BEA WebLogic Server®

Configuring Log Files and Filtering Log Messages

Version 10.0
Revised: March 30, 2007

Contents

Introduction and Roadmap

Document Scope and Audience 1-1

Guide to This Document 1-2

Related Documentation 1-2

Logging Samples and Tutorials 1-2

Avitek Medical Records Application (MedRec) and Tutorials 1-3

Log4j Integration in MedRec 1-3

Logging Examples in the WebLogic Server Distribution 1-3

New and Changed Logging Features in This Release 1-3

Understanding WebLogic Logging Services

What You Can Do With WebLogic Logging Services 2-1

How WebLogic Logging Services Work 2-2

Components and Environment 2-2

Terminology 2-3

Overview of the Logging Process 2-3

Best Practices: Integrating Java Logging or Log4j with WebLogic Logging Services . . 2-5

Server Log Files and Domain Log Files 2-5

How a Server Instance Forwards Messages to the Domain Log 2-6

Server and Subsystem Logs 2-8

Server Log 2-8

Subsystem Logs 2-9

Log Message Format 2-11
    Format of Output to Standard Out and Standard Error 2-11
Message Attributes 2-12
Message Severity. 2-13
Viewing WebLogic Server Logs. 2-15

Configuring WebLogic Logging Services

Configuration Scenarios 3-1
Overview of Logging Services Configuration 3-2
    Using Log Severity Levels. 3-3
    Using Log Filters 3-3
Logging Configuration Tasks: Main Steps 3-4
Log4j and the Commons Logging API 3-5
    About Log4j 3-5
        Loggers 3-5
        Appenders 3-5
        Layouts 3-6
How to Use Log4j with WebLogic Logging Services 3-6
    Enabling Log4j Logging 3-6
How to Use the Commons API with WebLogic Logging Services 3-9
Rotating Log Files. 3-11
    Specifying the Location of Archived Log Files. 3-12
    Notification of Rotation 3-13
Redirecting JVM Output 3-13

Filtering WebLogic Server Log Messages

The Role of Logger and Handler Objects 4-2
Filtering Messages by Severity Level or Other Criteria 4-3

Setting the Severity Level for Loggers and Handlers 4-4
  Setting the Level for Loggers 4-5
  Setting the Level for Handlers 4-5
Setting a Filter for Loggers and Handlers 4-7
  Filtering Domain Log Messages 4-10
Setting a Severity Level and Filter on a Log4j Appender 4-10

Subscribing to Messages

Overview of Message Handlers 5-1
Creating and Subscribing a Handler: Main Steps 5-3
Example: Subscribing to Messages in a Server JVM 5-4
  Example: Implementing a Handler Class 5-4
  Example: Subscribing to a Logger Class 5-7
Example: Implementing a Log4j Appender Class 5-9
Comparison of Java Logging Handlers with JMX Listeners. 5-12

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Configuring Log Files and Filtering Log Messages*.

- "Document Scope and Audience" on page 1-1
- "Guide to This Document" on page 1-2
- "Related Documentation" on page 1-2
- "Logging Samples and Tutorials" on page 1-2
- "New and Changed Logging Features in This Release" on page 1-3

Document Scope and Audience

This document describes how you use BEA WebLogic Server® logging services to monitor server, subsystem, and application events. It explains how you configure WebLogic Server to write messages to log files and listen for the log messages that WebLogic Server broadcasts, and describes how to view log messages through the WebLogic Server Administration Console.

This document is a resource for system administrators who configure WebLogic logging services and monitor server and subsystem events, and for Java Platform, Enterprise Edition (Java EE) application developers who want to integrate their application logs with WebLogic Server logs. This document is relevant to all phases of a software project, from development through test and production phases.

This document does not address application logging or localization and internationalization of log message catalogs. For links to information on these topics, see Related Documentation.

It is assumed that the reader is familiar with Java EE and Web technologies, object-oriented programming techniques, and the Java programming language.

Guide to This Document

The document is organized as follows:

- This chapter, "Introduction and Roadmap," describes the scope of this guide and lists related documentation.
- Chapter 2, "Understanding WebLogic Logging Services," discusses the logging process, log files, and log messages.
- Chapter 3, "Configuring WebLogic Logging Services," describes basic configuration scenarios and tasks.
- Chapter 4, "Filtering WebLogic Server Log Messages," describes how to specify which types of messages WebLogic Server writes to its logs and to standard out.
- Chapter 5, "Subscribing to Messages," describes how WebLogic Server instantiates and subscribes a set of message handlers that receive and print log messages.

Related Documentation

The BEA corporate Web site provides all documentation for WebLogic Server. Specifically, "View and Configure Logs" in the *Administration Console Online Help* describes how to view and configure log files that a WebLogic Server instance generates, and *Using WebLogic Logging Services for Application Logging* describes how you can use WebLogic Server message catalogs, non-catalog logging, and servlet logging to produce log messages from your application or a remote Java client, and describes WebLogic's support for internationalization and localization of log messages.

Logging Samples and Tutorials

In addition to this document, BEA Systems provides a variety of logging code samples and tutorials that show logging configuration and API use.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Server.

Log4j Integration in MedRec

The MedRec domain installed with WebLogic Server is configured to enable Log4j logging. Several action classes and MedRec utility classes use the `weblogic.logging.log4j.Log4jLoggingHelper` class to create a new logger, access a Log4j Appender, and register the Appender with the logger. Classes extending the base classes then use the logger to write informational messages to the WebLogic Server log file.

Logging Examples in the WebLogic Server Distribution

WebLogic Server 9.0 optionally installs API code examples in `WL_HOME\samples\server\examples\src\examples`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server 9.0 Start menu.

New and Changed Logging Features in This Release

For information about the new and changed features in WeLogic Server 10.0, see What's New in WebLogic Server 10.0.

CHAPTER 2

Understanding WebLogic Logging Services

WebLogic logging services provide facilities for writing, viewing, filtering, and listening for log messages. These log messages are generated by WebLogic Server instances, subsystems, and Java EE applications that run on WebLogic Server or in client JVMs.

The following sections describe the WebLogic logging services environment, logging process, and log files:

- "What You Can Do With WebLogic Logging Services" on page 2-1
- "How WebLogic Logging Services Work" on page 2-2
- "Server and Subsystem Logs" on page 2-8
- "Log Message Format" on page 2-11
- "Message Attributes" on page 2-12
- "Message Severity" on page 2-13
- "Viewing WebLogic Server Logs" on page 2-15

What You Can Do With WebLogic Logging Services

WebLogic Server subsystems use logging services to provide information about events such as the deployment of new applications or the failure of one or more subsystems. A server instance uses them to communicate its status and respond to specific events. For example, you can use WebLogic logging services to report error conditions or listen for log messages from a specific subsystem.

Each WebLogic Server instance maintains a server log. Because each WebLogic Server 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. The domain log provides the overall status of the domain. See "Server Log Files and Domain Log Files" on page 2-5.

How WebLogic Logging Services Work

The following sections describe the logging environment and provide an overview of the logging process.

Components and Environment

There are two basic components in any logging system: a component that produces log messages and another component to distribute (publish) messages. WebLogic Server subsystems use a message catalog feature to produce messages and the Java Logging APIs to distribute them, by default. Developers can also use message catalogs for applications they develop.

The 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.

See "Using Message Catalogs with WebLogic Server" in *Using WebLogic Logging Services for Application Logging*.

In addition to using the message catalog framework, your application can use the `weblogic.logging.NonCatalogLogger` APIs to send messages to the WebLogic server log, for logging messages that do not need to be internationalized or are internationalized outside the WebLogic I18n framework. With `NonCatalogLogger`, instead of calling messages from a catalog, you place the message text directly in your application code.

See "Using the NonCatalogLogger APIs" in *Using WebLogic Logging Services for Application Logging*.

To distribute messages, WebLogic Server supports Java based logging by default. The `LoggingHelper` class provides access to the `java.util.logging.Logger` object used for server logging. This lets developers take advantage of the Java Logging APIs to add custom handlers, filters, and formatters. See the Sun API documentation at http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/package-summary.html.

Alternatively, you can configure WebLogic Server to use the Jakarta Project Log4j APIs to distribute log messages. See "Log4j and the Commons Logging API" on page 3-5.

Terminology

Logger—A `Logger` object logs messages for a specific subsystem or application component. WebLogic logging services use a single instance of `java.util.logging.Logger` for logging messages from the Message Catalogs, `NonCatalogLogger`, and the Debugging system.

