tool runs. If you do not specify the -profileName option, the profile is the default profile. The HTML file is sent to IBM Support when you initiate a PMR.

Run the **collector** command to create the JAR file if IBM Support needs more information to solve your problem.

To run the collector summary option, start from a temporary directory outside of the WebSphere Application Server product installation root directory and enter one of the following commands:

- *app_server_root*/bin/collector -Summary [-profileName profileName]

  The -profileName option specifies the profile to summarize.

## Configuring first failure data capture log file purges

The first failure data capture (FFDC) log file saves information that is generated from a processing failure. These files are deleted after a maximum number of days has passed. The captured data is saved in a log file for analyzing the problem.

### Before you begin

The first failure data capture (FFDC) feature preserves the information that is generated from a processing failure and returns control to the affected engines. The captured data is saved in a log file for analyzing the problem. FFDC is intended primarily for use by IBM Support. FFDC instantly collects events and errors that occur during the product runtime. The information is captured as it occurs and is written to a log file that can be analyzed by IBM Support personnel. The data is uniquely identified for the servant region that produced the exception.

The FFDC configuration properties files are located in the properties directory under the Application Server product installation. You must set the ExceptionFileMaximumAge property to the same value in all three files: ffdcRun.properties, ffdcStart.properties, and ffdcStop.properties. You can set the ExceptionFileMaximumAge property to configure the amount of days between purging the FFDC log files. The value of the ExceptionFileMaximumAge property must be a positive number. The FFDC feature does not affect the performance of the Application Server product.

### About this task

Perform the following steps to configure the number of days between the FFDC log file purges. The value is in days.

1. Open the ffdcRun.properties file.

The file is located in the *profile_root*/properties directory.

2. Change the value for the *ExceptionFileMaximumAge* property to the number of days between the FFDC log file purges. The value of the *ExceptionFileMaximumAge* property must be a positive number. The default is seven days. For example, ExceptionFileMaximumAge = 3 sets the default time to three days. The FFDC log file is purged after three days.
3. Save the ffdcRun.properties file and exit.
4. Repeat the previous steps to modify the ffdcStart.properties and ffdcStop.properties files.

### Results

The FFDC file management function removes the FFDC log files that have reached the maximum age and generates a message in the SystemOut.log file.

## Using IBM Support Assistant

IBM Support Assistant is a free troubleshooting application that helps you research, analyze, and resolve problems using various support features and tools. IBM Support Assistant enables you to find solutions yourself using the same troubleshooting techniques used by the IBM Support team, and it allows you to organize and transfer your troubleshooting efforts between members of your team or to IBM for further support.

### About this task

**Note:** IBM Support Assistant V4.0 is released with a host of new features and enhancements, making this version the most comprehensive and flexible yet. Our one-stop-shop solution to research, analyze and resolve software issues is now better than ever before, and you can still download it at no charge.

IBM Support Assistant version 4.0 enhancements include:

- **Remote System Troubleshooting:** Explore file systems, run automated data collectors and troubleshooting tools, and view the system inventory on remote systems.
- **Activity-based Workflow:** Choose from support-related activities, or use the Guided Troubleshooter for step-by-step help with analysis and resolution.
- **Case Management:** Organize your troubleshooting data in ″cases″; then export and share these cases with other problem analysts or with IBM Support.
- **Improved Flexibility:** Add your own search locations, control updates by hosting your own update site, get the latest product news and updates.

The IBM Support Assistant V4.0 consists of the following three distinct entities:

**IBM Support Assistant Workbench**

The IBM Support Assistant Workbench, or simply the Workbench, is the client-facing application that you can download and install on your workstation. It enables you to use all the troubleshooting features of the Support Assistant such as Search, Product Information, Data Collection, Managing Service Requests, and Guided Troubleshooting. However, the Workbench can only perform these functions locally, for example, on the system where it is installed (with the exception of the Portable Collector).

If you need to use the IBM Support Assistant features on remote systems, additionally install the Agent Manager and Agent. However, if your problem determination needs are purely on the local system, the Agent and Agent Manager are not required.

The Workbench has a separate download and this is all that is required to get started with the Support Assistant.

**IBM Support Assistant Agent**

The IBM Support Assistant Agent, or simply the Agent, is the piece of software that needs to be installed on EVERY system that you need to troubleshoot remotely. Once an Agent is installed on a system, it registers with the Agent Manager and you can use the Workbench to communicate with the Agent and use features such as remote system file transfer, data collections and inventory report generation on the remote machine.

**IBM Support Assistant Agent Manager**

The IBM Support Assistant Agent Manager, or simply the Agent Manager, needs to be installed only ONCE in your network. The Agent Manager provides a central location where information on all available Agents is stored and acts as the certificate authority. For the remote troubleshooting to work, all Agent and Workbench instances register with this Agent Manager. Any time a Support Assistant Workbench needs to perform remote functions, it authenticates with the Agent Manager and gets a list of the available Agents. After this, the Workbench can communicate directly with the Agents.

The Agent and Agent Manager can be downloaded in a combined installer, separate from the Workbench.

IBM Support Assistant Version 4 has the following functions:

**Search interface and access to the latest product information**

IBM Support Assistant allows you to search multiple knowledge repositories with one click and gives you quick access to the latest product information so that you spend less time looking for the solution and more time building skills and solving problems.

**Troubleshooting tools**

Whether you are new to an IBM product or an advanced user, IBM Support Assistant can help. You can choose to be guided through your problem symptoms or view a complete listing of advanced tooling for analyzing everything from logs to memory dumps.

**Access to local and remote systems**

Using the IBM Support Assistant Workbench installed on a local workstation running the Windows or Linux Intel operating system, you can connect to the IBM Support Assistant Agent installed on a remote system running on the AIX, Linux, Windows, or Solaris operating system through the IBM Support Assistant Agent Manager on the Workbench. This function enables you to explore, transfer data, and run diagnostic tooling not only on your system but on any other system where the IBM Support Assistant Agent is installed.

**Automated data gathering and efficient support**

Instead of manually gathering information, you can use IBM Support Assistant to run automated, symptom-specific data collectors. This data can then be attached to an IBM Service Request so that you can get support from the experts at IBM Support.

- Follow the installation instructions on IBM Support Assistant (ISA) Web site at: IBM Support Assistant (ISA).
- Read the “First Steps″ section of the documentation for IBM Support Assistant to run the customization wizard, or migrate from a previous version of IBM Support Assistant. Read the ″Tutorials″ section to learn more about the capabilities of ISA.

**Related information**

IBM Software support: IBM Support Assistant (ISA)

## Diagnosing problems using IBM Support Assistant tooling

The IBM Support Assistant (ISA) is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products.

## About this task

Tools for IBM Support Assistant perform numerous functions from memory-heap dump analysis and Java core-dump analysis to enabling remote assistance from IBM Support. All of these tools come with help and usage documentation that allow you to learn about the tools and start using them to analyze and resolve your problems.

The following are samples of the tools available in IBM Support Assistant:

**Memory Dump Diagnostic for Java (MDD4J)**
The Memory Dump Diagnostic for Java tool analyzes common formats of memory dumps (heap dumps) from the Java virtual machine (JVM) that is running the WebSphere Application Server or any other standalone Java applications. The analysis of memory dumps is targeted towards identifying data structures within the Java heap that might be root causes of memory leaks. The analysis also identifies major contributors to the Java heap footprint of the application and their ownership relationship. The tool is capable of analyzing very large memory dumps obtained from production-environment application servers encountering OutOfMemoryError issues.

**IBM Thread and Monitor Dump Analyzer (TMDA)**
IBM Thread and Monitor Dump Analyzer (TMDA) provides analysis for Java thread dumps or javacores such as those from WebSphere Application Server. You can analyze thread usage at several different levels, starting with a high-level graphical view and drilling down to a detailed tally of individual threads. If any deadlocks exist in the thread dump, TMDA detects and reports them.

**Log Analyzer**
Log Analyzer is a graphical user interface that provides a single point of contact for browsing, analyzing, and correlating logs produced by multiple products. In addition to importing log files from multiple products, Log Analyzer enables you to import and select symptom catalogs against which log files can be analyzed and correlated.

**IBM Visual Configuration Explorer**
The IBM Visual Configuration Explorer provides a way for you to visualize, explore, and analyze configuration information from diverse sources.

**IBM Pattern Modeling and Analysis Tool for Java Garbage Collector (PMAT)**
The IBM Pattern Modeling and Analysis Tool for Java Garbage Collector (PMAT) parses IBM verbose garbage-collection (GC) trace, analyzes Java heap usage, and recommends key configurations based on pattern modeling of Java heap usage. Only verbose GC traces that are generated from IBM Java Development Kits (JDKs) are supported.

**IBM Assist On-site**
IBM Assist On-site provides remote desktop capabilities. You run this tool when you are instructed to do so by IBM Support personnel. With this live remote-assistance tool, a member of the IBM Support team can view your desktop and share control of your mouse and keyboard to help you find a solution. The tool can speed up problem determination, data collection, and ultimately your problem solution.

To install tools for the IBM Support Assistant Workbench on a Windows or Linux Intel operating system, go to the **Update** menu and select **Tool Add-ons**. A list of all available tools appears, and you can select the tools that you would like to install.
You can install, update, or remove tools from the IBM Support Assistant Workbench at any time.

# Troubleshooting help from IBM

If you are not able to resolve a WebSphere Application Server problem by following the steps described in the Troubleshooting guide, by looking up error messages in the message reference, or looking for related documentation on the online help, contact IBM Technical Support.

Purchase of WebSphere Application Server entitles you to one year of telephone support under the Passport Advantage® program. For details on the Passport Advantage program, visit http://www.lotus.com/services/passport.nsf/WebDocs/Passport_Advantage_Home.

The number for Passport Advantage members to call for WebSphere Application Server support is 1-800-237-5511. Please have the following information available when you call:

- Your Contract or Passport Advantage number.
- Your WebSphere Application Server version and revision level, plus any installed fixes.
- Your operating system name and version.
- Your database type and version.
- Basic topology data: how many machines are running how many application servers, and so on.
- Any error or warning messages related to your problem.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

### IBM Support Assistant

IBM Support Assistant allows you to search multiple knowledge repositories and gives you access to the latest product information. You can choose to be guided through your problem symptoms or view a complete listing of advanced tooling for analyzing everything from logs to memory dumps. Using the IBM Support Assistant Workbench installed on a local workstation running the Windows or Linux Intel operating system, you can connect to the IBM Support Assistant Agent installed on a remote system running on the AIX, Linux, Windows, or Solaris operating system. You can use IBM Support Assistant to run automated, symptom-specific data collectors. This data can then be attached to an IBM Service Request so that you can get help from IBM Support.

### The Collector Tool

### Consulting

For complex issues such as integration with legacy systems, education, and help in getting started quickly with the WebSphere product family, consider using IBM consulting services. To learn about these services, browse the Web site http://www.ibm.com/services/fullservice.html.

