BEA WebLogic Server™

Using WebLogic Logging Services

Release 8.1
Revised: June 28, 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

Overview of WebLogic Logging Services

Writing Messages to the WebLogic Server Log

About This Document

This document describes how your application can write messages to the BEA WebLogic Server™ log files and listen for the log messages that WebLogic Server broadcasts. The document also outlines how you can use the WebLogic Server Administration Console to view log messages.

The document is organized as follows:

- Chapter 1, "Overview of WebLogic Logging Services," which summarizes WebLogic logging services.
- Chapter 2, "Writing Messages to the WebLogic Server Log," which describes how to create and use message catalogs and how to use the `NonCatalogLogger` class to write log messages.
- Chapter 3, "Viewing the WebLogic Server Logs," which describes how to use the Administration Console's log viewer.
- Chapter 4, "Filtering WebLogic Server Log Messages," which describes how to specify which types of messages WebLogic Server writes to it logs and to standard out.
- Chapter 5, "Subscribing to Messages," which describes how your application can receive messages that it and WebLogic Server subsystems generate.

Audience

This document is written for application developers who want to build Web applications or other Java 2 Platform, Enterprise Edition (J2EE) components that run on WebLogic Server. It is

assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. Specifically, "Logging" in the Administration Console Online Help describes how to configure log files that a WebLogic Server generates, and the Internationalization Guide describes how to set up message catalogs that your application can use.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`

Convention	Item
monospace italic text	Identifies variables in code. *Example*: `String expr`
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
. . .	Indicates one of the following in a command line: • That an argument can be repeated several times in a command line • That the statement omits additional optional arguments • That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER 1

Overview of WebLogic Logging Services

The WebLogic Server logging services include facilities for writing, viewing, filtering, and listening for log messages. While WebLogic Server subsystems use these services to provide information about events such as the deployment of new applications or the failure of one or more subsystems, your application can also use them to communicate its status and respond to specific events. For example, you can use WebLogic logging services to keep a record of which user invokes specific application components, to report error conditions, or to help debug your application before releasing it to a production environment. In addition, you can configure your application to listen for a log message from a specific subsystem and to respond appropriately.

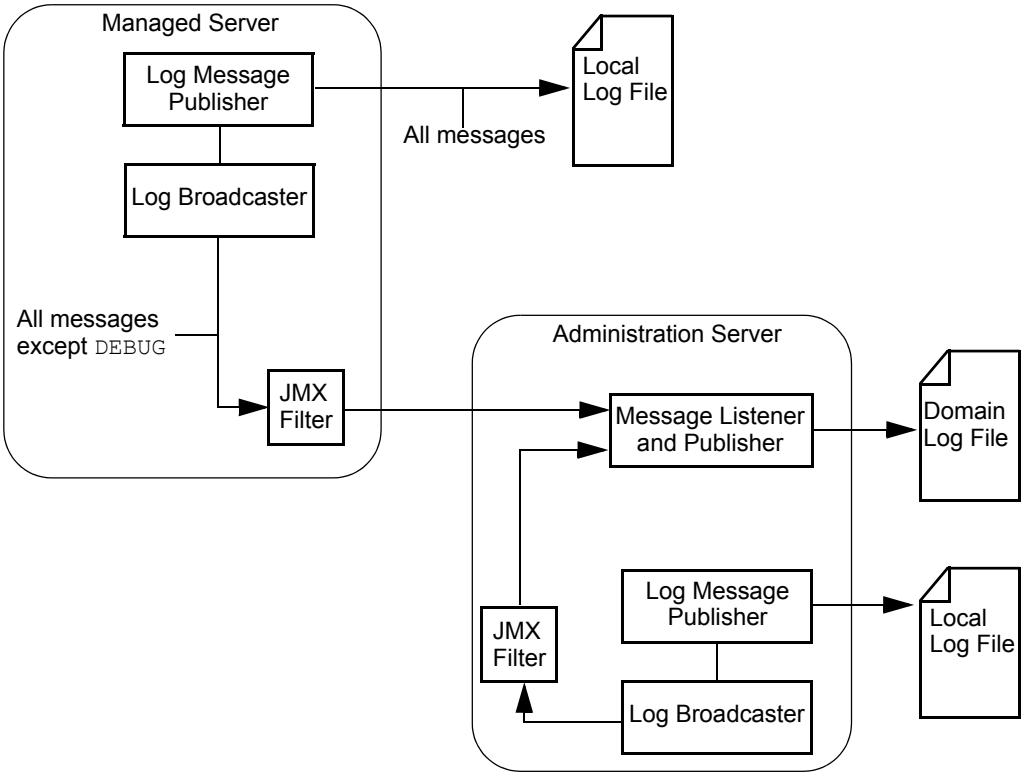
Because each WebLogic Server administration domain can run concurrent, multiple instances of WebLogic Server, the logging services collect messages that are generated on multiple server instances into a single, domain-wide message log. You can use this domain-wide message log to see the overall status of the domain.

To provide this overview of a domain's status, each server instance broadcasts its log messages as Java Management Extensions (JMX) notifications. A server broadcasts all messages and message text except for the following:

- Messages of the `DEBUG` severity level.
- Any stack traces that are included in a message.

The Administration Server listens for a subset of these messages and writes them to the domain log file. To listen for these messages, the Administration Server registers a JMX listener with each Managed Server. By default, the listener includes a filter that allows only messages of severity level `WARNING` and higher to be forwarded to the Administration Server. (See Figure 1-1.)

Figure 1-1 WebLogic Server Logging Services



The remainder of this document describes how your application can write, filter, and subscribe to messages, and how you can view messages through the WebLogic Server Administration Console.

CHAPTER 2

Writing Messages to the WebLogic Server Log

The following sections describe how you can facilitate the management of your application by writing log messages to the WebLogic Server log files:

In addition, this section includes the following sections:

Using the I18N Message Catalog Framework: Main Steps

The internationalization (I18N) message catalog framework provides a set of utilities and APIs that your application can use to send its own set of messages to the WebLogic Server log. The framework is ideal for applications that need to localize the language in their log messages, but even for those applications that do not need to localize, it provides a rich, flexible set of tools for communicating status and output.

To write log messages using the I18N message catalog framework, complete the following tasks:

Step 1: Create Message Catalogs

A message catalog is an XML file that contains a collection of text messages. Usually, an application uses one message catalog to contain a set of messages in a default language and optional, additional catalogs to contain messages in other languages.

To create and edit a properly formatted message catalog, use the WebLogic Message Editor utility, which is a graphical user interface (GUI) that is installed with WebLogic Server. To create corresponding messages in local languages, use the Message Localizer, which is also a GUI that WebLogic Server installs.

To access the Message Editor, do the following from a WebLogic Server host:

1. Set the classpath by entering *WL_HOME*`\server\bin\setWLSEnv.cmd` (`setWLSEnv.sh` on UNIX), where *WL_HOME* is the directory in which you installed WebLogic Server.
2. Enter the following command: `java weblogic.MsgEditor`
3. To create a new catalog, choose File→New Catalog.