Handler—A class that extends `java.util.logging.Handler` and receives log requests sent to a logger. Each `Logger` instance can be associated with a number of handlers to which it dispatches log messages. A handler attaches to a specific type of a log message; for example, the File Handler for the server log file.

Appender—An appender is Log4j terminology for a handler, in this case, an instance of a class that implements `org.apache.log4j.Appender` and is registered with an `org.apache.log4j.Logger` to receive log events.

Overview of the Logging Process

WebLogic Server subsystems or application code send log requests to `Logger` objects. These `Logger` objects allocate `LogRecord` objects which are passed to `Handler` objects for publication. Both loggers and handlers use severity levels and (optionally) filters to determine if they are interested in a particular `LogRecord` object. When it is necessary to publish a `LogRecord` object externally, a handler can (optionally) use a formatter to localize and format the log message before publishing it to an I/O stream.

Figure 2-1 shows the WebLogic Server logging process: WebLogic Catalog APIs or Commons Logging APIs are used for producing messages; Java Logging (default) and Log4j are options for distributing messages.

Figure 2-1 WebLogic Server Logging Process

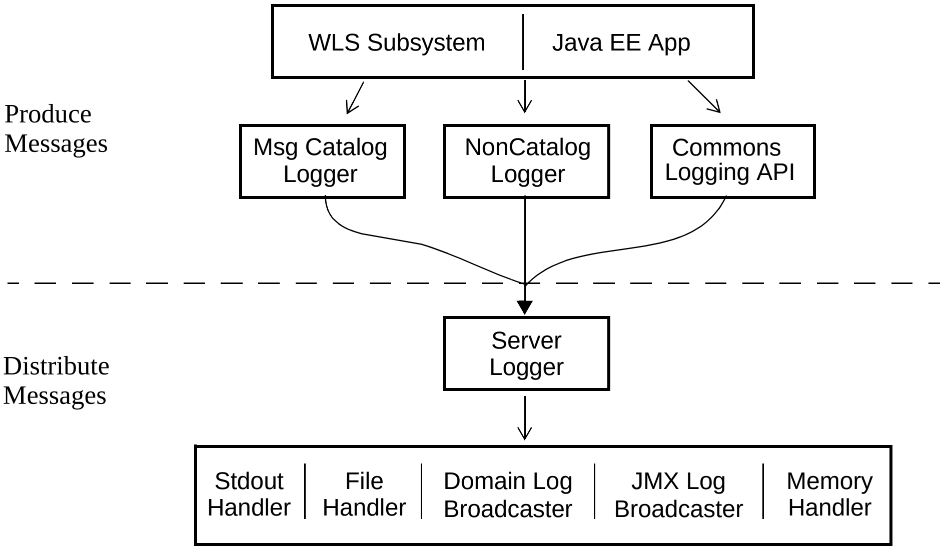


Figure 2-1 illustrates the following process:

1. The client, in this case, a WebLogic Server subsystem or Java EE application, invokes a method on one of the generated Catalog Loggers or the Commons Logging implementation for WebLogic Server.

    a. When WebLogic Server message catalogs and the `NonCatalogLogger` generate messages, they distribute their messages to the server `Logger` object.

    b. The Jakarta Commons Logging APIs define a factory API to get a `Logger` reference which dispatches log requests to the server `Logger` object.

    The server `Logger` object can be an instance of `java.util.logging.Logger` or `org.apache.log4j.Logger`.

2. The server `Logger` object publishes the messages to any message handler that has subscribed to the `Logger`.

    For example, the Stdout Handler prints a formatted message to standard out and the File Handler writes formatted output to the server log file. The Domain Log Broadcaster sends log messages to the domain log, which resides on the Administration Server, and the JMX Log Broadcaster sends log messages to JMX listeners on remote clients.

Best Practices: Integrating Java Logging or Log4j with WebLogic Logging Services

Consider the following recommendations for using WebLogic logging services with Java Logging or Log4j:

- Use the Catalog and NonCatalog loggers or the Commons API for producing log messages for publishing by WebLogic logging services.
- Use a Java Logging or Log4j `Logger` reference for adding custom handlers, appenders, or filters to WebLogic logging services for publishing messages.
- If your application is configured for Java Logging or Log4j, in order to publish application events using WebLogic logging services, create a custom handler or appender that relays the application events to WebLogic logging services using the message catalogs or Commons API.

For more information, see "How to Use Log4j with WebLogic Logging Services" on page 3-6.

Server Log Files and Domain Log Files

Each WebLogic Server instance writes all messages from its subsystems and applications to a server log file that is located on the local host computer. By default, the server log file is located in the `logs` directory below the server instance root directory; for example, *`DOMAIN_NAME`*`\servers\`*`SERVER_NAME`*`\logs\`*`SERVER_NAME.`*`log`, where *`DOMAIN_NAME`* is the name of the directory in which you located the domain and *`SERVER_NAME`* is the name of the server.

In addition to writing messages to the server log file, each server instance forwards a subset of its messages to a domain-wide log file. By default, servers forward only messages of severity level `NOTICE` or higher. While you can modify the set of messages that are forwarded, servers can never forward messages of the `DEBUG` severity level. See "Specifying the Messages That a Server Forwards to the Domain Log" in the *Administration Console Online Help*.

The domain log file provides a central location from which to view the overall status of the domain. The domain log resides in the Administration Server `logs` directory. The default name and location for the domain log file is *`DOMAIN_NAME`*`\servers\`*`ADMIN_SERVER_NAME`*`\logs\`*`DOMAIN_NAME.`*`log`, where *`DOMAIN_NAME`* is the name of the directory in which you located the domain and

`ADMIN_SERVER_NAME` is the name of the Administration Server. See "Changing the Name and Location of the Domain Log File" in the *Administration Console Online Help*.

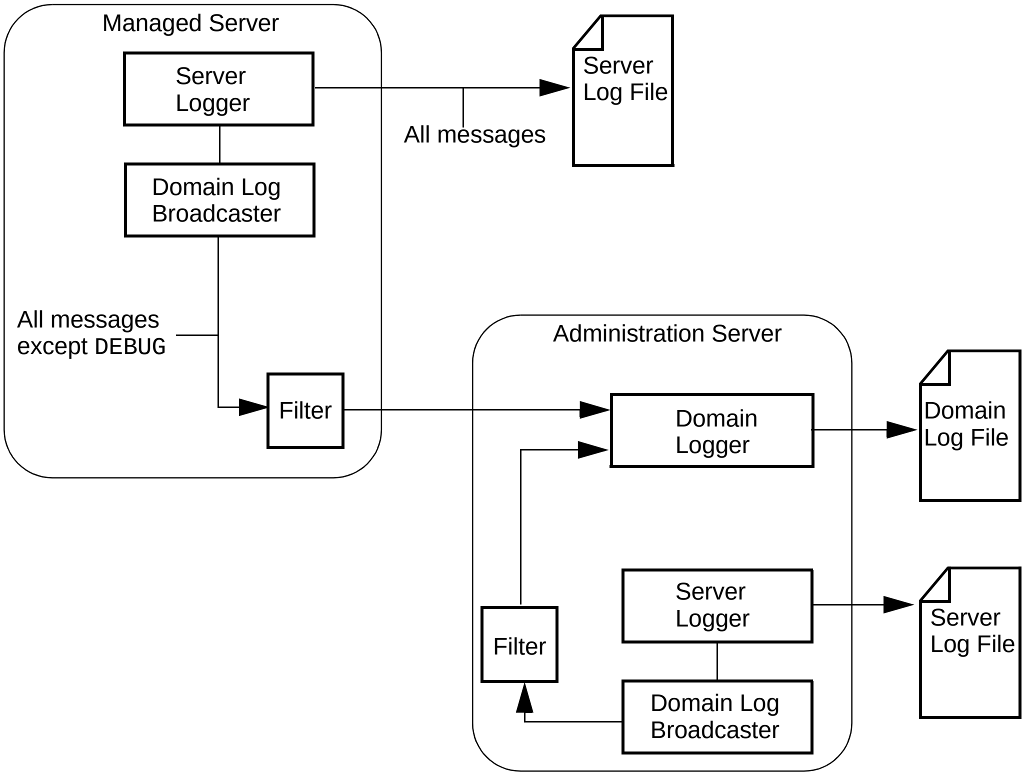
The timestamp for a record in the domain log is the timestamp of the server where the message originated. Log records in the domain log are not written in the order of their timestamps; the messages are written as soon as they arrive. It may happen that a Managed Server remains out of contact with the Administration Server for some period of time. In that case, the messages are buffered locally and sent to the Administration Server once the servers are reconnected.