## Diagnosing and fixing problems: Resources for learning

In addition to the information center, there are several Web-based resources for researching and resolving problems related to the WebSphere Application Server.

### The WebSphere Application Server support page

The official site for providing tools and sharing knowledge about WebSphere Application Server problems is the WebSphere Application Server support page: http://www.ibm.com/software/webservers/appserv/support.html. Among the features it provides are:

- A search field for searching the entire support site for documentation and fixes related to a specific exception, error message, or other problem. Use this search function before contacting IBM Support directly.
- *Solve a problem* links take you to specific problems and resolutions documented by WebSphere Application Server technical support personnel.
- The *Download* links provide free WebSphere Application Server maintenance upgrades and problem determination tools.

  The fix packs and refresh packs for the WebSphere Application Server for i5/OS products are also provided in the group PTF for the WebSphere Application Server product. The WebSphere Application

Server group PTF also includes other i5/OS group PTFs which are necessary for running your application servers. For more information see the PTFs page.
- *fixes* are software patches which address specific WebSphere Application Server defects. Selecting a specific defect from the list in the *Fixes by version* page takes you to a description of what problem the fix addresses.
- Fix packs are bundles of multiple fixes, tested together and released as a maintenance upgrade to WebSphere Application Server. Refresh packs are fix packs that also contain new function. If you select a fix pack from this page, you are taken to a page describing the target platform, WebSphere Application Server prerequisite level, and other related information. Selecting the *fix list* link on that page displays a list of the fixes which the fix pack includes. If you intend to install a fix which is part of a fix pack, it is usually better to upgrade to the complete fix pack rather than to just install the individual fix.

### Accessing WebSphere Application Server support page resources

Some resources on the WebSphere Application Server support page are marked with a key icon. To access these resources, you must supply a user ID and password, or register if do not already have an ID. When registering, you are asked for your contract number, which is supplied as part of a WebSphere Application Server purchase.

### WebSphere Developer Domain

The Developer Domains are IBM-supported sites for enabling developers to learn about IBM software products and how to use them. They contain resources such as articles, tutorials, and links to newsgroups and user groups. You can reach the WebSphere Developer Domain at http://www7b.software.ibm.com/wsdd/.

### The IBM Support page

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the Must gather documents for information to gather to send to IBM Support.

---

## Debugging Service details

Use this page to view and modify the settings used by the Debugging Service.

To view this administrative console page, click **Servers > Application Servers** > *server name* > **Debugging Service**.

The steps below describe how to enable a debug session on WebSphere Application Server. Debugging can prove useful when your program behaves differently on the application server than on your local system.

## Enable service at server startup

Specifies whether the server will attempt to start the Debug service when the server starts.

## JVM debug port

Specifies the port that the Java virtual machine will listen on for debug connections.

## JVM debug arguments

Specifies the debugging argument string used to start the JVM in debug mode.

## Debug class filters

Specifies an array of classes to ignore during debugging. When running in step-by-step mode, the debugger will not stop in classes that match a filter entry.

---

# Configuration problem settings

Use this page to identify and view problems that exist in the current configuration.

To view this administrative console page, click **Troubleshooting > Configuration Problems** in the console navigation tree.

To view a configuration problem, click **Configuration Validation** in the console navigation tree, then select the type of configuration you want to view.

## Configuration document validation

Use these fields to specify the level of validation to perform on configuration documents.

**Maximum**
Selecting **Maximum: Validate all documents** turns on validation for all documents, regardless of whether or not they are extracted, and regardless of the relationships between the documents.

**High** Selecting **High: Validate extracted, parent, and sibling documents** turns on validation for extracted documents and their parent documents, and turns on validation for the sibling documents of the documents which have been extracted. For example, if **High** validation is selected, and if the `server.xml` document is extracted, when performing validation, validation is performed on the three documents: `server.xml`, `node.xml`, and `cell.xml`. and on the two sibling documents `variables.xml` and `resources.xml` within the server1 directory.

**Medium**
Selecting **Medium: Validate extracted and parent documents** turns on validation for the documents which have been extracted by the user interface, and also turns on validation of the parent documents of the documents which have been extracted. For example, using the partial directory structure from above, if **Medium** validation is selected, and if the `server.xml` document is extracted, when performing validation, validation is performed on all three of the documents `server.xml`, `node.xml`, and `cell.xml`.

**Low** Selecting **Low: Validate extracted documents** turns on validation for just those documents which have been extracted by the user interface.

**None** Selecting **None** disables validation. No configuration documents are validated.

## Enable Cross validation

Enables cross validation of configuration documents. Enabling cross validation enables comparison of configuration documents for conflicting settings.

## Configuration Problems

Displays current configuration problem error messages. Click a message for detailed information about the problem.

## Scope

Sorts the configuration problem list by the configuration file where each error occurs. Click a message for detailed information about the problem.

## Message

Displays the message returned from the validator.

## Explanation
A brief explanation of the problem.

## User action
Specifies the recommended action to correct the problem.

## Target Object
Identifies the configuration object where the validation error occurred.

## Severity
Indicates the severity of the configuration error. There are three possible values for severity.

**Error** This means that there is a problem with the configuration that might cause partial or complete failure of server function. This is the most severe warning.

**Warning**
This means that there is a problem with the configuration that might cause a failure of server function, or that might cause the server to function in an unexpected manner.

**Information**
A setting of the configuration that is unexpected and noteworthy, which requires customer notification. Information is used when the configuration has a value which is probably okay, but should be double checked by the administrator. This is the least crucial level of severity.

## Local URI
Specifies the local URI of the configuration file where the error occurred.

## Full URI
Specifies the full URI of the configuration file where the error occurred.

## Validator classname
The classname of the validator reporting the problem.

## Runtime events
Use the Runtime event pages of the administrative console to view the events published by application server classes.

To view these administrative console pages, click **Troubleshooting**. Expand **Runtime Messages** and click either **Runtime Error**, **Runtime Warning**, or **Runtime Information**.

Separate pages show error events, warning events, and informational events. Each page displays events in the same format.

You can adjust the number of messages that appear on the page in the **Preferences** settings.

Click a message to view event details.

### Timestamp
When the event occurred.

### Message originator
Internal application server class that published the event.

### Message
Identifier and short description of the event.

## Message details
Use the Message Details panel of the administrative console to view detailed information about errors, warnings, and informational messages.

To view these administrative console pages, click **Troubleshooting**. Expand **Runtime Messages** and click either **Runtime Error**, **Runtime Warning**, or **Runtime Information**. Click a message to display this panel.

Each message has the following general property fields.

### Message
The message ID and text.

### Message type
Error, Warning, or Information.

### Explanation
A description of the message.

### User action
What you should do about the message.

### Message originator
The name of the product class that originated the message.

### Source object type
The name of the component that originated the message.

### Timestamp
The date and time that the message originated.

### Thread ID
The thread identifier.

### Node name
The name of the node of the application server that originated the message.

### Server name
The name of the application server process that originated the message.

### Diagnostic Provider ID
The Diagnostic Provider ID of the component that originated the message. Click on Configuration Data, State Data, or Tests to run the corresponding diagnostic action against the originating component. A Diagnostic Provider ID will not be supplied with all messages.

---

## Showlog commands for Common Base Events

The showlog command converts the service log from binary format into plain text.

### Purpose

These showlog commands to produce output in Common Base Event XML format.

- `showlog -format CBE-XML-1.0.1` *filename*

where:

*filename*
Is the service log file name.

For examples of showlog scripts, see Showlog Script.

## Working with Diagnostic Providers

Diagnostic Providers enable you to query the startup configuration, current configuration, and current state of a diagnostic domain. In addition, Diagnostic Providers can also provide access to any self diagnostic tests that are available from a diagnostic domain.

### About this task

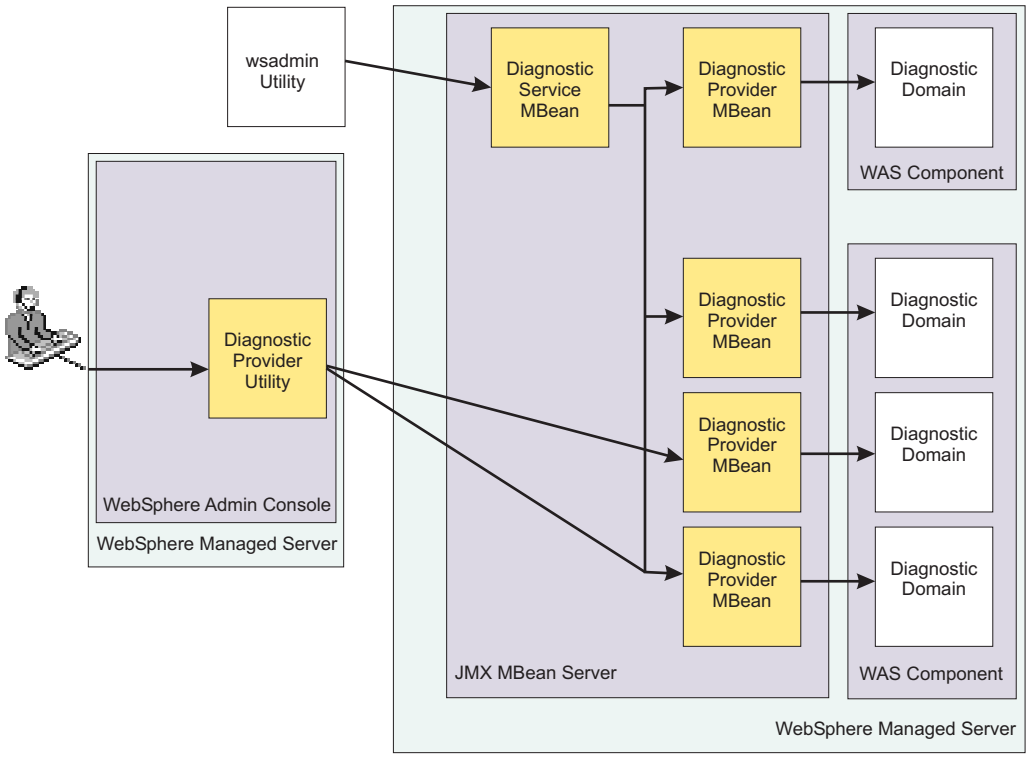
The Diagnostic Provider Utility is a simple front end in the administration console that presents the available set of Diagnostic Providers and enables you to work with them.

Learn about Diagnostic Providers

## Diagnostic Providers

Diagnostic Providers are a quick method for viewing configuration and the current state of individual components within an application server environment.

WebSphere Application Server components can be considered as being divisible into *diagnostic domains*. A diagnostic domain refers to a set of classes within the component that share a set of diagnostics. Some larger components might have multiple diagnostic domains. For example, the Connection Manager logically consists of multiple data sources and connection factories that each have separate diagnostic domains.