   For information on using the Message Editor, refer to the following:

   – Using the BEA WebLogic Server Message Editor in the *BEA WebLogic Server Internationalization Guide*.

   – Using Message Catalogs with BEA WebLogic Server in the *BEA WebLogic Server Internationalization Guide*.
4. When you finish adding messages in the Message Editor, select File→Save Catalog. Then select File→Exit.

To access the Message Localizer, do the following from a WebLogic Server host:

1. Set the classpath by entering *WL_HOME*`\server\bin\setWLSEnv.cmd` (`setWLSEnv.sh` on UNIX), where *WL_HOME* is the directory in which you installed WebLogic Server.
2. Enter the following command: `java weblogic.MsgLocalizer`
3. Use the Message Localizer GUI to create locale-specific catalogs.

Step 2: Compile Message Catalogs

After you create message catalogs, you use the `i18ngen` and `l10ngen` command-line utilities to generate properties files and to generate and compile Java class files. The utilities take the message catalog XML files as input and create compiled Java classes. The Java classes contain methods that correspond to the messages in the XML files.

To compile the message catalogs, do the following:

1. From a command prompt, use *`WL_HOME`*`\server\bin\setWLSEnv.cmd` (`setWLSEnv.sh` on UNIX) to set the classpath, where *`WL_HOME`* is the directory in which you installed WebLogic Server.
2. Enter the following command:

   ```
   java weblogic.i18ngen -build -d targetdirectory source-files
   ```

   where:

   – *`targetdirectory`* is the root directory in which you want the `i18ngen` utility to locate the generated and compiled files. The Java files are placed in sub-directories based on the `i18n_package` and `l10n_package` values in the message catalog.

   The catalog properties file, `i18n_user.properties`, is placed in the *`targetdirectory`*. The default target directory is the current directory.

   – *`source-files`* specifies the message catalog files that you want to compile. If you specify one or more directory names, `i18ngen` processes all XML files in the listed directories. If you specify file names, the names of all files must include an XML suffix. All XML files must conform to the `msgcat.dtd` syntax.

   Note that when the `i18ngen` generates the Java files, it appends `Logger` to the name of each message catalog file.
3. If you created locale-specific catalogs in Step 1: Create Message Catalogs, do the following to generate properties files:

   a. In the current command prompt, add the *`targetdirectory`* that you specified in step 2. to the CLASSPATH environment variable. To generate locale-specific properties files, all of the classes that the `i18ngen` utility generated must be on the classpath.

   b. Enter the following command:
   ```
   java l10ngen -d targetdirectory source-files
   ```
   where:

- *targetdirectory* is the root directory in which you want the `l10ngen` utility to locate the generated properties files. Usually this is the same *targetdirectory* that you specified in step 2. The properties files are placed in sub-directories based on the `l10n_package` values in the message catalog.
- *source-files* specifies the message catalogs for which you want to generate properties files. You must specify top-level catalogs that the Message Editor creates; you do not specify locale-specific catalogs that the Message Localizer creates. Usually this is the same set of *source-files* or source directories that you specified in step 2.

4. In most cases, the recommended practice is to include the message class files and properties files in the same package hierarchy as your application.

   However, if you do not include the message classes and properties in the application's package hierarchy, you must make sure the classes are in the application's classpath.

For complete documentation of the `i18ngen` commands, refer to Using the BEA WebLogic Server Internationalization Utilities in the *BEA WebLogic Server Internationalization Guide*.

Example: Compiling Message Catalogs

In this example, the Message Editor created a message catalog that contains one message of type `loggable`. The Message Editor saves the message catalog as the following file: `c:\MyMsgCat\MyMessages.xml`.

Listing 2-1 shows the contents of the message catalog.

Listing 2-1 Sample Message Catalog

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls810/dtd/msgcat.dtd">

<message_catalog
   i18n_package="com.xyz.msgcat"
   l10n_package="com.xyz.msgcat.l10n"
   subsystem="MyClient"
   version="1.0"
   baseid="700000"
   endid="800000"
   loggables="true"
```

```
   prefix="XYZ-"
>
<!--  Welcome message to verify that the class has been invoked-->
   <logmessage
     messageid="700000"
     datelastchanged="1039193709347"
     datehash="-1776477005"
     severity="info"
     method="startup()"
>
      <messagebody>
         The class has been invoked.
         </messagebody>
      <messagedetail>
         Verifies that the class has been invoked
         and is generating log messages
      </messagedetail>
      <cause>
         Someone has invoked the class in a remote JVM.
      </cause>
      <action> </action>
   </logmessage>
</message_catalog>
```

In addition, the Message Localizer creates a Spanish version of the message in `MyMessages.xml`. The Message Localizer saves the Spanish catalog as `c:\MyMsgCat\es\ES\MyMessages.xml`.

Listing 2-2 Locale-Specific Catalog for Spanish

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE locale_message_catalog PUBLIC
"weblogic-locale-message-catalog-dtd"
"http://www.bea.com/servers/wls810/dtd/l10n_msgcat.dtd">
```

```
<locale_message_catalog
version="1.0"
>
<!-- Mensaje agradable para verificar que se haya invocado la clase. -->
<logmessage
    messageid="700000"
   datelastchanged="1039546411623"
   >
      <messagebody>
         La clase se haya invocado.
         </messagebody>
      <messagedetail>
         Verifica que se haya invocado la clase y está
         generando mensajes del registro.
      </messagedetail>
      <cause>Alguien ha invocado la clase en un JVM alejado.</cause>
      <action> </action>
   </logmessage>
</locale_message_catalog>
```

To compile the message catalog that the Message Editor created, enter the following command:

```
java weblogic.i18ngen -build -d c:\MessageOutput c:\MyMsgCat\MyMessages.xml
```

The `i18ngen` utility creates the following files:

- `c:\MessageOutput\i18n_user.properties`
- `c:\MessageOutput\com\xyz\msgcat\MyMessagesLogger.java`
- `c:\MessageOutput\com\xyz\msgcat\MyMessagesLogger.class`
- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizer.properties`
- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizerDetails.properties`

To create properties files for the Spanish catalog, you do the following:

1. Add the `i18n` classes to the command prompt's classpath by entering the following:
   ```
   set CLASSPATH=%CLASSPATH%;c:\MessageOutput
   ```
2. Enter
   ```
   java l10ngen -d c:\MessageOutput c:\MyMsgCat\MyMessages.xml
   ```

The `l10ngen` utility creates the following files:

- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizer_es_ES.properties`
- `c:\MessageOutput\com\xyz\msgcat\l10n\MyMessagesLogLocalizerDetails_es_ES.properties`

Step 3: Use Messages from Compiled Message Catalogs

The classes and properties files generated by `i18ngen` and `l10ngen` provide the interface for sending messages to the WebLogic Server log. Within the classes, each log message is represented by a method that your application calls.