How a Server Instance Forwards Messages to the Domain Log

To forward messages to the domain log, each server instance broadcasts its log messages. A server broadcasts all messages and message text except for messages of the `DEBUG` severity level.

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 listener with each Managed Server. By default, the listener includes a filter that allows only messages of severity level `NOTICE` and higher to be forwarded to the Administration Server. (See Figure 2-2.)

Figure 2-2 WebLogic Server and Domain Logs



For any given WebLogic Server instance, you can override the default filter and create a log filter that causes a different set of messages to be written to the domain log file. For information on setting up a log filter for a WebLogic Server instance, see "Create Log Filters" in the *Administration Console Online Help*.

If the Administration Server is unavailable, Managed Servers continue to write messages to their local server log files. However, by default, when the servers are reconnected, not all the messages written during the disconnected period are forwarded to the domain log file. A Managed Server keeps a specified number of messages in a buffer so they can be forwarded to the Administration Server when the servers are reconnected. The number of messages kept in the buffer is configured by the `DomainLogBroadcasterBufferSize` attribute on the LogMBean. The default is `1`. With that default value, only the last logged message is forwarded to the Administration Server once it is reconnected. For example, if the Administration Server is unavailable for two hours and then is restored, the domain log will not contain any messages that were generated during the two hours. See "MSI Mode and the Domain Log File" in *Managing Server Startup and Shutdown*.

If you have configured a value greater than `1`, that number of messages will be forwarded to the domain log when the Managed Server is reconnected to the Administration Server.

Note: This can result in a domain log file that lists messages with earlier timestamps after messages with later timestamps. When messages from the buffer of a previously disconnected Managed Server are flushed to the Administration Server, those messages are simply appended to the domain log, even though they were generated before the previous messages in the domain log.

Server and Subsystem Logs

Each subsystem within WebLogic Server generates log messages to communicate its status. For example, when you start a WebLogic Server instance, the Security subsystem writes a message to report its initialization status. To keep a record of the messages that its subsystems generate, WebLogic Server writes the messages to log files.

Server Log

The server log records information about events such as the startup and shutdown of servers, the deployment of new applications, or the failure of one or more subsystems. The messages include information about the time and date of the event as well as the ID of the user who initiated the event.

You can view and sort these server log messages to detect problems, track down the source of a fault, and track system performance. You can also create client applications that listen for these messages and respond automatically. For example, you can create an application that listens for messages indicating a failed subsystem and sends E-mail to a system administrator.

The server log file is located on the computer that hosts the server instance. Each server instance has its own server log file. By default, the server log file is located in the `logs` directory below the server instance root directory; for example, `DOMAIN_NAME\servers\SERVER_NAME\logs\SERVER_NAME.log`, where `DOMAIN_NAME` is the name of the directory in which you located the domain and `SERVER_NAME` is the name of the server. See "Changing the Name and Location of the Server Log File" in the *Administration Console Online Help*.

To view messages in the server log file, you can log on the WebLogic Server host computer and use a standard text editor, or you can log on to any computer and use the log file viewer in the Administration Console. See "Viewing Server Logs" in the *Administration Console Online Help*.

Note: BEA Systems recommends that you do not modify log files by editing them manually. Modifying a file changes the timestamp and can confuse log file rotation. In addition, editing a file might lock it and prevent updates from WebLogic Server and interfere with the Accessor functionality.

For information about the Diagnostic Accessor Service, see "Accessing Diagnostic Data Using the Data Accessor" in *Configuring and Using the Weblogic Diagnostic Framework*.

In addition to writing messages to a log file, each server instance prints a subset of its messages to standard out. Usually, standard out is the shell (command prompt) in which you are running the server instance. However, some operating systems enable you to redirect standard out to some other location. By default, a server instance prints only messages of a `NOTICE` severity level or higher to standard out. (A subsequent section, "Message Severity," describes severity levels.) You can modify the severity threshold so that the server prints more or fewer messages to standard out.

If you use the Node Manager to start a Managed Server, the messages that would otherwise be output to `stdout` or `stderr` when starting a Managed Server, are instead displayed in the Administration Console and written to a single log file for that server instance, *`SERVER_NAME`*`.out`. The server instance's output log is located in the same `logs` directory, below the server instance root directory, along with the WebLogic Server *`SERVER_NAME`*`.log` file; for example, *`DOMAIN_NAME`*`\servers\`*`SERVER_NAME`*`\logs\`*`SERVER_NAME`*`.out`, where *`DOMAIN_NAME`* is the name of the directory in which you located the domain and *`SERVER_NAME`* is the name of the server.

The Node Manager writes its own startup and status messages to a single log file, *`NM_HOME`*`/nodemanager.log`, where *`NM_HOME`* designates the Node Manager installation directory, by default, *`WL_HOME`*`/common/nodemanager`.

For more information on Node Manager log files, see "Node Manager Log and Configuration Files" in *Managing Server Startup and Shutdown*.

Subsystem Logs

The server log messages and log file communicate events and conditions that affect the operation of the server or the application. Some subsystems maintain additional log files to provide an audit of the subsystem's interactions under normal operating conditions. The following list describes each of the additional log files:

- The HTTP subsystem keeps a log of all HTTP transactions in a text file. The default location and rotation policy for HTTP access logs is the same as the server log. You can set the attributes that define the behavior of HTTP access logs for each server or for each virtual host that you define. See "Setting Up HTTP Access Logs" in *Designing and Configuring WebLogic Server Environments* and "Enabling and Configuring HTTP Access Logs" in the *Administration Console Online Help.*
- Each server has a transaction log which stores information about committed transactions coordinated by the server that may not have been completed. WebLogic Server uses the transaction log when recovering from system crashes or network failures. You cannot directly view the transaction log—the file is in a binary format.

  The Transaction Manager uses the default persistent store to store transaction log files. Using the Administration Console, you can change where the default store is located. See "Configure the default persistent store for Transaction Recovery Service migration" in the *Administration Console Online Help*.
- The WebLogic Auditing provider records information from a number of security requests, which are determined internally by the WebLogic Security Framework. The WebLogic Auditing provider also records the event data associated with these security requests, and the outcome of the requests. Configuring an Auditing provider is optional. The default security realm (myrealm) does not have an Auditing provider configured. See "Configuring a WebLogic Auditing Provider" in *Securing WebLogic Server.*

  All auditing information recorded by the WebLogic Auditing provider is saved in `WL_HOME\`*`DOMAIN_NAME`*`\servers\`*`SERVER_NAME`*`\logs\DefaultAuditRecorder.log`. Although an Auditing provider is configured per security realm, each server writes auditing data to its own log file in the server directory.
- The JDBC subsystem records various events related to JDBC connections, including registering JDBC drivers and SQL exceptions. The events related to JDBC are now written to the server log, such as when connections are created or refreshed or when configuration changes are made to JDBC objects. See "Monitoring WebLogic JDBC Resources" in *Configuring and Managing WebLogic JDBC*.
- JMS logging is enabled by default when you create a JMS server, however, you must specifically enable it on message destinations in the JMS modules targeted to this JMS server (or on the JMS template used by destinations).

  JMS server log files contain information on basic message life cycle events, such as message production, consumption, and removal. When a JMS destination hosting the subject message is configured with message logging enabled, then each of the basic message life cycle events will generate a message log event in the JMS message log file.

The message log is located in the `logs` directory, below the server instance root directory, `DOMAIN_NAME\servers\SERVER_NAME\logs\jmsServers\SERVER_NAMEJMSServer\jms.messages.log`, where `DOMAIN_NAME` is the name of the directory in which you located the domain and `SERVER_NAME` is the name of the server.

After you create a JMS server, you can change the default name of its log file, as well as configure criteria for moving (rotating) old log messages to a separate file. See "Configure Message Logging" in the *Administration Console Online Help* and "Monitoring JMS Statistics and Managing Messages" in *Configuring and Managing WebLogic JMS*.

Log Message Format

When a WebLogic Server instance writes a message to the server log file, the first line of each message begins with #### followed by the message attributes. Each attribute is contained between angle brackets.

The following is an example of a message in the server log file:

```
####<Sept 22, 2004 10:46:51 AM EST> <Notice> <WebLogicServer> <MyComputer>
<examplesServer> <main> <<WLS Kernel>> <> <null> <1080575211904> <BEA-000360>
<Server started in RUNNING mode>
```

In this example, the message attributes are: Locale-formatted Timestamp, Severity, Subsystem, Machine Name, Server Name, Thread ID, User ID, Transaction ID, Diagnostic Context ID, Raw Time Value, Message ID, and Message Text. (A subsequent section, "Message Attributes," describes each attribute.)