This image shows the relationships between the parts that make up the Diagnostic Provider (DP) utility.

**Diagnostic Provider MBeans**

A single diagnostic domain receives its diagnostic services from a Diagnostic Provider MBean. The Diagnostic Provider MBean enables you to query the startup configuration, current configuration, and current state of the diagnostic domain. In addition, Diagnostic Provider MBeans can also provide access to any self diagnostic tests that are available from the diagnostic domain. Some characteristics of Diagnostic Provider MBeans include:

- Diagnostic Provider MBeans are Java Management Extensions (JMX) MBeans
- Diagnostic Provider MBeans all implement a DiagnosticProvider interface which includes methods for configuration dumps, state dumps, and self diagnostic tests
- Diagnostic Provider MBeans provide a way to expose information about running components so administrators can more easily debug problems related to those components. As with other MBeans running in WebSphere Application Server, they can be accessed from JMX client code, or through the *wsadmin* tool.

**Diagnostic Provider Infrastructure**

Diagnostic Provider MBeans are efficient at delivering Java object representations of configuration, state, and self test information. This is good for when programs interact. For human users to access the information, WebSphere Application Server provides a set of facilities to extend the value of Diagnostic Provider MBeans.

**The Diagnostic Service MBean**
: provides methods to convert Diagnostic Provider MBean output into human readable formats. The Diagnostic Service MBean also provides some methods to facilitate looking up the Diagnostic Provider MBeans on the same server as the Diagnostic Service MBean. For administrators that want to access diagnostic data from a command line, the wsadmin tool can be used directly with the Diagnostic Service MBean to get formatted results

**The Diagnostic Provider utility**
: a set of panels included in the WebSphere Application Server administration console through which administrators can interact with Diagnostic Provider MBeans. The Diagnostic Provider utility is a simple front end in the administration console that presents the available set of Diagnostic Provider MBeans present on each managed server, and provides a means to execute and view the results of configuration dumps, state dumps, and diagnostic self tests.

.

## The purpose of Diagnostic Providers

Diagnostic Providers give you more information for quickly discovering and diagnosing system problems. The following scenario contrasts the experience of an administrator working with a component that does not have a Diagnostic Provider to one that does.

When the administrator works with a component that is without a Diagnostic Provider, the events are as follows:

1. A log entry indicates that a particular component is experiencing a problem.
2. The system administrator sees the log entry through the runtime messages panel.
3. The system administrator cannot tell what is wrong, so calls IBM support for assistance, with a potentially ill-defined problem.

When the administrator works with a component with a Diagnostic Provider, and the Diagnostic Provider ID is registered with the component's logger, the situation changes as follows:

1. A log entry that contains a Diagnostic Provider ID (DPID) indicates that something has gone wrong in a specific component.
2. The system administrator sees the log entry through the runtime messages panel.

3. The administrator clicks a button on the runtime message panel to execute a state dump or a configuration dump, or to be taken to the list of component self tests.
4. From the self test, the administrator is warned that the component is configured in a way that could lead to poor performance or failures.

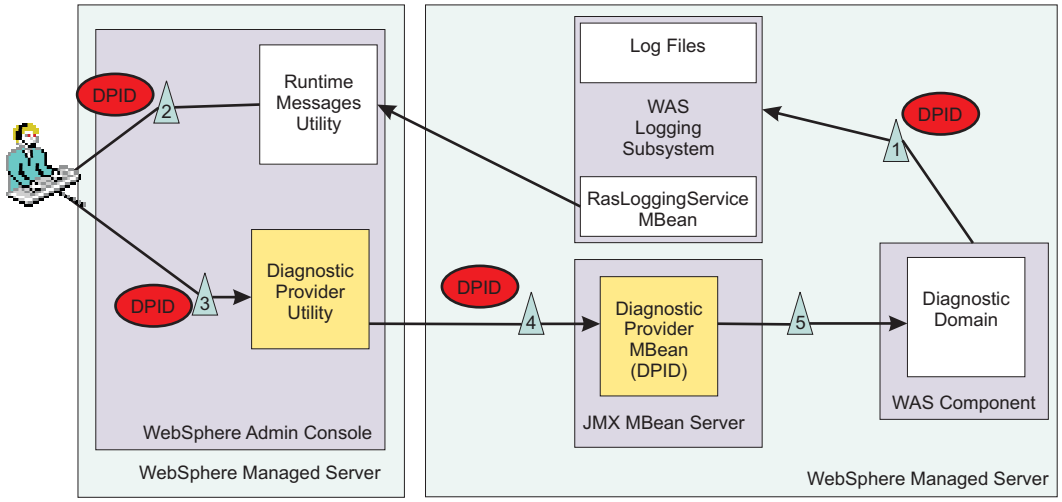
Furthermore, when the administrator works with a component with a Diagnostic Provider, and the Diagnostic Provider ID is **not** registered with the component's logger, the situation might unfold like this:
1. A log entry which doesn't contain a DPID indicates that something has gone wrong in a component.
2. The system administrator sees the log entry through the runtime messages panel.
3. The system administrator uses the administrative console to navigate through the available set of Diagnostic Providers and selects one that sounds appropriate.
4. He runs a configuration dump, a state dump, or a self diagnostic test against the Diagnostic Provider to collect information about the component.
5. From the state dump, the administrator is able to notice that the component state is not what would be expected for its workload.
6. The administrator works with the test team to determine which of the flows is causing the state of the component to diverge from what is expected (as evidenced by repeated execution of the state dump).

## Diagnostic Provider IDs

A Diagnostic Provider ID (DPID) is the unique address of a Diagnostic Provider MBean. Components that have associated Diagnostic Provider MBeans can include the DPID in their log entries.

Diagnostic Provider IDs are implemented in WebSphere Application Server as Java Management Extensions (JMX) *MBean ObjectNames*, and can be used at JMX MBean servers to look up Diagnostic Provider MBeans.

By including the String representation of the DPID in each logged message, the message can be tracked back to the Diagnostic Provider related to the component. A method is provided to associate Diagnostic Provider IDs with Loggers (from the *java.util.logging* logging API).




The diagram above shows how the use of DPIDs in log entries enables callbacks to the component that originally created the log entry.
1. Shows the component logging with a DPID included in the log entry.
2. The administrator examines the log entry through the Runtime Messages Utility and notices that the entry has a link to a Diagnostic Provider.
3. The administrator uses the link to gain access to the relevant MBean in the Diagnostic Provider Utility .
4. The Diagnostic Provider Utility contacts the Diagnostic Provider MBean to ask for more information.
5. The request for more information is sent back to the source of the original log entry.
6. The response from the Diagnostic Provider is provided in the administration console.

## Diagnostic Provider configuration dumps, state dumps, and self tests

The Diagnostic Provider (DP) infrastructure allows for a software component or stack product in the WebSphere Application Server space to expose key information about its configuration, current state, and current ability to perform operations.

The methods that expose this information might be driven as a result of a message put out by the component (by a logger which automatically includes the Diagnostic Provider ID in each message), or might be driven as a result of an overall system health-check when an administrator or automated tool is monitoring the system.

### Configuration dumps

A *Configuration dump* is an operation you can perform on a Diagnostic Provider to list the startup or current values of the configuration attributes for the DP. The name for each data item in this dump should reflect its disposition. That is, each item should be called *startup-xxx* or *current-xxx* to show whether this is a startup or current value. The collection of attributes returned from this operation can be thought of as the **payload** of the configuration dump. More information about payloads can be found in “Diagnostic Provider method implementation” on page 130.

You can find several ways to filter the output of a configuration dump in “Diagnostic Provider registered attributes and registered tests.”

### State dumps

A *State dump* is similar to a configuration dump, but it differs in two key areas. First, a state dump displays current information about the operation of a component. An example is a connection pool. A configuration dump can show *DataSource name*, the *minConnections* (configured or current), the *maxConnections*, the *DataBase name*, and so on. A state dump is more likely to show the current connections in use, the high concurrent use count, the number of times the pool has been expanded, the average time between requesting a connection and returning it, and so forth.

State dumps can be impacted by the values in the State Collection Specification. This is a dynamic specification that controls additional data collection that the component can do at runtime. If additional data is being collected, then a State dump might display more information. The same filters and payload information that apply to Configuration dumps (see “Diagnostic Provider registered attributes and registered tests”) apply to State dumps.

### Self Diagnostic tests

Self diagnostic tests are non-invasive operations that a Diagnostic Provider exposes. *Non-invasive* means that if they modify anything for the test, the conclusion of the test reverses the modification. These tests give an administrator the option to test simple functions of a component to see if it is able to perform them.

The filters for a self diagnostic test apply to the test itself, not to the output of the test. A typical use of Self Diagnostic tests could be for a pool manager of some sort to pull an object out of the pool and return it to the pool to verify that this operation can still be performed, and with acceptable performance.

## Diagnostic Provider registered attributes and registered tests

Each Diagnostic Provider (DP) provides a list of state dump attributes, configuration dump attributes, self tests, and self test attributes. The tests are operations that the DP can perform. The attributes are pieces of information that are available for collection from a Configuration dump, a State dump, or a specific Self Diagnostic test.

Each attribute can be seen as a piece of information with a label on it. Each attribute is also considered to be either *registered* or *not registered*. A registered attribute is one that should be available from one

release of WebSphere Application Server to the next. A nonregistered attribute might not be available in its current form in future releases of the product (no commitment has been made).

When performing a Configuration dump, a State dump, or a Self Diagnostic test, an administrator or automatic tooling can request *only registered* values, or *all* values, depending on the needs of the administrator or tool. Note that the option of filtering results is only available through the Diagnostic Provider's Java Management Extension (JMX) MBean interface, which you can access programmatically or through the wsadmin tool.

### The DiagnosticProviderRegistration XML file

The DiagnosticProviderRegistration Extensible Markup Language (XML) file is used in conjunction with the method signatures to filter the results of calling the various methods. This XML file defines the configuration information, state information, and self diagnostic tests exposed by the component. In the configuration and state information, the key working unit is referred to as the attribute. Specification of an attribute is as follows:

```
<attribute>
        <id><Regular Expression representing the attribute name></id>
        <descriptionKey><MsgKey into a ResourceBundle for localization of the label></descriptionKey>
        <registered>true</registered>
   </attribute>
```

The parts are as follows:

**ID:** The attribute's name. This name can be expressed with wildcard characters conforming to regular expression syntax. The registered attribute ID is used in the following places:

- Within Diagnostic Provider configuration dump and state dump methods to determine which attributes to return.
- In the administration console to match description keys to attributes returned from a request to a Diagnostic Provider for a configuration dump, state dump, or self diagnostic operation.