To use messages from compiled message catalogs:

1. In the class files for your application, import the `Logger` classes that you compiled in Step 2: Compile Message Catalogs.

   To verify the package name, open the message catalog XML file in a text editor and determine the value of the `i18n_package` attribute. For example, the following segment of the message catalog in Listing 2-1 indicates the package name:

   ```
   <message_catalog
       i18n_package="com.xyz.msgcat"
   ```

   To import the corresponding class, add the following line:

   ```
   import com.xyz.msgcat.MyMessagesLogger;
   ```

2. Call the method that is associated with a message name.

   Each message in the catalog includes a `method` attribute that specifies the method you call the display the message. For example, the following segment of the message catalog in Listing 2-1 shows the name of the method:

   ```
   <logmessage
     messageid="700000"
     datelastchanged="1039193709347"
     datehash="-1776477005"
     severity="info"
   ```

```
    method="startup()"
>
```

Listing 2-3 illustrates a simple class that calls this `startup` method.

Listing 2-3 Example Class That Uses a Message Catalog

```
import com.xyz.msgcat.MyMessagesLogger;

public class MyClass {
     public static void main (String[] args) {
     MyMessagesLogger.startup();
     }
}
```

If the JVM's system properties specify that the current location is Spain, then the message is printed in Spanish.

Using the NonCatalogLogger APIs

In addition to using the I18N message catalog framework, your application can use the `weblogic.logging.NonCatalogLogger` APIs to send messages to the WebLogic Server log. With `NonCatalogLogger`, instead of calling messages from a catalog, you place the message text directly in your application code. BEA recommends that you do not use this facility as the sole means for logging messages if your application needs to be internationalized.

`NonCatalogLogger` is also intended for use by client code that is running in its own JVM (as opposed to running within a WebLogic Server JVM). A subsequent section in this topic, "Writing Messages from a Remote Application" on page 2-12, provides more information.

To use `NonCatalogLogger` in an application that runs within the WebLogic Server JVM, add code to your application that does the following:

1. Imports the `weblogic.logging.NonCatalogLogger` interface.
2. Uses the following constructor to instantiate a `NonCatalogLogger` object:

   `NonCatalogLogger(java.lang.String` *`myApplication`*`)`

where *myApplication* is a name that you supply to identify messages that your application sends to the WebLogic Server log.

3. Calls any of the `NonCatalogLogger` methods.

   Use the following methods to report normal operations:

   - `info(java.lang.String msg)`
   - `info(java.lang.String msg, java.lang.Throwable t)`

   Use the following methods to report a suspicious operation, event, or configuration that does not affect the normal operation of the server/application:

   - `warning(java.lang.String msg)`
   - `warning(java.lang.String msg, java.lang.Throwable t)`

   Use the following methods to report errors that the system/application can handle with no interruption and with limited degradation in service.

   - `error(java.lang.String msg)`
   - `error(java.lang.String msg, java.lang.Throwable t)`

   Use the following methods to provide detailed information about operations or the state of the application. These debug messages are not broadcast as JMX notifications. If you use this severity level, we recommend that you create a "debug mode" for your application. Then, configure your application to output debug messages only when the application is configured to run in the debug mode. For information about using debug messages, refer to "Writing Debug Messages" on page 2-12.

   - `debug(java.lang.String msg)`
   - `debug(java.lang.String msg, java.lang.Throwable t)`

All methods that take a `Throwable` argument can print the stack trace in the error log. For information on the `NonCatalogLogger` APIs, refer to the `weblogic.logging.NonCatalogLogger` Javadoc.

Listing 2-4 illustrates a servlet that uses `NonCatalogLogger` APIs to write messages of various severity levels to the WebLogic Server log.

Listing 2-4 Example NonCatalogLogger Messages

```
import java.io.PrintWriter;
import java.io.IOException;
```

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.naming.Context;

import weblogic.jndi.Environment;
import weblogic.logging.NonCatalogLogger;

public class MyServlet extends HttpServlet {

     public void service (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

       PrintWriter out = response.getWriter();
       NonCatalogLogger myLogger = null;

       try {

          out.println("Testing NonCatalogLogger. See
                         WLS Server log for output message.");

// Constructing a NonCatalogLogger instance. All messages from this
// instance will include a <MyApplication> string.
          myLogger = new NonCatalogLogger("MyApplication");

// Outputting an INFO message to indicate that your application has started.
          mylogger.info("Application started.");

// For the sake of providing an example exception message, the next
// lines of code purposefully set an initial context. If you run this
// servlet on a server that uses the default port number (7001), the
// servlet will throw an exception.
         Environment env = new Environment();
         env.setProviderUrl("t3://localhost:8000");

         Context ctx = env.getInitialContext();

      }

    catch (Exception e){
      out.println("Can't set initial context: " + e.getMessage());

// Prints a WARNING message that contains the stack trace.
     mylogger.warning("Can't establish connections. ", e);

    }

   }

}
```

When the servlet illustrated in the previous example runs on a server that specifies a listen port other than `8000`, the following messages are printed to the WebLogic Server log file. Note that the message consists of a series of strings, or fields, surrounded by angle brackets (< >).

Listing 2-5 NonCatalogLogger Output

```
####<Jun 26, 2002 12:04:21 PM EDT> <Info> <MyApplication> <MyHost>
<examplesServer> <ExecuteThread: '10' for queue: 'default'> <kernel identity>
<> <000000> <Application started.>

####<Jun 26, 2002 12:04:23 PM EDT> <Warning> <MyApplication> <MyHost>
<examplesServer> <ExecuteThread: '10' for queue: 'default'> <kernel identity>
<> <000000> <Can't establish connections. >

javax.naming.CommunicationException.  Root exception is
java.net.ConnectException: t3://localhost:8000: Destination unreachable; nested
exception is:

...
```

Using GenericServlet

The `javax.servlet.GenericServlet` servlet specification provides the following APIs that your servlets can use to write a simple message to the WebLogic Server log:

- `log(`*`java.lang.String msg`*`)`
- `log(`*`java.lang.String msg, java.lang.Throwable t`*`)`

For more information on using these APIs, refer to the J2EE Javadoc for `javax.servlet.GenericServlet` at http://java.sun.com/products/servlet/2.3/javadoc/javax/servlet/GenericServlet.html.

Writing Messages from a Remote Application

If your application runs in a JVM that is separate from a WebLogic Server, it can use message catalogs and `NonCatalogLogger`, but the messages are not written to a WebLogic Server log. Instead, the application's messages are written to the remote JVM's standard out.

If you want the WebLogic logging service to send these messages to a log file that the remote JVM maintains, include the following argument in the command that starts the remote JVM:

```
-Dweblogic.log.FileName=logfilename
```

where *`logfilename`* is the name that you want to use for the remote log file.

If you want a subset of the message catalog and `NonCatalogLogger` messages to go to standard out as well as the remote JVM log file, include the following additional startup arguments:

```
-Dweblogic.StdoutEnabled=true
```

```
-Dweblogic.StdoutDebugEnabled=boolean
```

```
-Dweblogic.StdoutSeverityLevel = [64 | 32 | 16 | 8 | 4 | 2 | 1 ]
```

where *`boolean`* is either `true` or `false` and the numeric values for `StdoutSeverityLevel` correspond to the following severity levels:

INFO(64) WARNING(32), ERROR(16), NOTICE(8), CRITICAL(4), ALERT(2) and EMERGENCY(1).

Writing Messages from a Remote JVM to a File

A remote JVM can generate its own set of messages that communicate information about the state of the JVM itself. By default, the JVM sends these messages to standard out. You cannot redirect these messages to the JVM's log file, but you can save them to a separate file. For more information, refer to "Redirecting System.out and System.err to a File" in the Administration Console Online Help.

Writing Debug Messages

While your application is under development, you might find it useful to create and use messages that provide verbose descriptions of low-level activity within the application. You can use the `DEBUG` severity level to categorize these low-level messages. All `DEBUG` messages that your application generates are sent to the WebLogic Server log file. (Unlike Log4j, which is a third-party logging service that enables you to dynamically exclude log messages based on level

of severity, the WebLogic Server log includes all levels of messages that your application generates.)

You also can configure the WebLogic Server to send DEBUG messages to standard out. For more information refer to "Specifying Which Messages a Server Sends to Standard Out" in the *Administration Console Online Help*.

If you use the DEBUG severity level, we recommend that you create a "debug mode" for your application. For example, your application can create an object that contains a boolean value. To enable or disable the debug mode, you toggle the value of the boolean. Then, for each DEBUG message, you can create a wrapper that outputs the message only if your application's debug mode is enabled.

For example, the following code can produce a debug message:

```
private static boolean debug = Boolean.getBoolean("my.debug.enabled");
if (debug) {
   mylogger.debug("Something debuggy happened");
}
```

You can use this type of wrapper both for messages that use the message catalog framework and that use the `NonCatalogLogger` API.

To enable your application to print this message, you include the following Java option when you start the application's JVM:

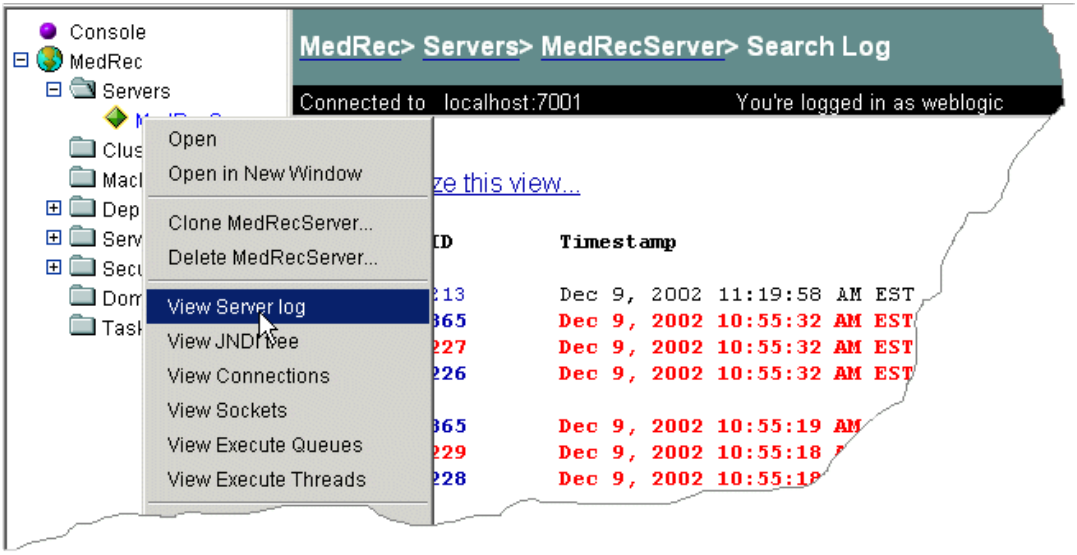
```
-Dmy.debug.enabled=true
```

CHAPTER 3

Viewing the WebLogic Server Logs

The WebLogic Server Administration Console provides separate but similar log viewers for the local server log and the domain-wide message log. The log viewer can search for messages based on fields within the message. For example, it can find and display messages based on the severity, time of occurrence, user ID, subsystem, or the short description. It can also display messages as they are logged, or search for past log messages. (See Figure 3-1.)

Figure 3-1 Log Viewer



In addition to viewing messages from the Administration Console, you can specify which messages are sent to standard out. By default, only messages of `WARNING` or higher are sent to standard out.

For information about viewing, configuring, and searching message logs, refer to the following topics:

- "Viewing Server Logs" in the Administration Console Online Help
- "Specifying Which Messages a Server Sends to Standard Out" in the Administration Console Online Help
- "Viewing the Domain Log" in the Administration Console Online Help
- "Overview of WebLogic Server Log Messages and Log Files" in the Administration Console Online Help

CHAPTER 4

Filtering WebLogic Server Log Messages

The WebLogic Server logging services provide several levels of filtering that give you the flexibility to determine which messages are written to the WebLogic Server log files and standard out, and to the log file and standard out that a client JVM maintains. Most of these filtering features are implementations of the JDK 1.4 logging APIs, which are available in the `java.util.logging` package.

The following sections describe how to filter messages that the WebLogic Server logging services generates:

Note: Prior to WebLogic Server 8.1, filtering was available only for the domain-wide log file and for any Java Management Extensions (JMX) listeners that you registered with the WebLogic Server logging services. For example, you could use a filter to determine which messages the Administration Server wrote to the domain-wide message log, but you could not use a filter to determine which messages a server instance wrote to its local log file. Server instances always wrote all messages to their local log files. With the introduction of the JDK 1.4 logging APIs in WebLogic Server 8.1, you can now create separate filters for the messages that each server instance writes to its local log file, to its standard out, or that it broadcasts to the domain-wide message log.

Overview of Distributing and Filtering Messages

When WebLogic Server message catalogs and the `NonCatalogLogger` generate messages, they distribute their messages to a `java.util.logging.Logger` object. The `Logger` object publishes the messages to any message handler that has subscribed to the `Logger`.

WebLogic Server instantiates `Logger` and `Handler` objects in three distinct contexts (See Figure 4-1):

- In client JVMs that use WebLogic Server logging services. This client `Logger` object publishes messages that are sent from client applications running in the client JVM.

  The following handlers subscribe to the `Logger` object in a client JVM:

  - `ConsoleHandler`, which prints messages from the client JVM to the client's standard out.

    If you use the `-Dweblogic.StdoutSeverityLevel` Java startup option for the client JVM, WebLogic logging services create a filter for this handler that limits the messages that the handler writes to standard out. For more information, refer to "Writing Messages from a Remote Application" on page 2-12.

  - `FileStreamHandler`, which writes messages from the client JVM to the client's log file. For information about configuring log file for a client JVM, refer to "Writing Messages from a Remote JVM to a File" on page 2-12.

- In each instance of WebLogic Server. This server `Logger` object publishes messages that are sent from subsystems and applications that run on a server instance.

  The following handlers subscribe to the domain `Logger` object:

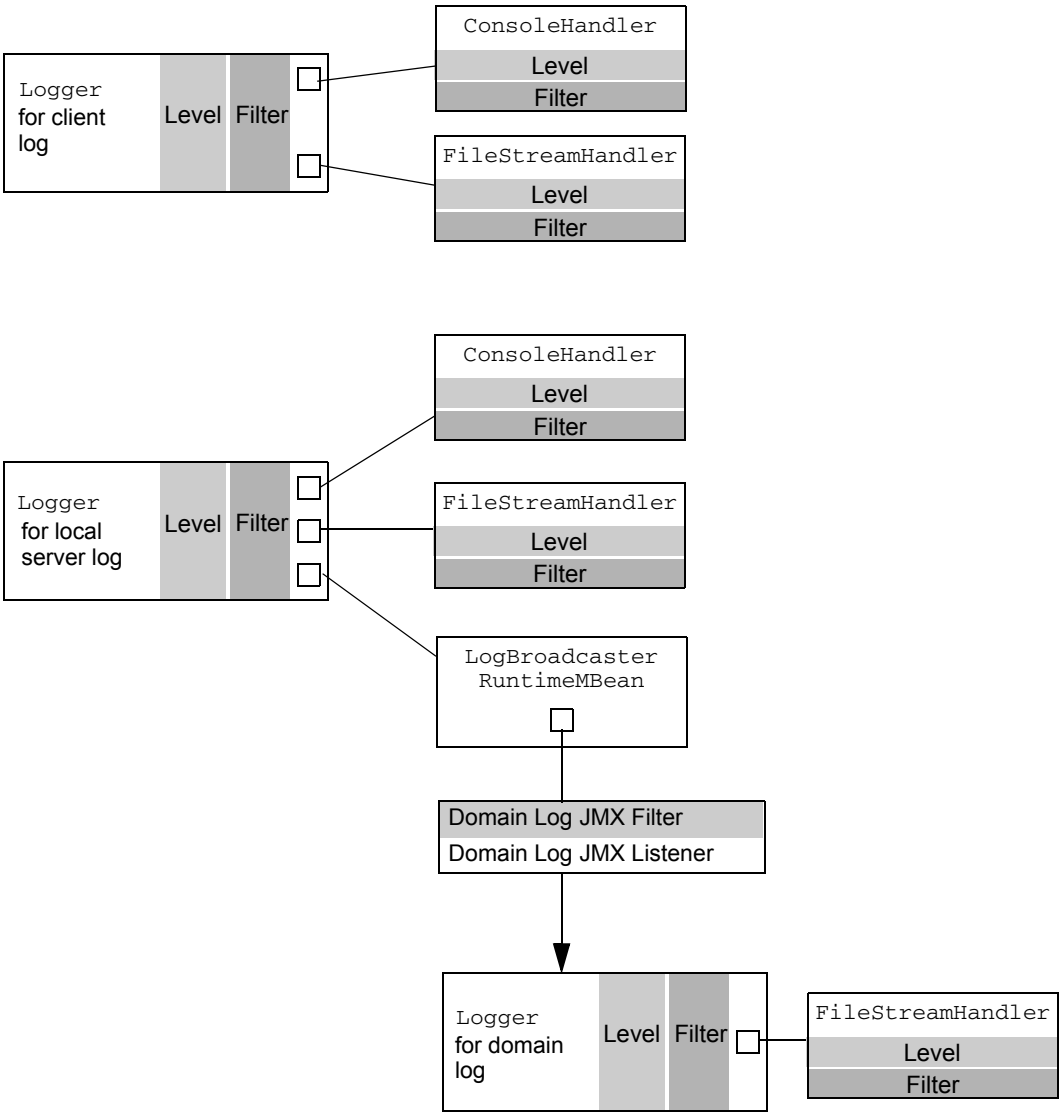
  - `ConsoleHandler`, which makes messages available to the server's standard out.
  - `FileStreamHandler`, which writes messages to the server's local log file.
  - The `LogBroadcasterRuntimeMBean`, which publishes to the domain `Logger` object on the Administration Server.

- The Administration Server maintains a domain `Logger` object in addition to a server `Logger` object. The domain `Logger` object receives messages from each server's `LogBroadcasterRuntimeMBean` handler.

  The following handler subscribes to the domain `Logger` object:

  - `FileStreamHandler`, which writes messages to the domain's log file.

Figure 4-1 JDK 1.4 and WebLogic Server Logging Services



Setting the Level and Filters

When WebLogic Server message catalogs and the `NonCatalogLogger` generate messages, they convert the message severity to a `weblogic.logging.WLLevel` object. A `WLLevel` object can specify any of the following values, from lowest to highest impact:

```
DEBUG, INFO, WARNING, ERROR, NOTICE, CRITICAL, ALERT, EMERGENCY
```

By default, a `Logger` object publishes messages of all levels. To set the lowest-level message that a `Logger` object publishes, you use a simple `Logger.setLevel` API. When a `Logger` object receives an incoming message, it checks the message level with the level set by the `setLevel` API. If the message level is below the `Logger` level, it returns immediately. If the message level is above the `Logger` level, the `Logger` allocates a `WLLogRecord` object to describe the message.

For example, if you set a `Logger` object's level to `WARNING`, the `Logger` object publishes only `WARNING, ERROR, NOTICE, CRITICAL, ALERT`, or `EMERGENCY` messages.

To provide more control over the messages that a `Logger` object publishes, you can also create and set a filter. A filter is a class that compares data in the `WLLogRecord` object with a set of criteria. The `Logger` object publishes only the `WLLogRecord` objects that satisfy the filter criteria. For example, a filter can configure a `Logger` to publish only messages from the JDBC subsystem. To create a filter, you instantiate a `java.util.logging.Filter` object and use the `Logger.setFilter` API to set it for a `Logger` object.

Instead of (or in addition to) setting the level and a filter for the messages that a `Logger` object **publishes**, you can set the level and filters on individual message handlers.

For example, you can specify that a `Logger` publishes messages that are of the `WARNING` level or higher. Then you can do the following for each handler:

- For the `ConsoleHandler`, set a level and filter that selects only `ALERT` messages from the JDBC, JMS, and EJB subsystems. This causes standard out to display only `ALERT` messages from the JDBC, JMS, and EJB subsystems.
- For the `FileHandler`, set no additional level or filter criteria. Because the `Logger` object has been configured to publish only messages of the `WARNING` level or higher, the log file will contain all messages from all subsystems that are of the `WARNING` level or higher.
- The `LogBroadcasterRuntimeMBean` will publish all messages of the `WARNING` level or higher to the `Logger` object that the Administration Server uses to maintain the domain-wide message log.

Domain Log Filters

To filter the messages that the `LogBroadcasterRuntimeMBean` publishes, you can use the Administration Console to create a Domain Log Filter. Unlike the filters for `Logger` and `Handler` objects, a Domain Log Filter is implemented in JMX and is registered only with the `LogBroadcasterRuntimeMBean` that a server uses to publish messages to the domain `Logger` (in Figure 4-1, this filter is labeled **Domain Log JMX Filter**).

Any JDK 1.4 level or filter that you set on the `Logger` object that manages a server instance's local log file **supersedes** a Domain Log Filter. For example, if the level of the server's `Logger` object is set to `WARNING`, a Domain Log Filter will receive only messages of the `WARNING` level or higher.

Setting the Level for Loggers and Handlers

The Administration Console and the `weblogic.Admin` utility do not provide a way to set JDK 1.4 level and filters. You must use the `Logger`, `ConsoleHandler`, and `FileStreamHandler` APIs.

Setting the Level for Loggers

To set the level for a `Logger` object, create a class that does the following:

1. Invokes one of the following `LoggingHelper` methods:
   - `getClientLogger` if the current context is a client JVM.
   - `getServerLogger` if the current context is a server JVM and you want to retrieve the `Logger` object that a server uses to manage its local server log.
   - `getDomainLogger` if the current context is the Administration Server and you want to retrieve the `Logger` object that manages the domain log.

   The `LoggerHelper` method returns a `Logger` object. For more information, refer to the Sun API documentation for `Logger`:
   http://java.sun.com/j2se/1.4/docs/api/java/util/logging/Logger.html
2. Invokes the `Logger.setLevel(WLevel level)` method.

   To set the level of a WebLogic Server `Logger` object, you must pass a value that is defined in the `weblogic.logging.WLLevel` class. For a list of valid values, refer to the `WLLevel` Javadoc.

   For example:

```
setLevel(WLLevel.ALERT)
```

Setting the Level for Handlers

To set the level for a `Handler` object, create a class that does the following (see Listing 4-1):

1. Invokes one of the following `LoggingHelper` methods:
   - `getClientLogger` if the current context is a client JVM.
   - `getServerLogger` if the current context is a server JVM and you want to retrieve the `Logger` object that a server uses to manage its local server log.
   - `getDomainLogger` if the current context is the Administration Server and you want to retrieve the `Logger` object that manages the domain log.

   The `LoggerHelper` method returns a `Logger` object. For more information, refer to the Sun API documentation for `Logger`:
   http://java.sun.com/j2se/1.4/docs/api/java/util/logging/Logger.html
2. Invokes the `Logger.getHandlers()` method.

   The method returns an array of all handlers that are registered with the `Logger` object.
3. Iterates through the list of handlers until it finds the `Handler` object for which you want to set a level.

   Use `Handler.getClass().getName()` to determine the type of handler to which the current array index refers.
4. Invokes the `Handler.setLevel(WLevel level)` method.

   To set the level of a WebLogic Server `Handler` object, you must pass a value that is defined in the `weblogic.logging.WLLevel` class. For a list of valid values, refer to the `WLLevel` Javadoc.

   For example:

   ```
   setLevel(WLLevel.ALERT)
   ```

Listing 4-1 Example: Setting Level for a Handler Object

```
import java.util.logging.Logger;
import java.util.logging.Handler;
```

```
import weblogic.logging.LoggingHelper;
import weblogic.logging.WLLevel;

public class LogLevel {

    public static void main(String[] argv) throws Exception {

        Logger serverlogger = LoggingHelper.getServerLogger();
        Handler[] handlerArray = serverlogger.getHandlers();
        for (int i=0; i < handlerArray.length; i++) {
            Handler h = handlerArray[i];
            if(h.getClass().getName().equals
                      (“weblogic.logging.ConsoleHandler”)){
                h.setLevel(WLLevel.getLevel( WLLevel.ALERT) );
            }
        }
    }
}
```

Setting a Filter for Loggers and Handlers

When you set a filter on the `Logger` object, the filter specifies which messages the object publishes; therefore, the filter affects all handlers that are registered with the `Logger` object. When you set a filter on a handler, the filter affects only the behavior of the specific handler.

To set a filter:

1. Create a class that implements `java.util.logging.Filter`. See Listing 4-2.

   The class must include the `Filter.isLoggable` method and logic that evaluates incoming messages. If the logic evaluates as true, the `isLoggable` method enables the `Logger` object to publish the message.

2. Place the filter object in the classpath of the JVM on which the `Logger` object is running.

3. To set a filter for a `Logger` object, create a class that does the following:

   a. Invokes one of the following `LoggingHelper` methods:

   – `getClientLogger` if the current context is a client JVM.

   – `getServerLogger` if the current context is a server JVM and you want to filter the `Logger` object that a server uses to manage its local server log.

   – `getServerLogger` if the current context is the Administration Server and you want to filter the `Logger` object that manages the domain server log.

b. Invokes the `Logger.setFilter(Filter newFilter)` method.

To set a filter for a `Handler` object, create a class that does the following:

a. Invokes one of the following `LoggingHelper` methods:

- `getClientLogger` if the current context is a client JVM.
- `getServerLogger` if the current context is a server JVM and you want to filter the `Logger` object that a server uses to manage its local server log.
- `getServerLogger` if the current context is the Administration Server and you want to filter the `Logger` object that manages the domain server log.

b. Iterates through the list of handlers until it finds the `Handler` object for which you want to set a level.

Use `Handler.getClass().getName()` to determine the type of handler to which the current array index refers.

c. Invokes the `Handler.setFilter(Filter newFilter)` method.

Listing 4-2 provides an example class that rejects all messages from the Deployer subsystem.

Listing 4-2 Example Filter for a JDK 1.4 Logger Object

```
import java.util.logging.Logger;
import java.util.logging.Filter;
import java.util.logging.LogRecord;

import weblogic.logging.WLLogRecord;
import weblogic.logging.WLLevel;

public class MyFilter implements Filter {
    public boolean isLoggable(LogRecord record) {
        if (record instanceof WLLogRecord) {
            WLLogRecord rec = (WLLogRecord)record;
            if (rec.getLoggerName().equals("Deployer")) {
              return false;
            } else {
              return true;
            }
        } else {
          return false;
```