If a message is not logged within the context of a transaction, the angle brackets for Transaction ID are present even though no Transaction ID is present.

If the message includes a stack trace, the stack trace is included in the message text.

WebLogic Server uses the host computer's default character encoding for the messages it writes.

Format of Output to Standard Out and Standard Error

When a WebLogic Server instance writes a message to standard out, the output does not include the #### prefix and does not include the Server Name, Machine Name, Thread ID, User ID, Transaction ID, Diagnostic Context ID, and Raw Time Value fields.

The following is an example of how the message from the previous section would be printed to standard out:

```
<Sept 22, 2004 10:51:10 AM EST> <Notice> <WebLogicServer> <BEA-000360> <Server
started in RUNNING mode>
```

In this example, the message attributes are: Locale-formatted Timestamp, Severity, Subsystem, Message ID, and Message Text.

Message Attributes

The messages for all WebLogic Server instances contain a consistent set of attributes as described in Table 2-1. In addition, if your application uses WebLogic logging services to generate messages, its messages will contain these attributes.

Table 2-1 Server Log Message Attributes

Attribute	Description
Locale-formatted Timestamp	Time and date when the message originated, in a format that is specific to the locale. The Java Virtual Machine (JVM) that runs each WebLogic Server instance refers to the host computer operating system for information about the local time zone and format.
Severity	Indicates the degree of impact or seriousness of the event reported by the message. See “Message Severity” on page 2-13.
Subsystem	Indicates the subsystem of WebLogic Server that was the source of the message; for example, Enterprise Java Bean (EJB) container or Java Messaging Service (JMS).
Machine Name Server Name Thread ID	Identifies the origins of the message: • `Server Name` is the name of the WebLogic Server instance on which the message was generated. • `Machine Name` is the DNS name of the computer that hosts the server instance. • `Thread ID` is the ID that the JVM assigns to the thread in which the message originated. Log messages that are generated within a client JVM do not include these attributes. For example, if your application runs in a client JVM and it uses the WebLogic logging services, the messages that it generates and sends to the WebLogic client log files will not include these attributes.

Table 2-1 Server Log Message Attributes (Continued)

Attribute	Description
User ID	The user ID under which the associated event was executed. To execute some pieces of internal code, WebLogic Server authenticates the ID of the user who initiates the execution and then runs the code under a special Kernel Identity user ID. Java EE modules such as EJBs that are deployed onto a server instance report the user ID that the module passes to the server. Log messages that are generated within a client JVM do not include this field.
Transaction ID	Present only for messages logged within the context of a transaction.
Diagnostic Context ID	Context information to correlate messages coming from a specific request or application.
Raw Time Value	The timestamp in milliseconds.
Message ID	A unique six-digit identifier. All message IDs that WebLogic Server system messages generate start with `BEA-` and fall within a numerical range of 0-499999. Your applications can use a Java class called `NonCatalogLogger` to generate log messages instead of using an internationalized message catalog. The message ID for `NonCatalogLogger` messages is always `000000`. See "Writing Messages to the WebLogic Server Log" in *Using WebLogic Logging Services for Application Logging*.
Message Text	A description of the event or condition.

Message Severity

The severity attribute of a WebLogic Server log message indicates the potential impact of the event or condition that the message reports.

Table 2-2 lists the severity levels of log messages from WebLogic Server subsystems, starting from the lowest level of impact to the highest.

Table 2-2 Message Severity

Severity	Meaning
`TRACE`	Used for messages from the Diagnostic Action Library. Upon enabling diagnostic instrumentation of server and application classes, `TRACE` messages follow the request path of a method. See "Diagnostic Action Library" in *Configuring and Using the Weblogic Diagnostic Framework*.
`INFO`	Used for reporting normal operations; a low-level informational message.
`NOTICE`	An informational message with a higher level of importance.
`WARNING`	A suspicious operation or configuration has occurred but it might not affect normal operation.
`ERROR`	A user error has occurred. The system or application can handle the error with no interruption and limited degradation of service.
`CRITICAL`	A system or service error has occurred. The system can recover but there might be a momentary loss or permanent degradation of service.
`ALERT`	A particular service is in an unusable state while other parts of the system continue to function. Automatic recovery is not possible; the immediate attention of the administrator is needed to resolve the problem.
`EMERGENCY`	The server is in an unusable state. This severity indicates a severe system failure or panic.
`DEBUG`	A debug message was generated.

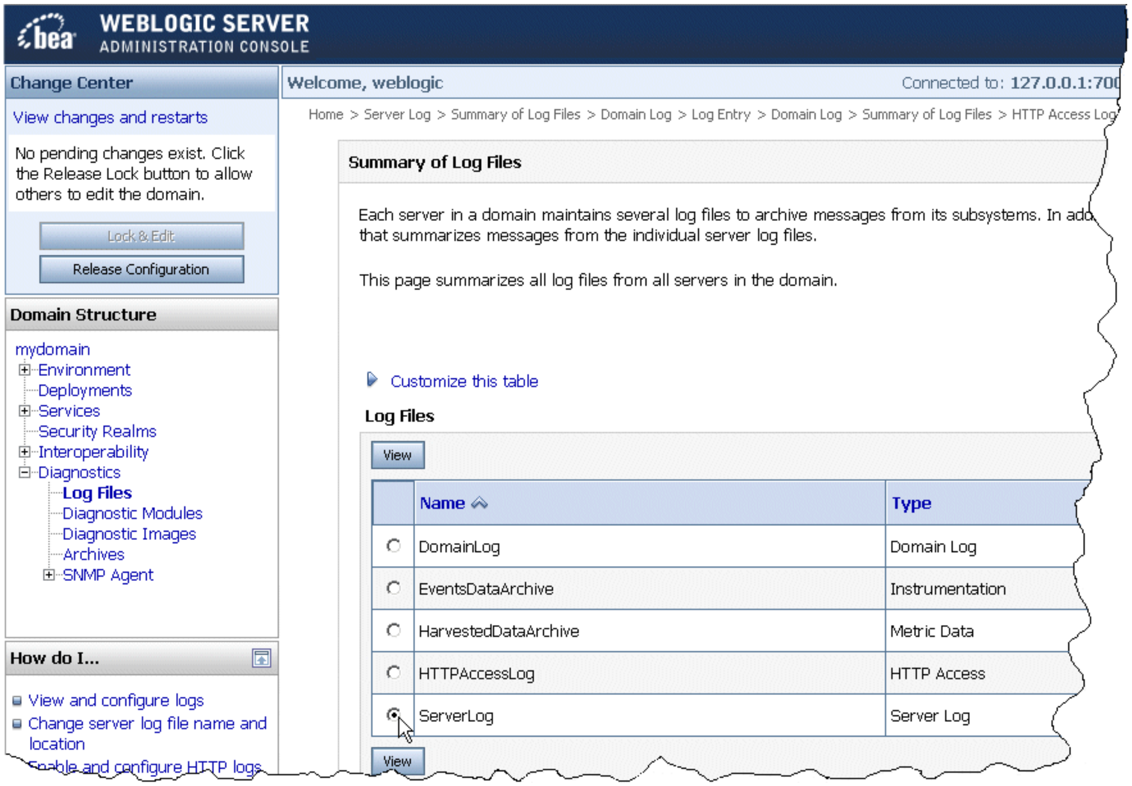
WebLogic Server subsystems generate many messages of lower severity and fewer messages of higher severity. For example, under normal circumstances, they generate many `INFO` messages and no `EMERGENCY` messages.

If your application uses WebLogic logging services, it can use an additional severity level, `DEBUG`. See "Writing Debug Messages" in *Using WebLogic Logging Services for Application Logging*.

Viewing WebLogic Server Logs

The WebLogic Server Administration Console provides a log viewer for all the log files in a domain. The log viewer can find and display the messages based on any of the following message attributes: date, susbsystem, severity, machine, server, thread, user ID, transaction ID, context ID, timestamp, message ID, or message. It can also display messages as they are logged or search for past log messages. (See Figure 2-3.)

Figure 2-3 Log Viewer



No pending changes exist. Click the Release Lock button to allow others to edit the domain.

Summary of Log Files

Each server in a domain maintains several log files to archive messages from its subsystems. In add
that summarizes messages from the individual server log files.

This page summarizes all log files from all servers in the domain.

Log Files

	Name	Type
○	DomainLog	Domain Log
○	EventsDataArchive	Instrumentation
○	HarvestedDataArchive	Metric Data
○	HTTPAccessLog	HTTP Access
●	ServerLog	Server Log

For information about viewing, configuring, and searching message logs, see the following topics in the *Administration Console Online Help*:

- "View and Configure Logs"
- "Viewing Server Logs"

- "Viewing the Domain Log"

For a detailed description of log messages in WebLogic Server message catalogs, see "WebLogic Server Message Catalogs." This index of messages describes all of the messages emitted by WebLogic subsystems and provides a detailed description of the error, a possible cause, and a recommended action to avoid or fix the error.