As an example, if a configuration dump returns an attribute with ID **cachedServlet-MyServlet-servletPath** to the administration console, the administration console could use the *descriptionKey* corresponding to the attribute registered as **<id>cachedServlet-.*-servletPath</id>** when selecting what description text to put next to this attribute's name and return values.

**descriptionKey:**
This is a key into a *resourceBundle* for localization.

**registered:**
This is a boolean qualifying whether this attribute will be available from one release of the software to the next. If registered is **true**, then this attribute should be available in the next release. If registered is **false**, then there is no guarantee that this attribute will continue to exist. Automation should use some caution when handling non-registered attributes.

Specification of a selfDiagnosticTest is as follows:

```
<test>
       <id><Regular Expression for the name of the test></id>
       <descriptionKey><MsgKey into a ResourceBundle for localization of the label></descriptionKey>
       <attribute><One or more attributes which will be output from this test></attribute>
</test>
```

The parts are as follows:

**ID:** Similar to the ID for the attribute, but in this case, describing the test to be performed instead of the attribute to be returned.

**descriptionKey:**
This is a key into a resourceBundle for localization.

### Method interfaces

```
public DiagnosticEvent [] configDump(String aAttributeId, boolean aRegisteredOnly);
public DiagnosticEvent [] stateDump(String aAttributeId, boolean aRegisteredOnly);
```

These methods invoke the configuration or state dump on the component, and specify a regular expression filter for the attributes to return as well as filtering the output to include all matching attributes, or only those attributes which are registered. This enables the administrator or automated software driving the method to specify a subset of the overall fields (especially important if many attributes are exposed or if the State Collection Specification increases the amount of data available). The following helper methods are available to assist with filtering the output.

To take a list of Attributes that are available to return, and filter them:

```
public static AttributeInfo [] queryMatchingDPInfoAttributes(String aAttributeId,
    AttributeInfo [] inAttrs, String [] namesToCheck, boolean aRegisteredOnly) {
```

To take a single Attribute that is available to return, and filter it:

```
public static AttributeInfo queryMatchingDPInfoAttributes(String aAttributeId,
    AttributeInfo [] inAttrs, String nameToCheck, boolean aRegisteredOnly) {
```

To go through a populated set of Attribute Information and remove unneeded parts:

```
public static void filterEventPayload(String aAttributeId, HashMap payLoad) {
```

For details on these messages, please review the API documentation for the **DiagnosticProviderHelper** class. The basic concept is that, once the component knows what attributes are able to be returned, the helper method will determine which of them should be returned based on the regular expression logic and registration boolean.

The selfDiagnostic Method interface here is similar to that of Configdump and Statedump:

```
public DiagnosticEvent[] selfDiagnostic(String aTestId, boolean aRegisteredOnly)
```

The difference is that the first parameter is a regular expression filter for which test to run.

## Diagnostic Provider names

In addition to the Diagnostic Provider ID (DPID), each component that implements the Diagnostic Provider interface must have a Diagnostic Provider Name. While the DPID must be unique within the entire WebSphere Application Server domain, the Diagnostic Provider Name need only be unique within the Java Virtual Machine (JVM).

Unlike the Diagnostic Provider ID, which tends to be long and not human-friendly, the Diagnostic Provider Name should be shorter and easier to read. In addition, by convention it should end in *DP*. The Diagnostic Service MBean (see “The simpler interfaces provided by the Diagnostic Service MBean”) can drive methods on a Diagnostic Provider using its name.

## The simpler interfaces provided by the Diagnostic Service MBean

All services for a Diagnostic Provider (DP) are also available through a Java Management Extensions (JMX) interface known as the *Diagnostic Service* interface. The Diagnostic Service interface enables administrators to drive methods against DPs using the Diagnostic Provider Name or Diagnostic Provider ID.

When formatted output is requested of the Diagnostic Service, it is localized to the client Locale. This makes the Diagnostic Service MBean ideal for clients using an interface where consuming complex Java objects, such as those returned from the Diagnostic Provider MBeans, is not feasible. An example of such an interface is the wsadmin tool.

The Diagnostic Service interface provides four signatures for each of the key methods available on the Diagnostic Providers (*configDump*, *stateDump*, and *selfDiagnostic*) objects. Because these method

signatures look so similar, this example shows it all through the configDump methods. The four Diagnostic Service methods that map to configDump on a Diagnostic Provider are:

```
public DiagnosticEvent [] configDump(String aDPName, String aAttributeIdSpec, boolean aRegisteredOnly)
public DiagnosticEvent [] configDumpById(String aDPid, String aAttributeIdSpec, boolean aRegisteredOnly)
public String [] configDumpFormatted(String aDPName, String aAttributeIdSpec,
       boolean aRegisteredOnly, Locale aLocale)
public String [] configDumpFormattedById(String aDPid, String aAttributeIdSpec,
       boolean aRegisteredOnly, Locale aLocale) {
```

The first two return exactly what the Diagnostic Provider does. The second two methods act as a pass-through to the actual Diagnostic Provider, but they take the array of Diagnostic Events that the Diagnostic Provider returns, and convert it into a more easily consumable *String* array. In addition, these methods handle localizing the output to the appropriate locale. It is important to note that the same method can be driven using the Diagnostic Provider ID or the Diagnostic Provider Name.

**Related tasks**

"Working with Diagnostic Providers" on page 121
Diagnostic Providers enable you to query the startup configuration, current configuration, and current state of a diagnostic domain. In addition, Diagnostic Providers can also provide access to any self diagnostic tests that are available from a diagnostic domain.

# Creating a Diagnostic Provider

Use Diagnostic Providers to query the startup configuration, current configuration, and current state of a diagnostic domain. Diagnostic Providers also provide access to any self diagnostic tests that are available from a diagnostic domain.

## Before you begin

To complete this task you must have programming knowledge of your system and the proper authorities to perform the following steps.

## About this task

The steps that follow outline a general process for creating Diagnostic Providers (DP).

1. Determine your diagnostic domain. Look for *configuration* MBeans that control a similar domain in the same component. Extending an existing configuration MBean with a DP interface avoids proliferation of new MBeans and has the benefit that mapping from a diagnostic MBean to a configuration MBean requires no additional information.
2. Determine what configuration attributes you want to expose. Include information that is used to configure your component from the configuration MBeans.
3. Determine what state attributes you want to expose. Anything you might want to know about the state of your component for troubleshooting can go here.
4. Determine what self diagnostic tests you will expose.
5. Determine what test attributes you will return for each self diagnostic.
6. Create your DP registration Extensible Markup Language (XML) file.
7. Create your DP implementation.
   a. To see an example, refer to "Implementing a Diagnostic Provider" on page 129 and keep in mind that most things that a Diagnostic Provider should do are already done for you in the *DiagnosticProviderHelper* class.
   b. To ensure that you do not collect unwanted data, add hooks in your component code where you need to collect state data using the *DiagnosticConfig* object.
   c. Add hooks in your component code where you need to store or be able to access configuration data.

8. Add code to register your DP implementation. Typically, the best place to do this is where your component is initialized.
9. Add Diagnostic Provider IDs (DPID) to your logged messages. Registering a DPID with a logger makes that information available in any messages logged with this logger. This enables fast paths in the DP utility to function on this particular Diagnostic Provider.
   a. Register your DPID with your loggers (for any of your loggers that you only want to associate a single DPID with).
   b. When you use multiple DPIDs with the same Logger, you can (instead of registering a single DPID with a Logger) add DPIDs to individual logging calls in the **parm[0]** position. Do not put **{0}** in the corresponding localized messages. It is bad practice to print the DPID in your messages as this would be inconsistent with messages from loggers with statically assigned DPIDs.

## Diagnostic Provider Extensible Markup Language

Some conventions to follow for Diagnostic Provider (DP) Extensible Markup Language (XML) declarations.

These guidelines are to help keep your use of Diagnostic Providers (DP) consistent.

- Include the Document type definition (DTD) for your Diagnostic Provider at the top of every DP declaration Extensible Markup Language (XML) file.
- Start all names and name segments with lower case. Use *camel case* for attribute names. That is, capitalize every initial letter in the name, except the first. For example, *traceCollectionSpec*.
- Indicate hierarchy with dashes. Dashes work better than dots because attribute names are regular expressions. For example, *traceService-traceCollectionSpec*.
- Indicate string dynamic parts to attribute names using an asterisk (*). For example,

  ```
  vhosts-.*-webgroups-.*-webapps-.*-listeners-filterInvocationListeners
  ```

  which would match vhosts-**someHost**-webgroups-**someGroup**-webapps-**someApp**-listeners-filterInvocationListeners
- Indicate numeric dynamic parts to attribute names using **[0-9]***. For example,

  ```
  vhosts-index-[0-9]*
  ```

  which would match webcontainer-vhosts-index-123
- If you have a general purpose self diagnostic test that can be run without significant performance cost, name it *general*.

### Some tips for configDump implementation

- configDump should contain information used to define the component's environment. Some examples are:
  - configuration data set by Java Management Extensions (JMX)
  - configuration from system properties, xml files, and property files
  - configuration information hard-wired and unchanging in code (such as, if a resource adapter implements interface X, or has some static final field Y, then those could indicate aspects of configuration and be included in the configDump).
- configDump should not contain dynamically registered attributes, such as:
  - a list of registered loggers (this belongs in stateDump)
  - a list of servlets in an application (this belongs in stateDump).
- configDump should be separated into 2 sections -- *startup* and *current*.
  - All configDump attributes must start with either **startup-** or **current-**.
  - The *startup* section details the component's environment at startup time. Startup configDump attributes start with **startup-** .
  - The *current* section details the component's environment at the moment the configDump is requested. Current configDump attributes start with **current-**.

### Best practices for configDump

- Group related attributes using an attribute hierarchy (such as, for two attributes about the traceLog: startup-traceLog-rolloverSize=20, startup-traceLog-maxNumberOfBackupFiles=1)
- For information in the current attribute list that refers to the same thing as a startup attribute, the names of both current and startup attributes should match.
- If an attribute has no use after startup, only show it in the startup section (for example, a configuration attribute that contains a file name from which startup data is read).

**Related tasks**

"Creating a Diagnostic Provider" on page 127
Use Diagnostic Providers to query the startup configuration, current configuration, and current state of a diagnostic domain. Diagnostic Providers also provide access to any self diagnostic tests that are available from a diagnostic domain.

"Working with Diagnostic Providers" on page 121
Diagnostic Providers enable you to query the startup configuration, current configuration, and current state of a diagnostic domain. In addition, Diagnostic Providers can also provide access to any self diagnostic tests that are available from a diagnostic domain.

## Choosing a Diagnostic Provider name

To ensure consistency when choosing Diagnostic Provider names to use with your components, you should consider the guidelines that follow.