```
            }
        }
    }
```

CHAPTER 5

Subscribing to Messages

When WebLogic Server message catalogs and the `NonCatalogLogger` generate messages, they distribute their messages to a `java.util.logging.Logger` object. The `Logger` object allocates a `WLLogRecord` object to describe the message and publishes the `WLLogRecord` to any message handler that has subscribed to the `Logger`.

WebLogic Server instantiates and subscribes a set of messages handlers that receive and print log messages. You can also create your own message handlers and subscribe them to the WebLogic Server `Logger` objects (see Figure 5-1).

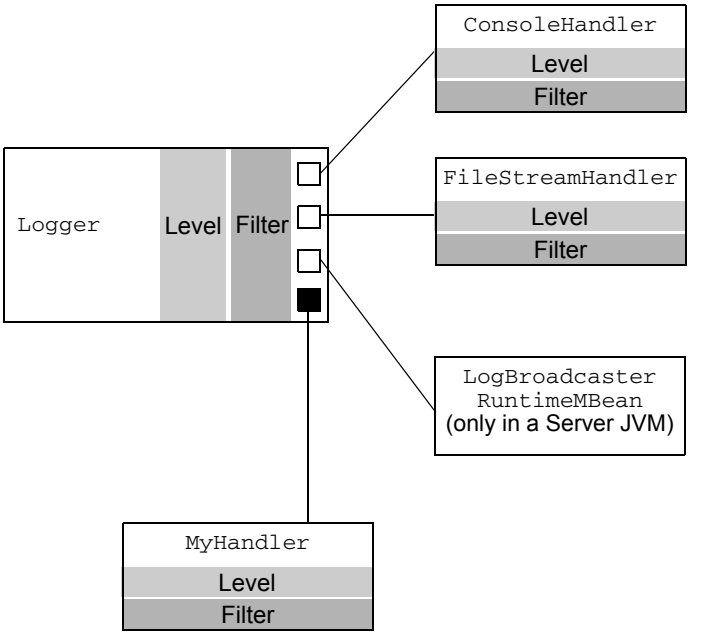
For example, if your application runs in a client JVM and you want the application to listen for the messages that your application generates, you can create a handler and subscribe it to the `Logger` object in the client JVM. If your application receives a log message that signals the failure of a specific subsystem, it can perform actions such as:

- E-mail the log message to the WebLogic Server administrator.
- Shut down or restart itself or its subcomponents.

The following sections describe creating and subscribing a message handler:

For more information about WebLogic Server loggers and handlers, refer to "Overview of Distributing and Filtering Messages" on page 4-2.

Figure 5-1 Subscribing a Handler



Creating and Subscribing a Handler: Main Steps

A handler that you create and subscribe to a `Logger` object receives all messages that satisfy the level and filter criteria of the logger. Your handler can specify additional level and filter criteria so that it responds only to a specific set of messages that the logger publishes.

To create and subscribe a handler:

1. Create a handler class that includes the following minimal set of import statements:

```
import java.util.logging.Handler;
import java.util.logging.LogRecord;
import java.util.logging.ErrorManager;

import weblogic.logging.WLLogRecord;
import weblogic.logging.WLLevel;
import weblogic.logging.WLErrorManager;
import weblogic.logging.LoggingHelper;
```

2. In the handler class, extend `java.util.logging.Handler`.

3. In the handler class, implement the `Handler.publish(LogRecord record)` method.

   This method:

   a. Casts the `LogRecord` objects that it receives as `WLLogRecord` objects.

   b. Applies any filters that have been set for the handler.

   c. If the `WLLogRecord` object satisfies the criteria of any filters, the method uses `WLLogRecord` methods to retrieve data from the messages.

   d. Optionally writes the message data to one or more resources.

4. In the handler class, implement the `Handler.flush` and `Handler.close` methods.

   All handlers that work with resources should implement the `flush` method so that it flushes any buffered output and the `close` method so that it closes any open resources.

   When the parent `Logger` object shuts down, it calls the `Handler.close` method on all of its handlers. The close method calls the `flush` method and then executes its own logic.

5. Create a filter class that specifies which types of messages your `Handler` object should receive. For more information, refer to "Setting a Filter for Loggers and Handlers" on page 4-7.

6. Place the handler and filter objects in the classpath of the JVM on which the `Logger` object is running.

7. Create a class that invokes one of the following `LoggingHelper` methods:

   – `getClientLogger` if the current context is a client JVM.

   – `getServerLogger` if the current context is a server JVM and you want to filter the `Logger` object that a server uses to manage its local server log.

   – `getDomainLogger` if the current context is the Administration Server and you want to filter the `Logger` object that manages the domain server log.

     `LoggingHelper.getDomainLogger()` retrieves the `Logger` object that manages the domain log. You can subscribe a custom handler to this logger and process log messages from all the servers in a single location.

8. In this class, invoke the `Logger.addHandler(Handler myHandler)` method. Then invoke the `Logger.setFilter(Filter myFilter)` method.

Example: Subscribing to Messages in a Server JVM

This example creates a handler that connects to a JDBC data source and issues SQL statements that insert messages into a database table. The example implements the following classes:

- A `Handler` class. See "Example: Implementing a Handler Class" on page 5-4.
- A `Filter` class. See "Setting a Filter for Loggers and Handlers" on page 4-7.
- A class that subscribes the handler and filter to a server's `Logger` class. See "Example: Subscribing to a Logger Class" on page 5-8.

Example: Implementing a Handler Class

The example `Handler` class in Listing 5-1 writes messages to a database by doing the following:

1. Extends `java.util.logging.Handler`.
2. Constructs a `javax.naming.InitialContext` object and invokes the `Context.lookup` method to look up a data source named `myPoolDataSource`.
3. Invokes the `javax.sql.DataSource.getConnection` method to establish a connection with the data source.
4. Implements the `setErrorManager` method, which constructs a `java.util.logging.ErrorManager` object for this handler.

   If this handler encounters any error, it invokes the error manager's `error` method. The `error` method in this example:

   a. Prints an error message to standard error.

   b. Disables the handler by invoking `LoggingHelper.getServerLogger().removeHandler(MyJDBCHandler.this)`.

   Note: Instead of defining the `ErrorManager` class in a separate class file, the example includes the `ErrorManager` as an anonymous inner class.

   For more information about error managers, refer to the Sun API documentation for `java.util.logging.ErrorManager`.
5. Implements the `Handler.publish(LogRecord record)` method. The method does the following:

   a. Casts each `LogRecord` object that it receives as a `WLLogRecord` objects.

b. Calls an `isLoggable` method to apply any filters that are set for the handler. The `isLoggable` method is defined at the end of this handler class.

c. Uses `WLLogRecord` methods to retrieve data from the messages.

   For more information about `WLLogRecord` methods, refer to the `WLLogRecord` Javadoc.

d. Formats the message data as a SQL `prepareStatement` and executes the database update.

   The schema for the table used in the example is as follows:

Table 5-1 Schema for Database Table in Handler Example

Name	Null?	Type
MSGID		CHAR(25)
LOGLEVEL		CHAR(25)
SUBSYSTEM		CHAR(50)
MESSAGE		CHAR(1024)

e. Invokes a `flush` method to flush the connection.

6. Implements the `Handler.close` method to close the connection with the data source.

   When the parent `Logger` object shuts down, it calls the `Handler.close` method, which calls the `Handler.flush` method before executing its own logic.

Listing 5-1 Example: Implementing a Handler Class

```
import java.util.logging.Handler;
import java.util.logging.LogRecord;
import java.util.logging.Filter;
import java.util.logging.ErrorManager;

import weblogic.logging.WLLogRecord;
import weblogic.logging.WLLevel;
import weblogic.logging.WLErrorManager;
```