CHAPTER 3

Configuring WebLogic Logging Services

The following sections describe WebLogic Server logging scenarios and basic configuration tasks. For detailed instructions on filtering and subscribing to messages, see "Filtering WebLogic Server Log Messages" on page 4-1 and "Subscribing to Messages" on page 5-1.

- "Configuration Scenarios" on page 3-1
- "Overview of Logging Services Configuration" on page 3-2
- "Logging Configuration Tasks: Main Steps" on page 3-4
- "Log4j and the Commons Logging API" on page 3-5
- "How to Use Log4j with WebLogic Logging Services" on page 3-6
- "How to Use the Commons API with WebLogic Logging Services" on page 3-9
- "Rotating Log Files" on page 3-11
- "Redirecting JVM Output" on page 3-13

Configuration Scenarios

WebLogic Server system administrators and developers configure logging output and filter log messages to troubleshoot errors or to receive notification for specific events.

The following tasks describe some logging configuration scenarios:

- Stop `DEBUG` and `INFO` messages from going to the log file.

- Allow `INFO` level messages from the HTTP subsystem to be published to the log file, but not to standard out.
- Specify that a handler publishes messages that are `WARNING` severity level or higher.
- Track log information for individual servers in a cluster.

Overview of Logging Services Configuration

Volume control of logging is provided through the `LogMBean` interface. In the logging process, a logging request is dispatched to subscribed handlers or appenders. WebLogic Server provides handlers for sending log messages to standard out, the server log file, broadcasting messages to the domain log, remote clients, and a memory buffer for tail viewing log events in the WebLogic Server Administration Console. You can achieve volume control for each type of handler by filtering log messages based on severity level and other criteria. The `LogMBean` defines attributes for setting the severity level and specifying filter criteria for WebLogic Server handlers.

In earlier versions of WebLogic Server, system administrators and developers had only programmatic access to loggers and handlers. In this release of WebLogic Server, you can configure handlers using MBeans, eliminating the need to write code for most basic logging configurations. The Administration Console and WebLogic Server Scripting Tool (WLST) provide an interface for interacting with logging MBeans. Additionally, you can specify `LogMBean` parameters on the command line using `Dweblogic.log.`*`attribute-name`*`=`*`value`*; for example, `Dweblogic.log.StdoutSeverity=Debug`. See "weblogic.Server Configuration Options" in *weblogic.Server Command-Line Reference*.

For advanced usage scenarios and for configuring loggers, you use the Java Logging APIs.

Setting the severity level on a handler is the simplest type of volume control; for example, any message of a lower severity than the specified threshold severity level, will be rejected. For example, by default, the Stdout Handler has a `NOTICE` threshold severity level. Therefore, `INFO` and `DEBUG` level messages are not sent to standard out.

Configuring a filter on a handler lets you specify criteria for accepting log messages for publishing; for example, only messages from the HTTP and JDBC subsystems are sent to standard out.

Note: `java.util.logging.LoggingPermission` is required for a user to change the configuration of a logger or handler. In production environments, BEA Systems recommends using the Java Security Manager with `java.util.logging.LoggingPermission` enabled for the current user.

See "Using the Java Security Manager to Protect WebLogic Resources" in *Programming WebLogic Security* and the Sun API documentation at http://java.sun.com/j2se/1.5.0/docs/guide/logging/overview.html.

The following sections describe in more detail the control points for configuring WebLogic Server logging behavior.

Using Log Severity Levels

Each log message has an associated severity level. The level gives a rough guide to the importance and urgency of a log message. WebLogic Server has predefined severities, ranging from `TRACE` to `EMERGENCY`, which are converted to a log level when dispatching a log request to the logger. A log level object can specify any of the following values, from lowest to highest impact:

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

You can set a log severity level on the logger and the handler. When set on the logger, none of the handlers receive an event which is rejected by the logger. For example, if you set the log level to `NOTICE` on the logger, none of the handlers will receive `INFO` level events. When you set a log level on the handler, the restriction only applies to that handler and not the others. For example, turning `DEBUG` off for the File Handler means no `DEBUG` messages will be written to the log file, however, `DEBUG` messages will be written to standard out.

See the `weblogic.logging.Severities` class for a description of the supported severity levels.

You set log levels for handlers using the Administration Console, WLST, or the command line. Loggers are configured only through the API. See "Setting the Severity Level for Loggers and Handlers" on page 4-4.

Using Log Filters

To provide more control over the messages that a `Logger` object publishes, you can create and set a filter. A filter is a class that uses custom logic to evaluate the log record content which you use to accept or reject a log message; for example, to filter out messages of a certain severity level, from a particular subsystem, or according to specified criteria. The `Logger` object publishes only the log messages that satisfy the filter criteria. You can create separate filters for the messages that each server instance writes to its server log file, standard out, memory buffer, or broadcasts to the domain-wide message log.

You can associate a filter with loggers and handlers. You configure filters for handlers using the Administration Console, WLST, or the command line. There are `LogFilterMBean` attributes to define filters for Stdout, Log File, Log Broadcaster, and Memory Handlers, or you can implement custom filtering logic programmatically. The `LogFilterMBean` interface defines the filtering criteria based on user ID and subsystem. Filters for loggers are configured only through the API.

See "Setting a Filter for Loggers and Handlers" on page 4-7.

Logging Configuration Tasks: Main Steps

The following steps summarize how you configure and filter log messages that WebLogic Server generates. Related documentation and later sections in this guide describe these steps in more detail.

1. Use the Administration Console to manage log files and configure the following logging options:
   a. Domain and server log file name and location, rotation pattern, location of archived log files, and number of log files stored. See "View and Configure Logs" in the *Administration Console Online Help*.
   b. Types of messages that the server sends to standard out. See "Specify Messages for Standard Out" in the *Administration Console Online Help*.
   c. Which messages a server instance sends to the domain log. See "Specifying the Messages That a Server Forwards to the Domain Log" in the *Administration Console Online Help*.
   d. Log files for HTTP requests. See "Enabling and Configuring HTTP Access Logs" in the *Administration Console Online Help*.
   e. Specify the logging implementation (Java Logging or Log4j). See "How to Use Log4j with WebLogic Logging Services" on page 3-6.
   f. Specify message destination and configure filtering log messages by severity level or other criteria. See "Filter Log Messages" in the *Administration Console Online Help*.
2. Alternatively, configure filtering log messages on the message handler using the WebLogic Scripting Tool. See "Configuring Existing Domains" in *WebLogic Scripting Tool*.
3. Filter log messages published by the logger using the Java APIs. See "Filtering Messages by Severity Level or Other Criteria" on page 4-3.

Log4j and the Commons Logging API

Application developers who want to use the WebLogic Server message catalogs and logging services as a way for their applications to produce log messages must know XML and the Java APIs. Many developers and system administrators use Log4j, which is a predecessor to the Java Logging APIs. Log4j is an open source tool developed for putting log statements in your application. The Log4j Java logging facility was developed by the Jakarta Project of the Apache Foundation. You can learn more about Log4j at The Log4j Project at `http://logging.apache.org/log4j/docs/`.

WebLogic Server supports Log4j as a configuration option for WebLogic logging services. See "How to Use Log4j with WebLogic Logging Services" on page 3-6.

The Jakarta Commons Logging APIs provide an abstraction layer that insulates users from the underlying logging implementation, which can be Log4j or Java Logging APIs. WebLogic Server provides an implementation of the Commons `LogFactory` interface, letting you issue requests to the server `Logger` using this API. See "How to Use the Commons API with WebLogic Logging Services" on page 3-9.

About Log4j

Log4j has three main components: loggers, appenders, and layouts. The following sections provide a brief introduction to Log4j.

Loggers

Log4j defines a `Logger` class. An application can create multiple loggers, each with a unique name. In a typical usage of Log4j, an application creates a `Logger` instance for each application class that will emit log messages. Loggers exist in a namespace hierarchy and inherit behavior from their ancestors in the hierarchy.

Appenders

Log4j defines appenders (handlers) to represent destinations for logging output. Multiple appenders can be defined. For example, an application might define an appender that sends log messages to standard out, and another appender that writes log messages to a file. Individual loggers might be configured to write to zero or more appenders. One example usage would be to send all logging messages (all levels) to a log file, but only `ERROR` level messages to standard out.