Diagnostic Provider name guidelines:

- Names must be unique within a Java Virtual Machine (JVM). One Diagnostic Provider name goes uniquely with one Diagnostic Provider ID within a server.
- If necessary, names can contain a dynamic element to help with uniqueness. Of course, the dynamic element should have meaning to the administrator.
- Although not a hard limit, the static part of names should be 16 characters or less.
- The static part of names must follow the class name convention. Start with a capital letter, no spaces, and capitalize each word in the name.
- The static part of names must end with **DP**.
- Valid names contain a static part only, or a static part followed by a dash (-), followed by a dynamic part. Some valid examples:
  - ConnMgrDP-instance_specific_stuff
  - WebContainerDP
  - AdvisorDP
  - NodeAgentDP

**Related tasks**

"Creating a Diagnostic Provider" on page 127
Use Diagnostic Providers to query the startup configuration, current configuration, and current state of a diagnostic domain. Diagnostic Providers also provide access to any self diagnostic tests that are available from a diagnostic domain.

"Working with Diagnostic Providers" on page 121
Diagnostic Providers enable you to query the startup configuration, current configuration, and current state of a diagnostic domain. In addition, Diagnostic Providers can also provide access to any self diagnostic tests that are available from a diagnostic domain.

## Implementing a Diagnostic Provider

To use a Diagnostic Provider you must configure an MBean with the methods and attributes required to handle the data from the application server and client applications.

## Before you begin

This task presumes that you have a programming knowledge of the creation of MBeans. For more information about the interaction of MBeans with WebSphere Application Server, refer to topic, Creating and registering standard, dynamic, and open custom MBeans in the *Administering applications and their environment* PDF book.

## About this task

The steps that follow outline a general process for implementing a Diagnostic Provider (DP).

1. Modify the MBean descriptor Extensible Markup Language (XML). To implement a Diagnostic Provider, you must have an MBean, and the MBean should include this statement in its descriptor XML as a direct child of the MBean element:

   ```
   <parentType type="DiagnosticProvider"/>
   ```

   This defines the operations, attributes, and aggregators necessary for an MBean to be a Diagnostic Provider. If you do not need to have this DP exist in z/OS® Controllers, then this XML inclusion handles all z/OS specifics for your MBean.
2. Modify the MBean Implementation. Your MBean should already have a class which instantiates it and registers it with the Java Management Extensions (JMX) server.

   The first difference here is that you must define a property in the *Properties* class that is passed to the registration (and becomes part of the *ObjectName*). The property is **diagnosticProvider=true** and it can be added with a line of code such as:

   ```
   MyProps.setProperty(DiagnosticProvider.DIAGNOSTIC_PROVIDER_KEY, DiagnosticProvider.DIAGNOSTIC_PROVIDER_VALUE) ;
   ```

   The second difference is that this class should register this Diagnostic Provider with the Diagnostic Service. A helper method is available to do this:

   ```
   DiagnosticProviderHelper.registerMBeanWithDiagnosticService(DiagnosticProviderPName, DiagnosticProviderId) ;
   ```

   Obviously this must be done after the registration when the *ObjectName* can be retrieved into the *DiagnosticProviderId* string.
3. Implement the Diagnostic Provider methods.

***Diagnostic Provider method implementation:***

To create a Diagnostic Provider (DP) you must have an MBean that includes the required methods in its deployment Extensible Markup Language (XML) file. These methods define the operations, attributes, and aggregators necessary for an MBean to be a Diagnostic Provider.

Adding these methods can be accomplished by adding the *parentType* directive to your existing XML file (see “Implementing a Diagnostic Provider” on page 129), or by including the operations directly into your deployment XML file. The definitions needed are included in “Diagnostic Provider registered attributes and registered tests” on page 124. The next step is for the MBean to actually implement these methods. The methods to implement include:

- “getRegisteredDiagnostics” on page 131
- “getDiagnosticProviderName” on page 131
- “getDiagnosticProviderID” on page 131
- “configDump” on page 131
- “stateDump” on page 132
- “selfDiagnostic” on page 132
- “localize” on page 133

### getRegisteredDiagnostics

This method exposes the registration information for this Diagnostic Provider. It is commonly used by the DP Utility in the administration console to gather information about Diagnostic Providers that are to be displayed in the console. This method returns a **DiagnosticProviderInfo** object that is usually attained by passing the appropriate XML to a **DiagnosticProviderHelper** helper class. Here is an example:

```
public DiagnosticProviderInfo getRegisteredDiagnostics() {
  InputStream regIS= Thread.currentThread().getContextClassLoader().getResourceAsStream(
   "com/ibm/ws/xxx/SampleDP2DiagnosticProvider.xml");
  dpInfo = DiagnosticProviderHelper.loadRegistry(regIS, sDPName) ;

         if (dpInfo == null) {
            sSampleDP2MBeanLogger.logp(Level.WARNING, sThisClass, "getRegisteredDiagnostics",
             "RasDiag.DPInfo.NoGotz") ;
  }
    return dpInfo ;
 }
```

Notice that the XML is packaged and available in the *classpath* of the current *classloader*. The "Registration XML" on page 133 contains crucial information that the Diagnostic Provider uses to "Populate the payload" on page 133 and "localize" on page 133 results.

### getDiagnosticProviderName

This is usually a pretty simple return of a constant as the following example shows

```
public String getDiagnosticProviderName() {
  return sDPName;
 }
```

### getDiagnosticProviderID

This is usually a pretty simple return of a Java Management Extensions (JMX) object ID that MBeans can pull out of the base class method. For example:

```
public String getDiagnosticProviderId() {
  return getObjectName().toString() ;
 }
```

### configDump

The *configDump* method enables the Diagnostic Provider to expose the configuration data that was in place when this Diagnostic Provider started (or the current values of them). The **DiagnosticEvent** objects that this method returns include a "Payload" on page 133 that contains the core data. The following is an excerpt from a configDump method:

```
public DiagnosticEvent [] configDump(String aAttributeIdSpec, boolean aRegisteredOnly) {
    HashMap cdHash = new HashMap(64) ;

  // "Populate the payload" on page 133

    DiagnosticEvent [] diagnosticEvent = new DiagnosticEvent[1] ;
    diagnosticEvent[0] = DiagnosticEventFactory.createConfigDump(getObjectName().toString(),
     "ThisClassName", "configDump", cdHash) ;

    return diagnosticEvent ;
 }
```

This returns an array of **DiagnosticEvent** objects. Normally, *configDump* and *stateDump* return only one object. However, the method accepts an array because on z/OS systems a server can have multiple servants, and aggregation of the output from the servants is stored in the array.

### stateDump

The *stateDump* method enables the Diagnostic Provider to expose the current state data, or data about the current operating conditions of the Diagnostic Provider. The data made available can be anything likely to assist a customer, an IBM support person, or automated tooling in analyzing the health of the component and problem determination if there is an issue. The amount of data available is impacted by the State Collection Specification in effect at the time. If the current State Collection Specification involves the collection of additional data by the Diagnostic Provider, then this additional data can be exposed in the stateDump. The **DiagnosticEvent** objects that this method returns include a "Payload" on page 133 that contains the core data. The following is an excerpt from a stateDump method:

```
public DiagnosticEvent [] stateDump(String aAttributeIdSpec, boolean aRegisteredOnly) {
    HashMap sdHash = new HashMap(64) ;

  // "Populate the payload" on page 133

    DiagnosticEvent [] diagnosticEvent = new DiagnosticEvent[1] ;
    diagnosticEvent[0] = DiagnosticEventFactory.createStateDump(getObjectName().toString(),
     "ThisClassName", "stateDump", sdHash) ;

    return diagnosticEvent ;
 }
```

This returns an array of **DiagnosticEvent** objects. Normally, *configDump* and *stateDump* return only one object.

### selfDiagnostic

The *selfDiagnostic* method enables the Diagnostic Provider to perform certain predefined activities to test key functionalities of your system. These tests should not have a lasting effect on the system. For example, if the test is to create a TCP/IP connection to a remote host, the test should also break that connection before returning its results so that the state of the component is unchanged by the test. The information returned by the test is determined by the attributes included in the test section of the XML file. The following is an excerpt from a selfDiagnostic method:

```
public DiagnosticEvent [] selfDiagnostic(String aAttributeIdSpec, boolean aRegisteredOnly) {
  TestInfo [] testInfo = dpInfo.selfDiagnosticInfo.testInfo ; // Retrieve the test registry information
  Pattern testChecker = Pattern.compile(aAttributeIdSpec) ; // Compile test regexp parm for faster checking
  ArrayList deList = new ArrayList(8) ;    // Allocate expandable list of DiagnosticEvents
  for (int i = 0; i < testInfo.length; i++) {
   if (testChecker.matcher(testInfo[i].id).matches()) {
      HashMap deHash = new HashMap(32) ;

    // "Populate the payload" on page 133

      deList.add(DiagnosticEventFactory.createDiagnosticEvent(getObjectName().toString(),
       DiagnosticEvent.EVENT_TYPE_SELF_DIAGNOSTIC, DiagnosticEvent.LEVEL_INFO,
       "ThisClassName", "selfDiagnostic", dpInfo.resourceBundleName,
     "RasDiag.SDP2.createDE3",   // MsgKey for localization
           // Parms to incorporate in msg
     new Object [] { "OneParm", "TwoParm", "RedParm", "BlueParm"}, deHash)) ;
   }
  }

    DiagnosticEvent [] diagnosticEvent = new DiagnosticEvent[deList.size()] ;
  diagnosticEvent = (DiagnosticEvent [])deList.toArray(diagnosticEvent) ;

    return diagnosticEvent ;
 }
```

This returns an array of **DiagnosticEvent** objects. In this example, one **DiagnosticEvent** was created from each test that matched the parameter regular expression. The Diagnostic Provider is not required to

produce only one per test. The generation of “Payload” is similar to that of *configDump* and *stateDump*.

**localize**

The **DiagnosticEvents** that methods return contain payload **HashMaps** that contain **MessageKeys** and **ResourceBundles**. The final consumer of these events is often not on the server, and thus may not have the appropriate *classpath* to resolve this. For this purpose, a callback to the Diagnostic Provider to localize the variables is done. A helper method, however, makes it a simple method to write, as this example demonstrates:

```
public String [] localize(String [] aKeys, Locale aLocale) {
  return DiagnosticProviderHelper.localize(dpInfo.resourceBundleName, aKeys, aLocale) ;
 }
```

Note that the **dpInfo** (DiagnosticProviderInfo) object is needed as this object includes a reference to the **ResourceBundle**.