```
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.PreparedStatement;
import weblogic.logging.LoggingHelper;

public class MyJDBCHandler extends Handler {

   private Connection con = null;

   public MyJDBCHandler() throws NamingException, SQLException {

      InitialContext ctx = new InitialContext();
      DataSource ds = (DataSource)ctx.lookup("myPoolDataSource");
      con = ds.getConnection();
      setErrorManager(new ErrorManager() {
          public void error(String msg, Exception ex, int code) {
              System.err.println("Error reported by MyJDBCHandler "
                                 + msg + ex.getMessage());
              //Removing any prior istantiation of this handler
              LoggingHelper.getServerLogger().removeHandler(
                                 MyJDBCHandler.this);
          }
      });
   }

   public void publish(LogRecord record) {
      WLLogRecord rec = (WLLogRecord)record;
      if (!isLoggable(rec)) return;
      try {
          PreparedStatement stmt = con.prepareStatement(
          "INSERT INTO myserverLog VALUES (?, ?, ? ,?)");
          stmt.setEscapeProcessing(true);
          stmt.setString(1, rec.getId());
          stmt.setString(2, rec.getLevel().getLocalizedName());
          stmt.setString(3, rec.getLoggerName());
          stmt.setString(4, rec.getMessage());
```

```
            stmt.executeUpdate();
            flush();
        } catch(SQLException sqex) {
            reportError("Error publihsing to SQL", sqex,
                                ErrorManager.WRITE_FAILURE);
        }
    }
    public void flush() {
        try {
            con.commit();
        } catch(SQLException sqex) {
            reportError("Error flushing connection of MyJDBCHandler",
                                sqex, ErrorManager.FLUSH_FAILURE);
        }
    }
     public boolean isLoggable(LogRecord record) {
         Filter filter = getFilter();
         if (filter != null) {
             return filter.isLoggable(record);
         } else {
            return true;
         }
     }
    public void close() {
        try {
            con.close();
        } catch(SQLException sqex) {
             reportError("Error closing connection of MyJDBCHandler",
                                sqex, ErrorManager.CLOSE_FAILURE);
        }
    }
}
```

Example: Subscribing to a Logger Class

The example class in Listing 5-2 does the following:

1. Invokes the `LoggingHelper.getServerLogger` method to retrieve the `Logger` object.
2. Invokes the `Logger.addHandler(Handler myHandler)` method.
3. Invokes the `Logger.getHandlers` method to retrieve all handlers of the `Logger` object.
4. Iterates through the array until it finds `myHandler`.
5. Invokes the `Handler.setFilter(Filter myFilter)` method.

If you wanted your handler and filter to subscribe to the server's `Logger` object each time the server starts, you could deploy this class as a WebLogic Server startup class. For information about startup classes, refer to "Startup and Shutdown Classes" in the Administration Console Online Help.

Listing 5-2 Example: Subscribing to a Logger Class

```
import java.util.logging.Logger;
import java.util.logging.Handler;
import java.util.logging.Filter;
import java.util.logging.LogRecord;
import weblogic.logging.LoggingHelper;
import weblogic.logging.FileStreamHandler;
import weblogic.logging.WLLogRecord;
import weblogic.logging.WLLevel;
import java.rmi.RemoteException;
import weblogic.jndi.Environment;

import javax.naming.Context;

public class LogConfigImpl {

    public void configureLogger() throws RemoteException {
        Logger logger = LoggingHelper.getServerLogger();
        try {
            Handler h = null;
            h = new MyJDBCHandler();
            logger.addHandler(h);
```

```
            h.setFilter(new MyFilter());
        } catch(Exception nmex) {
            System.err.println("Error adding MyJDBCHandler to logger "
                               + nmex.getMessage());
            logger.removeHandler(h);
        }
    }

    public static void main(String[] argv) throws Exception {
        LogConfigImpl impl = new LogConfigImpl();
        impl.configureLogger();
    }

}
```

Comparison of JDK 1.4 Handlers with JMX Listeners

Prior to WebLogic Server 8.1, the only technique for receiving messages from the WebLogic logging services was to create a Java Management Extensions (JMX) listener and register it with a `LogBroadcasterRuntimeMBean`. With the release of WebLogic Server 8.1, you can now use JDK 1.4 handlers to receive (subscribe to) log messages.

While both techniques—JDK 1.4 handlers and JMX listeners—provide similar results, the JDK 1.4 APIs include a `Formatter` class that a `Handler` object can use to format the messages that it receives. JMX does not offer similar APIs for formatting messages. For more information about formatters, refer to the Sun API documentation for `Formatter`: http://java.sun.com/j2se/1.4/docs/api/java/util/logging/Formatter.html.

In addition, the JDK 1.4 `Handler` APIs are easier to use and require fewer levels of indirection than JMX APIs. For example, the following lines of code retrieve a JDK 1.4 `Logger` object and subscribe a handler to it:

```
Logger logger = LoggingHelper.getServerLogger();
Handler h = null;
h = new MyJDBCHandler();
logger.addHandler(h)
```

To achieve a similar result by registering a JMX listener, you must use lines of code similar to Listing 5-3. The code looks up the `MBeanHome` interface, looks up the `RemoteMBeanServer` interface, looks up the `LogBroadcasterRuntimeMBean`, and then registers the listener.

For information on using JMX listeners, refer to "Using WebLogic Server MBean Notifications and Monitors" in *Programming WebLogic Management Services with JMX*.

Listing 5-3 Registering a JMX Listener

```
MBeanHome home = null;
RemoteMBeanServer rmbs = null;

//domain variables
String url = "t3://localhost:7001";
String serverName = "Server1";
String domainName= "mydomain"
String username = "weblogic";
String password = "weblogic";

//Using MBeanHome to get MBeanServer.
try {
    Environment env = new Environment();
    env.setProviderUrl(url);
    env.setSecurityPrincipal(username);
    env.setSecurityCredentials(password);
    Context ctx = env.getInitialContext();

    //Getting the Administration MBeanHome.
    home = (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
    System.out.println("Got the Admin MBeanHome: " + home );
    rmbs = home.getMBeanServer();
} catch (Exception e) {
    System.out.println("Caught exception: " + e);
}

try {
    //Instantiating your listener class.
    MyListener listener = new MyListener();
    MyFilter filter = new MyFilter();

    //Construct the WebLogicObjectName of the server's
    //log broadcaster
    WebLogicObjectName logBCOname = new
            WebLogicObjectName("TheLogBroadcaster",
           "LogBroadcasterRuntime", domainName, serverName);
    //Passing the name of the MBean and your listener class to the
    //addNotificationListener method of MBeanServer.
    rmbs.addNotificationListener(logBCOname, listener, filter, null);
    } catch(Exception e) {
        System.out.println("Exception: " + e);
```

```
    }
}
```

Index

M

N

O

P

R

S

V

W

X