Layouts

Log4j defines layouts to control the format of log messages. Each layout specifies a particular message format. A specific layout is associated with each appender. This lets you specify a different log message format for standard out than for file output, for example.

How to Use Log4j with WebLogic Logging Services

WebLogic logging services use an implementation based on the Java Logging APIs, by default. Using the `LogMBean.isLog4jLoggingEnabled` attribute, you can direct the logging services to use Log4j instead.

In the Administration Console, you can specify Log4j or keep the default Java Logging implementation. (See "Specifying the Logging Implementation" in the *Administration Console Online Help*.)

Alternatively, you can configure Log4j logging through the `LogMBean` interface and by adding WebLogic-specific Log4j classes, `WL_HOME/server/lib/wllog4j.jar`, and the `log4j.jar` file to the server `CLASSPATH`. The recommended way to do this is to place the `wllog4j.jar` and `log4j.jar` files in the `DOMAIN_NAME/lib` directory; there, they will get added to the server `CLASSPATH` dynamically during server startup.

Note: WebLogic Server does not provide a Log4j version in its distribution.

When Log4j is enabled, you get a reference to the `org.apache.log4j.Logger` that the server is using from the `weblogic.logging.log4j.Log4jLoggingHelper` class.

With a Log4j `Logger` reference, you can attach you own custom appender to receive the server log events; for example, you might attach an appender that sends the server log events to Syslog or the Windows Event Viewer. Additionally, you can use the `Logger` reference to issue log requests to WebLogic logging services; this requires that the Log4j libraries be available to your deployed application.

If your application has no requirement to interact with WebLogic logging services, package the Log4j libraries in the application's `LIB` directory. The server logging will continue to use the default Java Logging implementation.

See Listing 3-2, a Log4j code example that demonstrates using the Log4j `Logger`.

Enabling Log4j Logging

To specify logging to a Log4j `Logger` instead of the default Java Logging:

- When you start the Administration Server, include the following Java option in the `weblogic.Server` command:

  `-Dweblogic.log.Log4jLoggingEnabled=true`

  See "weblogic.Server Configuration Options" in *weblogic.Server Command-Line Reference*.

- After the Administration Server has started, use the Administration Console to specify the Log4j logging implementation. See "Specifying the Logging Implementation" in the *Administration Console Online Help*.

- Use the WLST to set the value of the `Log4jLoggingEnabled` property and re-start the server.

  The WLST commands in Listing 3-1 enable logging to a Log4j `Logger` in the Administration Server:

Listing 3-1 Enabling Log4j Logging

```
#invoke WLST
C:\>java weblogic.WLST

#connect WLST to an Administration Server
wls:/offline> connect('username','password')

#navigate to the writable MBean configuration tree
wls:/mydomain/serverConfig> edit()
wls:/mydomain/edit> startEdit()

#set cmo to the server log config
wls:/mydomain/edit !> cd("Servers/myserver/Log/myserver")

#set log4j logging to true
wls:/mydomain/edit/Servers/myserver/Log/myserver !>
cmo.setLog4jLoggingEnabled(true)

#save and activate the changes
wls:/mydomain/edit/Servers/myserver/Log/myserver !> save()
wls:/mydomain/edit/Servers/myserver/Log/myserver !> activate()
```

For more information about using WLST, see "Using the WebLogic Scripting Tool" in *WebLogic Scripting Tool*. For more information about `isLog4jLoggingEnabled`, see LogMBean in the *WebLogic Server MBean Reference*.

You can enable Log4j for the server `Logger` as well as the domain `Logger`, which resides only on the Administration Server. The domain Log4j `Logger` reference is provided by invoking the `weblogic.logging.log4j.Log4jLoggingHelper.getLog4jDomainLogger()` method.

See Listing 3-2 and Listing 3-3, Log4j logging configuration examples that show:

- The server `Logger` using Log4j and the domain `Logger` using the default Java `Logger`.
- How to specify a severity level for Stdout and a filter for messages going to the server log file in the `config.xml` file.

Listing 3-2 Log4j Code Example

```
import org.apache.log4j.Logger;
import weblogic.logging.log4j.Log4jLoggingHelper;
import weblogic.logging.LoggerNotAvailableException;

/**
 * This example shows how to use the Log4j server Logger.
 */

public class MyLog4jTest {
  public void testWLSLog4j() {
    try {
      Logger logger = Log4jLoggingHelper.getLog4jServerLogger();
      logger.addAppender(myAppender); // The Appender is configured using either
the log4j props file or other custom mechanism.
      logger.info("Test log message");
    } catch(LoggerNotAvailableException lex) {
    System.err.println("Unable to get a reference to the log4j Logger: "+
      lex.getMessage())
    }
  }
}
```

Listing 3-3 Logging Configuration Example

```
<con:log>
    <con:name>medrec</con:name>
    <con:file-name>medrec.log</con:file-name>
    <con:rotation-type>bySize</con:rotation-type>
    <con:file-min-size>20000</con:file-min-size>
    <con:log4j-logging-enabled>false</con:log4j-logging-enabled>
</con:log>

<con:log>
    <con:name>MedRecServer</con:name>
    <con:rotation-type>bySize</con:rotation-type>
    <con:file-min-size>20000</con:file-min-size>
    <con:stdout-severity>Debug</con:stdout-severity>
    <con:stdout-filter>MyFilter</con:stdout-filter>
    <con:log4j-logging-enabled>true</con:log4j-logging-enabled>
</con:log>

<con:log-filter>
    <con:name>MyFilter</con:name>
    <con:subsystem-name>HTTP</con:subsystem-name>
    <con:subsystem-name>IIOP</con:subsystem-name>
    <con:subsystem-name>JDBC</con:subsystem-name>
    <con:subsystem-name>JMS</con:subsystem-name>
</con:log-filter>
```

You have programmatic access to the Log4j `Logger` and its appenders (handlers) and layouts (formatters) for configuration purposes. See "Setting a Severity Level and Filter on a Log4j Appender" on page 4-10.

Java Logging is the default for client and server-side logging; Log4j is available only for server-side and not client-side logging.

How to Use the Commons API with WebLogic Logging Services

WebLogic logging services provide the Commons `LogFactory` and `Log` interface implementations that direct requests to the underlying logging implementation being used by WebLogic logging services.

To use Commons Logging, put the WebLogic-specific Commons classes, *$BEA_HOME/*`modules/com.bea.core.weblogic.commons-logging_1.0.0.0.jar`, together with the `commons-logging.jar` file in one of the following locations:

- `APP-INF/LIB` or `WEB-INF/LIB` directory
- *`DOMAIN_NAME`*`/LIB` directory
- server `CLASSPATH`

Note: WebLogic Server does not provide a Commons logging version in its distribution.

Listing 3-4 illustrates how to use the Commons interface:

1. Set the system property `org.apache.commons.logging.LogFactory` to `weblogic.logging.commons.LogFactoryImpl`.

   This `LogFactory` creates instances of `weblogic.logging.commons.LogFactoryImpl` that implement the `org.apache.commons.logging.Log` interface.

2. From the `LogFactory`, get a reference to the Commons `Log` object by name.

   This name appears as the subsystem name in the log file.

3. Use the `Log` object to issue log requests to WebLogic logging services.

   The Commons `Log` interface methods accept an object. In most cases, this will be a string containing the message text.

   The Commons `LogObject` takes a message ID, subsystem name, and a string message argument in its constructor. See `org.apache.commons.logging` at `http://jakarta.apache.org/commons/logging/api/index.html`.

4. The `weblogic.logging.commons.LogImpl` log methods direct the message to the server log.

Listing 3-4 Commons Code Example

```
import org.apache.commons.logging.LogFactory;
import org.apache.commons.logging.Log;


public class MyCommonsTest {
  public void testWLSCommonsLogging() {
    System.setProperty(LogFactory.FACTORY_PROPERTY,
      "weblogic.logging.commons.LogFactoryImpl");
    Log clog = LogFactory.getFactory().getInstance("MyCommonsLogger");
    // Log String objects
    clog.debug("Hey this is common debug");
    clog.fatal("Hey this is common fatal", new Exception());
    clog.error("Hey this is common error", new Exception());
    clog.trace("Dont leave your footprints on the sands of time");
  }
}
```

Rotating Log Files

By default, when you start a WebLogic Server instance in **development mode**, the server automatically renames (rotates) its local server log file as *`SERVER_NAME.`*`log`*`.n`*. For the remainder of the server session, log messages accumulate in *`SERVER_NAME.`*`log` until the file grows to a size of 500 kilobytes.