**Payload**

A recurring theme in these methods is the ability to include a payload in return objects. This is a set of *name=value* pairs that include the information being exposed by the method. Diagnostic Events returned from a configDump, stateDump, or selfDiagnostic test are relatively complex Java objects. The majority of the information that is returned is contained in the **DiagnosticData** portion of the **DiagnosticEvent** object. Each attribute returned by the Diagnostic Provider is stored in an entry in a **HashMap**. There can be cascading HashMaps within a single DiagnosticEvent object (if breaking the data down into subGroups makes sense). Each HashMap entry contains either a reference to a child HashMap, or a **DiagnosticTypedValue** (which contains the value, the type of data, and a **MsgKey** for localization of the label or /name). The values to be returned should be filtered with:

- The type of method (that is, configDump, stateDump, or selfDiagnostic)
- The **AttributeIdSpec** sent in to filter the values
- The current State Collection Specification (which can impact the amount of data available).

**Populate the payload**

The API documentation for *DiagnosticProviderHelper.queryMatchingDPInfoAttributes* explains how to do the filtering before retrieving the data. In some cases, it is easier and helps performance for a Diagnostic Provider to retrieve all data into the Payload and then filter the HashMap after the fact. The post-population filtering can be done with the method *DiagnosticProviderHelper.filterEventPayload*. For information on use of the JavaBean type approach, see the API documentation for the *AttributeBeanInfo.populateMap* method.

**Registration XML**

Registration XML enables much of the information needed by the Diagnostic Provider to be externalized. It also provides a means of commonizing localization and consumption of the tests (thus aiding automation). An excerpt of this XML from a sample Diagnostic Provider follows:

```
<!DOCTYPE diagnosticProvider PUBLIC "RasDiag" "/DiagnosticProvider.dtd">

<diagnosticProvider>
    <resourceBundleName> com.ibm.ws.rasdiag.resources.RasDiagSample</resourceBundleName>
    <state>
 <attribute>
  <id>Leg-Foot</id>
  <descriptionKey>SampleDiagnostic.LegFoot.descriptionKey</descriptionKey>
  <registered>true</registered>
 </attribute>
 <attribute>
  <id>Leg-Ankle</id>
```

```
  <descriptionKey>SampleDiagnostic.LegAnkle.descriptionKey</descriptionKey>
  <registered>true</registered>
        </attribute>
    </state>
    <config>
        <attribute>
  <id>Arm-Hand-Size</id>
  <descriptionKey>SampleDiagnostic.HandSize.descriptionKey</descriptionKey>
  <registered>true</registered>
        </attribute>
        <attribute>
  <id>Leg-Foot-Size</id>
  <descriptionKey>SampleDiagnostic.FootSize.descriptionKey</descriptionKey>
  <registered>true</registered>
        </attribute>
    </config>
    <selfDiagnostic>
        <test>
  <id>Kick</id>
  <descriptionKey>SampleDiagnostic.Kick.descriptionKey</descriptionKey>
  <attribute>
   <id>Kick-Pain</id>
   <descriptionKey>SampleDiagnostic.KickPain.descriptionKey</descriptionKey>
  </attribute>
  <attribute>
   <id>Kick-Length</id>
   <descriptionKey>SampleDiagnostic.KickLength.descriptionKey</descriptionKey>
  </attribute>
        </test>
 <test>
  <id>Throw</id>
  <descriptionKey>SampleDiagnostic.Throw.descriptionKey</descriptionKey>
         <attribute>
   <id>Throw-Pain</id>
   <descriptionKey>SampleDiagnostic.ThrowPain.descriptionKey</descriptionKey>
   <registered>true</registered>
  </attribute>
  <attribute>
   <id>Throw-Length</id>
   <descriptionKey>SampleDiagnostic.ThrowLength.descriptionKey</descriptionKey>
   <registered>true</registered>
  </attribute>
        </test>
    </selfDiagnostic>
</diagnosticProvider>
```

For understanding the storage of this information into a **DiagnosticProviderInfo** object, see the API documentation for *DiagnosticProviderInfo*. For conceptual information about the purpose of the registration XML, see “Diagnostic Provider registered attributes and registered tests” on page 124.

*Diagnostic Provider XML example:*

Here is an example of the Diagnostic Provider Extensible Markup Language (XML).

```
version="6.0"
platform="common"
aggregationHandlerClass="com.ibm.ws.management.component.DiagnosticProviderAggregator"
description="DiagnosticProvider portion of Mbean for inclusion into MBeans implementing this interface">
  <attribute
    description="DiagnosticProviderName (not dependent on runtime, but subset of ObjectName"
    getMethod="getDiagnosticProviderName" name="diagnosticProviderName"
    type="java.lang.String" proxyInvokeType="unicall" proxySetterInvokeType="multicall"/>
  <operation
    description="Get the DiagnosticProvider ID"
    impact="INFO" name="getDiagnosticProviderId" role="operation"
    targetObjectType="objectReference" type="java.lang.String" proxyInvokeType="unicall">
```

```
  <signature/>
</operation>
<operation
  description="Return the registry information based on type (config/state/selfDiag)."
  impact="INFO" name="getRegisteredDiagnostics" role="operation"
  targetObjectType="objectReference"
  type="com.ibm.wsspi.rasdiag.diagnosticProviderRegistration.DiagnosticProviderInfo"
  proxyInvokeType="unicall">
  <signature/>
</operation>
<operation
  description="Dump the configuration information associated with managed resource."
  impact="INFO" name="configDump" role="operation"
  targetObjectType="objectReference" type="[Lcom.ibm.wsspi.rasdiag.DiagnosticEvent;"
  proxyInvokeType="multicall">
  <signature>
    <parameter description="Attribute ID to use"
      name="attributeId" type="java.lang.String"/>
    <parameter description="Report on just registered info, or all info"
      name="registeredOnly" type="boolean"/>
  </signature>
</operation>
<operation
  description="Dump state information for the managed resource."
  impact="INFO" name="stateDump" role="operation"
  targetObjectType="objectReference" type="[Lcom.ibm.wsspi.rasdiag.DiagnosticEvent;"
      proxyInvokeType="multicall">
  <signature>
    <parameter description="Attribute ID to use"
      name="attributeId" type="java.lang.String"/>
    <parameter description="Report on just registered info, or all info"
      name="registeredOnly" type="boolean"/>
  </signature>
</operation>
<operation
  description="Perform diagnostics on the managed resource driven by current diagnostic mode setting."
  impact="ACTION" name="selfDiagnostic" role="operation"
  targetObjectType="objectReference" type="[Lcom.ibm.wsspi.rasdiag.DiagnosticEvent;"
      proxyInvokeType="multicall">
  <signature>
    <parameter description="Test ID to use"
      name="testId" type="java.lang.String"/>
    <parameter description="Report on just registered info, or all info"
      name="registeredOnly" type="boolean"/>
  </signature>
</operation>
<operation
  description="localize messages for console display"
  impact="INFO" name="localize" role="operation"
  targetObjectType="objectReference" type="[Ljava.lang.String;"
      proxyInvokeType="unicall">
  <signature>
    <parameter description="Message Keys" name="msgKeys" type="[Ljava.lang.String;"/>
    <parameter description="Locale to use for output" name="locale" type="java.util.Locale"/>
  </signature>
</operation>
```

**Related tasks**

“Working with Diagnostic Providers” on page 121
Diagnostic Providers enable you to query the startup configuration, current configuration, and current state of a diagnostic domain. In addition, Diagnostic Providers can also provide access to any self diagnostic tests that are available from a diagnostic domain.

## Creating a Diagnostic Provider registration XML file

The Diagnostic Provider registration XML is used to provide information about the exposed configuration, state, and self diagnostic attributes and tests to the Diagnostic Provider utility. It is also used to populate objects needed later in the process, to assist in filtering, and to assist in localization.

### Before you begin

Programming knowledge of your system and the proper authorities to perform the following steps.

### About this task

The steps that follow outline a general process for creating a Diagnostic Provider (DP) registration Extensible Markup Language (XML) file.

1. Start with the DP document type definition (DTD). If you are using the helper methods (see the step called *Create your DP implementation* in “Creating a Diagnostic Provider” on page 127), you can use this DOCTYPE line to pick up the common DTD:

   ```
   <!DOCTYPE diagnosticProvider PUBLIC "RasDiag" "/DiagnosticProvider.dtd">
   ```

   If you are extending an existing MBean with an existing XML configuration, you might need either to add the DP XML to an existing DTD, or omit the DP XML entirely. If you omit the DP XML, you will not be able to validate that your XML file is well formed.
2. Follow the conventions described in “Diagnostic Provider Extensible Markup Language” on page 128 to help keep your XML consistent with other components. You can find an example of a small DP registration XML file in “Diagnostic Provider method implementation” on page 130.

# Associating a Diagnostic Provider ID with a logger

If you are using a Diagnostic Provider to manage alerts and messages, you need to associate the Diagnostic Provider ID with a logger. This can be done dynamically or through a static assignment.

## About this task

Components whose diagnostics are managed through a Diagnostic Provider MBean should include the Diagnostic Provider ID (DPID) in all logged messages. In some cases a single logger always logs with the same DPID. In those cases, it is appropriate to statically associate the DPID with the logger. In other cases, a logger might log on behalf of various diagnostic domains. For example, although every data source has a separate Diagnostic Provider MBean, they all share the same logger. In those cases, the DPID can be dynamically supplied on each logging call.

## Static Assignment

### About this task

The method below statically assigns a DPID to a logger.

Associate a DPID with a logger:

```
Logger logger = Logger.getLogger("com.ibm.ws.MyClass");
DiagnosticProviderHelper.addDiagnosticProviderIDtoLogger(logger, dpid);
```

### Dynamic Assignment
#### About this task

DPIDs can be associated with a single log request by including them as the first message parameter, prefixed with **DPID:.** To associate a DPID with a single log request using a logger:

```
Object[] parms = new Object[] { "DPID:" + dpid };
logger.logp(classname, methodname, "MSG0001", parms);
```

Note that in the dynamic case, the DPID does not need to actually show up in the formatted message. The two examples below illustrate:

```
(in resource bundle)
// by not including {0} first parm is not printed in the message.
MSG0001=This message does not include the DPID.

// note - it is not recommended to print the DPID in your message.
MSG0002=This message includes the DPID...it's value is {0}.
```

It is recommended that messages not include the DPID in the formatted message. As shown above, this is done by not including {0} in the message value in the resource bundle.

## Using Diagnostic Providers from wsadmin scripts

In addition to enabling Diagnostic Providers (DP) from the administration console, you can also use them through scripts from the Wsadmin tool.

### About this task

You might want to enable, disable, or configure Diagnostic Providers from the administrative console, but in some cases it might be more efficient or useful to do so with scripts using the wsadmin tool.

For more detailed information about the Wsadmin tool see the scripting tool chapter in the *Administering applications and their environment* PDF book

1. List the MBeans that implement the Diagnostic Provider (DP) interface. Enter

   ```
   $AdminControl queryNames diagnosticProvider=true,*
   ```

   And you will see an output that displays all of the Diagnostic Providers in a format like this:

   ```
   "WebSphere:name=Default Datasource,process=server1,platform=dynamicproxy,node=
     camelhair,JDBCProvider=Derby JDBC Provider,
   diagnosticProvider=true,j2eeType=JDBCDataSource,J2EEServer=server1,Server=server1,
     version=6.1.0.0,type=DataSource,
   mbeanIdentifier=cells/camelhairCell/nodes/camelhair/servers/server1/resources.xml#
     DataSource_1131113688564,
   JDBCResource=Derby JDBC Provider,cell=camelhairCell"
   "WebSphere:name=DefaultEJBTimerDataSource,process=server1,platform=dynamicproxy,
     node=camelhair,
   JDBCProvider=Derby JDBC Provider (XA),diagnosticProvider=true,j2eeType=
     JDBCDataSource,J2EEServer=server1,Server=server1,version=6.1.0.0,type=DataSource,
     mbeanIdentifier=cells/camelhairCell/nodes/camelhair/servers/server1/
     resources.xml#DataSource_1000001,
   JDBCResource=Derby JDBC Provider (XA),cell=camelhairCell"
   WebSphere:name=WebcontainerDiagnosticProvider,process=server1,platform=
     dynamicproxy,node=camelhair,diagnosticProvider=true,
   version=6.1.0.0,type=WebcontainerEventProvider,mbeanIdentifier=null,
     cell=camelhairCell
   ```

2. Capture the ObjectName of your Diagnostic Provider in a variable. This enables you to reference your Diagnostic Provider more easily, especially in a script. For example, instead of typing all of those lines, if you want to work with the WebContainer Diagnostic Provider, for example, you can do the following:
   - set DP [lindex [$AdminControl queryNames name=WebcontainerDiagnosticProvider,diagnosticProvider=true,*] 0]

This ObjectName stored in the DP variable can be used on the methods, or you can use the Diagnostic Provider name as text or a variable.

- Now that you have the ObjectName in a variable, you can get the Diagnostic Provider name in a variable with the command:

  ```
  set DPNm [$AdminControl invoke $DS getDiagnosticProviderNameById $DP]
  ```

  This provides the result:

  ```
  WebContainerDP
  ```

  Now the DiagnosticProvider (WebContainer) is addressable by its objectname in variable DP, or by its DiagnosticProvider name in variable DPNm. If you would prefer, you can hard-code the DPName *WebContainerDP* as it is short enough.

3. Save the ObjectName of the DiagnosticService MBean to a variable. For wsadmin, WebSphere Application Server provides this MBean so that the output of the Diagnostic Provider is more easily consumable. Enter

   ```
   set DS [lindex [$AdminControl queryNames name=DiagnosticService,*] 0]
   ```

4. Run a configDump. You can run a configDump and capture all attributes with the command:

   ```
   $AdminControl invoke $DS configDumpFormattedById [list $DP .* true null]
   ```

   This lists the values that the Diagnostic Provider used at start up (and possible current values). An excerpt of the configDump output follows.

| Item Concatenated Name | Value |
|---|---|
| customProperties = | Null |
| defaultVirtualHostName = | default_host |
| jvmProps = | Null |
| localeProps = | Null |
| servletCachingEnabled = | false |
| aliases = | *:9080;*:80;*:9443; |

5. Filter the output of your configDump. You can use configDumpFormatted (leaving off the ById) and switch **$DP** for **$DPNm** or the string **WebContainerDP**. This example uses *$DPNm* on this slightly modified version whereby it only picks up attributes dealing with automation:

   ```
   $AdminControl invoke $DS configDumpFormatted [list $DPNm .*auto.* true null]
   ```

   This results in just those attributes that contain **auto** in them. Full (but strict) regular expression syntax is allowed:

| Item Concatenated Name | Value |
|---|---|
| autoLoadFiltersEnabled = | false |
| autoRequestEncoding = | false |
| autoResponseEncoding = | false |
| autoLoadFiltersEnabled = | false |
| autoRequestEncoding = | false |
| autoResponseEncoding = | false |

The syntax is the same for stateDumps and selfDiagnostics

## Viewing the run time configuration of a component using Diagnostic Providers

You can use the administrative console to navigate to configuration data that can be used to check the health of a server runtime component.

## Before you begin

You must have sufficient authority to run the action.

## About this task

Runtime components that have associated diagnostic providers can include their Diagnostic Provider ID (DPID) in their log entries. If you know the DPID, you can enter it directly in the quick link text box. Otherwise, navigate to the desired process by using the tree view displayed at the bottom of the panel, as shown in the steps below.

1. Start the administration console.
2. From the task bar on the left side of the console, select **Troubleshooting**.
3. From the task bar on the left side of the console, select **Diagnostic Provider**.
4. From the task bar on the left side of the console, select **Configuration Data**.
5. Either directly enter a Diagnostic Provider ID in the **Quick link using diagnostic provider ID** text box, or select a process (cluster / node / server) from the available processes displayed at the bottom of the panel under the section title **Server selection topology**.
6. From the list of available diagnostic providers for the selected process, choose the desired diagnostic provider name. The configuration data for that diagnostic provider appears.

## Configuration data quick link or server selection

Use this panel to select a Diagnostic Provider server for viewing run time configuration data.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **Configuration Data**

***Quick link using Diagnostic Provider ID:*** From the Configuration data panel, enter a Diagnostic Provider ID to go directly to the page for the configuration data for the Diagnostic Provider for the specific server.

***Server selection topology:***

Use these folders to select server or cluster for viewing the configuration data for a Diagnostic Provider.

If you choose a cluster, whatever action you choose is performed on *each* server in the cluster.

The enterprise applications section shows you the servers that a particular application is running on. If you select a server from this list, the action is performed on that server, not specifically that application.

## Diagnostic Providers (selection)

Use this panel to select a Diagnostic Provider from the selected server or cluster.

The list will contain only Diagnostic Providers registered on the selected server or cluster. Not all Diagnostic Providers register with every server in the cell.

You can follow several navigation paths to view this administrative console page. For example, click **Troubleshooting** > **Diagnostic Provider** > **Tests** > select a server or cluster name.

***Name:***

Choose a diagnostic provider from this list.

The path you chose to get to this panel determines which panel displays next.

- If you chose Troubleshooting > Diagnostic Provider > Tests, you see a panel that lists all of the available tests to run on the Diagnostic Provider.

- If you chose Troubleshooting > Diagnostic Provider > State Data, you see a panel that shows the collected state data for the Diagnostic Provider.
- If you chose Troubleshooting > Diagnostic Provider > Configuration Data, you see a panel that shows the configuration data for the Diagnostic Provider.

### Configuration data

Use this panel to view the current configuration data for a Diagnostic Provider on a selected server or cluster. Not necessarily every piece of configuration data appears, but data that can be helpful in problem determination is shown.

You can follow several navigation paths to view this administrative console page. For example, click **Troubleshooting** > **Diagnostic Provider** > **Configuration data** > select a server or cluster name > select a Diagnostic Provider from the list.

The attributes show information that has been configured for the Diagnostic Provider. You can use the **Save** button to save the information to a file.

**Note:** Results from a configuration dump contain names that start with either *startup* or *current*. The *startup* entries represent data that was read in by the component at server startup time. The *current* entries contain data that is current – meaning the value of the attributes in use by the runtime at the time the configuration dump was requested.

***Node:***

This is the node name from where the configuration data was collected.

***Server:***

This is the server name from where the configuration data was collected.

***Name:***

This is the name of the attribute for the configuration data.

***Value:***

This is the value of the configuration data.

***Description:***

This is a description of the configuration data.

## Viewing the run time state data or configuring the state data collection specifications for a Diagnostic Provider

Use the administrative console to navigate to the state data that can be used to check the health of a server runtime component, or you can configure the state data to be collected for a server.

### Before you begin

You must have sufficient authority to execute the action.

### About this task

In the server selection topology section, use the view state data radio button to go to the list of registered diagnostic providers. Use the change state data collection specification radio button to modify the state collection specification for the runtime components for a server. Runtime components that have associated

diagnostic providers can include their Diagnostic Provider ID (DPID) in their log entries. If you know the DPID, you can enter it directly in the quick link text box.

1. Start the administration console.
2. Select **Troubleshooting**.
3. Select **Diagnostic Provider**.
4. Select **State Data**.
5. Select the **View State Data** radio button to simply look at the state data, or select the **Change state data collection specification** radio button to change the configuration.
6. Either directly enter a Diagnostic Provider ID in the **Quick link using diagnostic provider ID** text box, or select a process (cluster / node / server) from the available processes displayed at the bottom of the panel.
   - If you chose the **View State Data** radio button, a panel listing the available Diagnostic Providers appears. Choose one of the providers by clicking on it. A panel displaying the state data appears.
   - If you chose the **Change state data collection specification** radio button, a panel appears that contains a list of the available Diagnostic Providers and a text entry block. The state collection specification for the selected process is managed from this panel. Select one of the available providers by using the checkbox next to it.

## Diagnostic Provider State Collection Specification

The State Collection specification provides a mechanism for indicating what additional data diagnostic providers in the system should retain in cases where this additional data could be useful for problem determination or application tuning.

In normal operation, most components should work optimally and not store any operational data that is not needed. There are times, however, when an administrator or automated tool may want a component to collect more information than normal to help in problem determination. This data could then be exposed through a State dump. The State Collection specification was created as a syntax for indicating what additional data the diagnostic providers in the system should retain.

For the syntax of the *aCollectionSpec* string, refer to the **DiagnosticConfigHome** API documentation. It is basically a semicolon (;) separated list of collection specification clauses which are of the form:

```
<DiagnosticProviderName regexp>:<AttributeId regexp>=[0|1]
```

Where the *DiagnosticProviderName* regular expression will make this clause apply to any Diagnostic Provider Name that matches that regular expression. The *AttributeId regexp* and the boolean value (**0** for off, and **1** for on) are stored in the *DiagnosticConfig* object that each Diagnostic Provider uses. Turning on or off, and processing the clauses left to right allows relatively complex specification. Any specification that is not explicitly turned **on** is considered to be off. This format is explained further in the following examples.

To turn on tracing for all attributes in the **MyDP** Diagnostic Provider:

```
MyDP:.*=1
```

To turn on tracing for *all* attributes of *all* Diagnostic Providers (this will probably impact system performance):

```
.*:.*=1
```

To turn on all tracing for all attributes of all Diagnostic Providers beginning with **ConnMgr** (for example, Data Sources):

```
ConnMgr.*:.*=1
```

This specification turns on special collection attributes in the **MyDP** Diagnostic Provider that begin with the string **PoolInfo**. If, however, the attribute begins with **PoolInfo.Db2Pool**, then the collection is off (because it is read left to right).

```
MyDP:PoolInfo.*=1;MyDP:PoolInfo.Db2Pool.*=0
```

It should be noted that State dumps can return important information even in the case where there is no State Collection Specification turned on for a Diagnostic Provider. Diagnostic Providers frequently have to keep some state information in order to operate. Anything in this category is available in a State dump even if there is no special data collection going on. Using the State Collection Specification may increase the amount of data available.