Each time the server log file reaches this size, the server renames the log file and creates a new *`SERVER_NAME.`*`log` to store new messages. By default, the rotated log files are numbered in order of creation *`filenamennnnn`*, where *`filename`* is the name configured for the log file. You can configure a server instance to include a time and date stamp in the file name of rotated log files; for example, *`server-name-%yyyy%-%mm%-%dd%-%hh%-%mm%`*`.log`.

By default, when you start a server instance in **production mode**, the server rotates its server log file whenever the file grows to 5000 kilobytes in size. It does not rotate the local server log file when you start the server. For more information about changing the mode in which a server starts, see "Change to Production Mode" in the *Administration Console Online Help*.

You can change these default settings for log file rotation. For example, you can change the file size at which the server rotates the log file or you can configure a server to rotate log files based on a time interval. You can also specify the maximum number of rotated files that can accumulate. After the number of log files reaches this number, subsequent file rotations delete the oldest log file and create a new log file with the latest suffix.

Note: WebLogic Server sets a threshold size limit of 500 MB before it forces a hard rotation to prevent excessive log file growth.

For information on setting up log file rotation, see "Rotating Log Files" in the *Administration Console Online Help*.

To cause the immediate rotation of the server, domain, or HTTP access log file, use the `LogRuntime.forceLogRotation()` method. See LogRuntimeMBean in the *WebLogic Server MBean Reference*.

The WLST commands in Listing 3-5 cause the immediate rotation of the server log file.

Listing 3-5 Log Rotation on Demand

```
#invoke WLST
C:\>java weblogic.WLST
```

```
#connect WLST to an Administration Server
wls:/offline> connect('username','password')

#navigate to the ServerRuntime MBean hierarchy
wls:/mydomain/serverConfig> serverRuntime()
wls:/mydomain/serverRuntime>ls()

#navigate to the server LogRuntimeMBean
wls:/mydomain/serverRuntime> cd('LogRuntime/myserver')
wls:/mydomain/serverRuntime/LogRuntime/myserver> ls()
-r--   Name                                         myserver
-r--   Type                                         LogRuntime

-r-x   forceLogRotation                             java.lang.Void :

#force the immediate rotation of the server log file
wls:/mydomain/serverRuntime/LogRuntime/myserver> cmo.forceLogRotation()
wls:/mydomain/serverRuntime/LogRuntime/myserver>
```

The server immediately rotates the file and prints the following message:

```
<Mar 2, 2005 3:23:01 PM EST> <Info> <Log Management> <BEA-170017> <The log file
C:\diablodomain\servers\myserver\logs\myserver.log will be rotated. Reopen the
log file if tailing has stopped. This can happen on some platforms like Windows.>

<Mar 2, 2005 3:23:01 PM EST> <Info> <Log Management> <BEA-170018> <The log file
has been rotated to C:\diablodomain\servers\myserver\logs\myserver.log00001.
Log messages will continue to be logged in
C:\diablodomain\servers\myserver\logs\myserver.log.>
```

Specifying the Location of Archived Log Files

By default, the rotated files are stored in the same directory where the log file is stored. You can specify a different directory location for the archived log files by using the Administration Console or setting the `LogFileRotationDir` property of the `LogFileMBean` from the command line. See LogFileMBean in the *WebLogic Server MBean Reference*.

The following command specifies the directory location for the archived log files using the `-Dweblogic.log.LogFileRotationDir` Java startup option:

```
java -Dweblogic.log.LogFileRotationDir=c:\foo
-Dweblogic.management.username=installadministrator
-Dweblogic.management.password=installadministrator weblogic.Server
```

Notification of Rotation

When the log file exceeds the rotation threshold that you specify, the server instance prints a log message that states that the log file will be rotated. Then it rotates the log file and prints an additional message that indicates the name of the file that contains the old messages.

For example, if you set up log files to rotate by size and you specify 500K as the minimum rotation size, when the server determines that the file is greater than 500K in size, the server prints the following message:

```
<Sept 20, 2004 1:51:09 PM EST> <Info> <Log Management> <MachineName>
<MedRecServer> <ExecuteThread: '2' for queue: 'weblogic.kernel.System'> <<WLS
Kernel>> <> <> <1095692939895> <BEA-170017> <The log file
C:\bea\wls\samples\domains\medrec\servers\MedRecServer\logs\medrec.log will be
rotated. Reopen the log file if tailing has stopped. This can happen on some
platforms like Windows.>
```

The server immediately rotates the file and prints the following message:

```
<Sept 20, 2004 1:51:09 PM EST> <Info> <Log Management> <MachineName>
<MedRecServer> <ExecuteThread: '2' for queue: 'weblogic.kernel.System'> <<WLS
Kernel>> <> <> <1095692939895> <BEA-170018> <The log file has been rotated to
C:\bea\wls\samples\domains\medrec\servers\MedRecServer\logs\medrec.log00001.
Log messages will continue to be logged in
C:\bea\wls\samples\domains\medrec\servers\MedRecServer\logs\medrec.log.>
```

Note that the severity level for both messages is `INFO`. The message ID for the message before rotation is always `BEA-170017` and the ID for the message after rotation is always `BEA-170018`.

File systems such as the standard Windows file system place a lock on files that are open for reading. On such file systems, if your application is tailing the log file, or if you are using a command such as the DOS `tail -f` command in a command prompt, the tail operation stops after the server has rotated the log file. The `tail -f` command prints messages to standard out as lines are added to a file. For more information, enter `help tail` in a DOS prompt.

To remedy this situation for an application that tails the log file, you can create a JMX listener that notifies your application when the server emits the log rotation message. When your application receives the message, it can restart its tailing operation. To see an example of a JMX listener, see "Subscribing to Messages" on page 5-1.

Redirecting JVM Output

The JVM in which a WebLogic Server instance runs, sends messages to standard error and standard out. Server as well as application code write directly to these streams instead of using the logging mechanism. Through a configuration option, you can redirect the JVM output to all the registered log destinations, like the server terminal console and log file. When enabled, a log entry appears as a message of `NOTICE` severity. Redirecting the JVM output does not capture output from native code, for example thread dumps from the JVM.

For example, to redirect the JVM standard out messages:

- When you start the Administration Server, include the following Java option in the `weblogic.Server` command:

  `-Dweblogic.log.RedirectStdoutToServerLogEnabled=true`

  See "weblogic.Server Configuration Options" in *weblogic.Server Command-Line Reference*.

- After the Administration Server has started, use the Administration Console to redirect the JVM standard out messages. See "Redirect JVM Output" in the *Administration Console Online Help*.

- Use the WLST to change the value of the `RedirectStdoutToServerLogEnabled` property of the `LogMBean` and re-start the server.

  The WLST commands in Listing 3-6 redirect the JVM standard out messages in the Administration Server to the server logging destinations.

Listing 3-6 Redirecting Stdout to Server Logging Destinations

```
C:\>java weblogic.WLST
wls:/offline> connect('username','password')
wls:/mydomain/serverConfig> edit()
wls:/mydomain/edit> startEdit()
wls:/mydomain/edit !> cd("Servers/myserver/Log/myserver")
wls:/mydomain/edit/Servers/myserver/Log/myserver !>
cmo.setRedirectStdoutToServerLogEnabled(true)
wls:/mydomain/edit/Servers/myserver/Log/myserver !> save()
wls:/mydomain/edit/Servers/myserver/Log/myserver !> activate()
```

For more information about using WLST, see "Editing Configuration MBeans" in *WebLogic Scripting Tool*. For more information about `RedirectStdoutToServerLogEnabled`, see LogMBean in the *WebLogic Server MBean Reference*.

CHAPTER 4

Filtering WebLogic Server Log Messages

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

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

- "The Role of Logger and Handler Objects" on page 4-2
- "Filtering Messages by Severity Level or Other Criteria" on page 4-3
- "Setting the Severity Level for Loggers and Handlers" on page 4-4
- "Setting a Filter for Loggers and Handlers" on page 4-7
- "Setting a Severity Level and Filter on a Log4j Appender" on page 4-10

For related information, see:

- "Create Log Filters" for information on setting up a log filter for a WebLogic Server instance in the *Administration Console Online Help*.
- "Subscribing to Messages" on page 5-1 for information about creating and subscribing a message handler.

The Role of Logger and Handler Objects

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 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.log.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. See "Writing Messages from a Client Application" in *Using WebLogic Logging Services for Application Logging*.

  - `FileStreamHandler`, which writes messages from the client JVM to the client's log file.