## State Data Quick Link or Server Selection

Use this panel to select a server or cluster to either view collected state data, or to configure state data to collect for a Diagnostic Provider.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **State Data**.

***Quick link using Diagnostic Provider ID:***

Enter a Diagnostic Provider ID to go directly to the view page for the collected state data for the Diagnostic Provider.

***Server selection topology:***

Use these radio buttons and folders to select a specific server or cluster for viewing of state data or configuring the specification of state data.

If you choose a cluster, whatever action you choose is performed on *each* server in the cluster.

The enterprise applications section shows you the servers that a particular application is running on. If you select a server from this list, the action is performed on that server, not specifically that application.

**View state data**
Select this radio button to view the state data for a Diagnostic Provider. Then select the cell or cluster you want to work with.

**Change state data collection settings**
Select this radio button to configure the state collection specification for a Diagnostic Provider. Then select the cluster or managed server you want to work with.

## State data

Use this panel to view the current state data for a Diagnostic Provider on a selected server or cluster.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **State data** > select the View state data radio button and then select a server or cluster name > select a Diagnostic Provider from the list.

The attributes show information that has been collected as part of the enabled state collection specification for the Diagnostic Provider. You can use the **save...** button to save the information to a file

***Node:***

This is the node name from where the state data was collected.

***Server:***

This is the server name from where the state data was collected.

***Name:***

This is the name of the state collection specification used to collect the state data.

***Value:***

This is the value of the state collection specification used to collect the state data.

***Description:***

This is a description of the state collection specification used to collect the state data.

## Detailed state specification

Use this panel to view the attributes and descriptions of the Diagnostic Provider that you have selected.

To add attributes, select the checkbox next to your chosen diagnostic provider, then select the **Add to specification** button.

To remove a diagnostic provider's sub-component attribute from the state specification, select the sub-component attribute in the displayed list and then select the **Remove from specification** button.

When you are done adding or removing a diagnostic provider's sub-component attributes, select the **Done** button.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **State data** >select the View state data radio button and then select a server or cluster name > select a Diagnostic Provider from the list.

***Attribute:***

This is the individual state collection specification available for the Diagnostic Provider.

***Description:***

This is the description of the individual state collection specification item.

## Change state specification

Use this panel to add a Diagnostic Provider and its attributes to the specification for collecting state data.

To add a diagnostic provider (DP) and *all* of its attributes, select the checkbox next to your chosen DP, then click on the **Add to specification** button. To add only *some* of the DP's attributes, click on the DP name itself in the list, and a new panel where you can perform this task appears.

To put the state specification into affect, select the **Apply** or **OK** button.

To reset the specification to its original state, use the **Reset** button.

To manually enter a state specification, update the text area with the state specification and use the **Update** button.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **State data** > select the Change state data collection specification radio button and then select a server or cluster name > select a Diagnostic Provider from the list.

***Name:***

This is a list of available Diagnostic Providers for the server selected.

### Modifying the State Collection Specification from wsadmin scripts

In addition to modifying the State Collection Specification from the administrative console, you can also modify these settings using scripts and the wsadmin tool.

#### About this task

In doing problem determination, you might want to begin collecting additional data during normal processing. This can be accomplished by modifying the State Collection Specification dynamically. This section illustrates how to do that through the Wsadmin tool. This technique can be used to turn on traces, as well as to turn off traces. Depending on the usage pattern of the component, the impact should take affect shortly after it is set. For more information on this tool, see the chapter, Wsadmin tool in the *Administering applications and their environment* PDF book.

1. Capture the DiagnosticService ObjectName into a variable. Enter

   ```
   set DS [lindex [$AdminControl queryNames name=DiagnosticService,*] 0]
   ```

2. Use this variable to drive the method to set the specification. Enter

   ```
   $AdminControl invoke $DS setStateCollectionSpec "SampleDiagnosticProvider:player.*=1;
   SampleDiagnosticProvider:defense.*=1"
   ```

   The specification is of the form **DiagnosticProviderName:AttributeId=0|1...** (with a semicolon at the end, multiple sub-specifications can be entered similar to the TraceSpec). The DiagnosticProviderName and AttributeId can be proper regular expressions.

## Running a self diagnostic on a Diagnostic Provider

You can check the status of server runtime components with predefined tests that can be associated with a Diagnostic Provider. Use the administrative console to access these functions.

### Before you begin

You must have sufficient authority to execute the action.

### About this task

You can access a list of predefined diagnostic tests that you can use to check the status of a server runtime component. Runtime components that have associated diagnostic providers can include their Diagnostic Provide ID (DPID) in their log entries. If you know the DPID, you can enter it directly in the quick link text box. Otherwise, navigate to the desired process by using the tree view displayed at the bottom of the panel.

1. Start the administration console.
2. Select **Troubleshooting**.
3. Select **Diagnostic Provider**.
4. Select **Tests** .
5. Either directly enter a Diagnostic Provider ID in the **Quick link using diagnostic provider ID** text box, or select a process (cluster / node / server) from the available processes displayed in the **Server selection topology** section.
6. Select the desired self diagnostic test.
7. Read the output messages from the self diagnostic test.
8. Select a self diagnostic test message by clicking on it. The console displays a panel with the attributes related to the message you chose.

### Tests Quick Link or Server Selection

Use this panel to select a Diagnostic Provider server for diagnostic tests.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **Tests**.

***Quick link using Diagnostic Provider ID:***

Enter a Diagnostic Provider ID to go directly to the view page for the collected state data for the Diagnostic Provider.

***Server selection topology:***

Use these folders to select server or cluster for viewing the available tests for a Diagnostic Provider.

If you choose a cluster, whatever action you choose is performed on *each* server in the cluster.

The enterprise applications section shows you the servers that a particular application is running on. If you select a server from this list, the action is performed on that server, not specifically that application.

## Test selection

Use this panel to select one of the tests that are available for the chosen Diagnostic Provider on the chosen server or cluster.

You can follow several navigation paths to view this administrative console page. For example, click **Troubleshooting** > **Diagnostic Provider** > **Tests** > select a server or cluster name > select a Diagnostic Provider from the list.

***Test identification:***

Choosing a test ID causes the test to run. Results of the test are shown on the Test Results panel.

***Test description:***

A description of the test available to run on the Diagnostic Provider.

## Test Results

Use this panel to see the results from the server or cluster members for the selected test.

You can follow several navigation paths to view this administrative console page. For example, click **Troubleshooting** > **Diagnostic Provider** > **Tests** > select a cluster name > select a Diagnostic Provider from the list > select a Test identification from the list.

Multiple results can be returned from a test from each server. The results are sorted by Node, then by Server, then by Severity. You can page through the messages that are returned.

***Server:***

The name of the server where the test result came back from.

***Node:***

The name of the node where the test result came back from.

***Severity:***

The severity of the result from the test run.

***Message:***

A description of the test result.

The entries in this column are linked to another panel. If you click on a message, you can see additional attributes associated with the message.

## Test result details

Use this panel to see additional attributes for the selected test result.

To view this administrative console page, click **Troubleshooting** > **Diagnostic Provider** > **Tests** > select a cluster name > select a Diagnostic Provider from the list > select a Test identification from the list > select a message.

The attributes show information that helped to diagnose the condition described in the message. You can use the **Save** button to save to a file the attributes and the messages to which they correspond.

***Name:***

The name of the test.

***Value:***

This is the value of the test result.

***Description:***

This is a description of the test.

# Appendix. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories infer specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

## Default product locations (i5/OS)

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

*app_client_root*
The default installation root directory for the Java EE WebSphere Application Client is the /QIBM/ProdData/WebSphere/AppClient/V7/client directory.

*app_client_user_data_root*
The default Java EE WebSphere Application Client user data root is the /QIBM/UserData/WebSphere/AppClient/V7/client directory.

*app_client_profile_root*
The default Java EE WebSphere Application Client profile root is the /QIBM/UserData/WebSphere/AppClient/V7/client/profiles/*profile_name* directory.

*app_server_root*
The default installation root directory for WebSphere Application Server is the /QIBM/ProdData/WebSphere/AppServer/V7/Base directory.

*cip_app_server_root*
The default installation root directory is the /QIBM/ProdData/WebSphere/AppServer/V7/Base/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

A CIP is a WebSphere Application Server product bundled with optional maintenance packages, an optional configuration archive, one or more optional enterprise archive files, and other optional files and scripts.

*cip_profile_root*
The default profile root directory is the /QIBM/UserData/WebSphere/AppServer/V7/Base/cip/*cip_uid*/profiles/*profile_name* directory for a customized installation package (CIP) produced by the Installation Factory.

*cip_user_data_root*
The default user data root directory is the /QIBM/UserData/WebSphere/AppServer/V7/Base/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

*if_root* This directory represents the root directory of the IBM WebSphere Installation Factory. Because you can download and unpack the Installation Factory to any directory on the file system to which you have write access, this directory's location varies by user. The Installation Factory is an Eclipse-based tool which creates installation packages for installing WebSphere Application Server in a reliable and repeatable way, tailored to your specific needs.

*iip_root*
This directory represents the root directory of an *integrated installation package* (IIP) produced by the IBM WebSphere Installation Factory. Because you can create and save an IIP to any directory on the file system to which you have write access, this directory's location varies by user. An IIP is an aggregated installation package created with the Installation Factory that can include one or more generally available installation packages, one or more customized installation packages (CIPs), and other user-specified files and directories.

*java_home*
The following directories are the root directories for all supported Java Virtual Machines (JVMs).

| JVM | Directory |
| --- | --- |
| Classic JVM | /QIBM/ProdData/Java400/jdk6 |
| 32–bit IBM Technology for Java | /QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit |
| 64–bit IBM Technology for Java | /QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit |

*plugins_profile_root*
The default Web server plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver/profiles/*profile_name* directory.

*plugins_root*
The default installation root directory for Web server plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V7/webserver directory.

*plugins_user_data_root*
The default Web server plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver directory.

*product_library*
*product_lib*
This is the product library for the installed product. The product library for each Version 7.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS7*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 7.0 product installed on the system is `QWAS7A`. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, `was.install.library`, and is located under the *app_server_root* directory.

*profile_root*
The default directory for a profile named *profile_name* for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V7/Base/profiles/*profile_name* directory.

*shared_product_library*
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS7`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

*updi_root*
The default installation root directory for the Update Installer for WebSphere Software is the /QIBM/ProdData/WebSphere/UpdateInstaller/V7/updi directory.

*user_data_root*
The default user data directory for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V7/Base directory.

The profiles and profileRegistry subdirectories are created under this directory when you install the product.

*web_server_root*
The default web server path is /www/*web_server_name*.

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

# Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.