- 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 server `Logger` object:

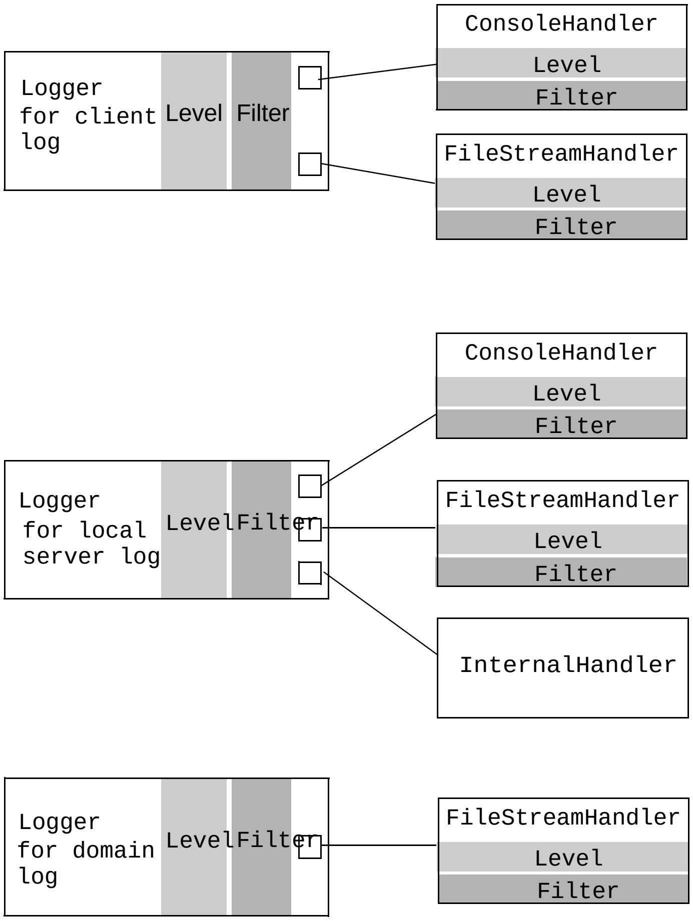
  - `ConsoleHandler`, which makes messages available to the server's standard out.
  - `FileStreamHandler`, which writes messages to the server log file.
  - An internal handler, which broadcasts messages to the domain log and JMX clients, and publishes messages to the Administration Server.

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

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

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

Figure 4-1 WebLogic Logging Services Contexts



Filtering Messages by Severity Level or Other Criteria

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:

`TRACE`, `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `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 level to `WARNING`, the `Logger` object publishes only `WARNING`, `ERROR`, `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 `FileStreamHandler`, 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 `WARNING` severity level or higher.
- Publish all messages of `WARNING` severity level or higher to the domain-wide message log on the Administration Server.

Setting the Severity Level for Loggers and Handlers

The Administration Console and WLST provide a way to set the severity level for a `Handler` object through standard MBean commands. To set the severity level for a `Logger` object, you must use the `Logger` API. For client-side logging, the only way to set a log severity level is through using the Java Logging API.

Setting the Level for Loggers

To set the severity 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. See the Sun API documentation for `Logger`: http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/Logger.html

2. Invokes the `Logger.setLevel(Level 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. WebLogic Server maps the `java.util.logging.Level` to the appropriate `WLLevel`. For a list of valid values, see the `WLLevel` Javadoc.

    For example:

    `setLevel(WLLevel.ALERT)`

Setting the Level for Handlers

To set the severity level for a `Handler` object using the API, 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. See the Sun API documentation for `Logger`: http://java.sun.com/j2se/1.5.0/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(Level 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. WebLogic Server maps the `java.util.logging.Level` to the appropriate `WLLevel`. For a list of valid values, see the `WLLevel` Javadoc.

   For example:

   `setLevel(WLLevel.ALERT)`

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

```
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.ALERT);
            }
        }
    }
}
```

You can configure the severity level for a `Handler` object through the `LogMBean` interface using the Administration Console or the command line:

- See "Filter Log Messages" in the *Administration Console Online Help*, for information on setting a severity level.
- The WLST commands in Listing 4-2 set the severity level for the Stdout Handler to `INFO`.

Listing 4-2 Setting the Severity Level for the Stdout Handler

```
C:\>java weblogic.WLST
wls:/offline> connect('username','password')
wls:/mydomain/serverConfig> edit()
wls:/mydomain/edit> startEdit()
wls:/mydomain/edit !> cd("Servers/myserver/Log/myserver")
wls:/mydomain/edit/Servers/myserver/Log/myserver !>
cmo.setStdoutSeverity("Info")
wls:/mydomain/edit/Servers/myserver/Log/myserver !> save()
wls:/mydomain/edit/Servers/myserver/Log/myserver !> activate()
```

For more information about using WLST, see "Using the WebLogic Scripting Tool" in *WebLogic Scripting Tool*. For more information about `setStdoutSeverity`, see LogMBean in the *WebLogic Server MBean Reference*.

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 as well. When you set a filter on a handler, the filter affects only the behavior of the specific handler.

The Administration Console and WLST provide a way to set a filter on the `Handler` object through standard MBean commands. To set a filter on the `Logger` object, you must use the `Logger` API. For client-side logging, the only way to set a filter is through using the Java Logging API.

To set a filter:

1. Create a class that implements `java.util.logging.Filter`. See Listing 4-3.

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.
   - `getDomainLogger` 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.
4. To set a filter for a `Handler` object using the API, 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.
   - `getDomainLogger` 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 4-3 provides an example class that rejects all messages from the Deployer subsystem.

Listing 4-3 Example Filter for a Java 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;
        }
    }
}
```

You can configure a filter for a `Handler` object through the `LogMBean` interface using the Administration Console or the command line:

- See "Create Log Filters" for information on setting up a log filter for a WebLogic Server instance in the *Administration Console Online Help.*
- The WLST commands in Listing 4-4 create and set a filter on the Domain Log Broadcaster.

Listing 4-4 Setting up a Domain Log Filter

```
C:\>java weblogic.WLST
wls:/offline> connect('username','password')
wls:/mydomain/serverConfig> edit()
wls:/mydomain/edit> startEdit()
wls:/mydomain/edit !> cmo.createLogFilter('myFilter')
wls:/mydomain/edit !> cd("Servers/myserver/Log/myserver")
wls:/mydomain/edit/Servers/myserver/Log/myserver !>
cmo.setDomainLogBroadcastFilter(getMBean('/LogFilters/myFilter'))
wls:/mydomain/edit/Servers/myserver/Log/myserver !> save()
wls:/mydomain/edit/Servers/myserver/Log/myserver !> activate()
```

For more information about using WLST, see "Using the WebLogic Scripting Tool" in *WebLogic Scripting Tool*. For more information about `setDomainLogBroadcastFilter`, see LogMBean in the *WebLogic Server MBean Reference*.

Filtering Domain Log Messages

To filter the messages that each Managed Server publishes to the domain log, you can use the Administration Console (see "Create Log Filters") or WLST (see Listing 4-4) to create a log filter for the domain log.

Any Java Logging severity level or filter that you set on the `Logger` object that manages a server instance's log file **supersedes** a domain log filter. For example, if the level of the server `Logger` object is set to `WARNING`, a domain log filter will receive only messages of the `WARNING` level or higher.

You can define a domain log filter which modifies the set of messages that one or more servers send to the domain log. By default, all messages of severity `NOTICE` or higher are sent.

Note: Messages of severity `DEBUG` are never sent to the domain log, even if you use a filter.

See "Filter Log Messages" in the *Administration Console Online Help* which describes configuring a domain log filter for a WebLogic Server instance using the Administration Console.

Setting a Severity Level and Filter on a Log4j Appender

The Administration Console and WLST provide a way to set the level for an `Appender` object through standard MBean commands. To set the level for a `Logger` object, you must use the `Logger` API.

To set the level for an `Appender` object using the API, create a class that does the following:

1. Invokes the one of the following `Log4jLoggingHelper` methods (See Listing 4-5).
   - `getLog4jServerLogger` 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.
   - `getLog4jDomainLogger` if the current context is the Administration Server and you want to retrieve the `Logger` object that manages the domain log.
2. Invokes the `logger.getAllAppenders()` method.

   ```
   Enumeration e = logger.getAllAppenders();
   ```

   The method returns all the appenders that are registered with the `Logger` object.

3. Iterates through the list of appenders and gets each appender name.
4. Invokes the `app.setThreshold(WLLog4jLevel level)` method.

   To set the level of a Log4j `Appender` object, you must pass a value that is defined in the `weblogic.logging.log4j.WLLog4jLevel` class. WebLogic Server maps the `org.apache.log4j.Level` to the appropriate `WLLevel`. For a list of valid values, see the `WLLevel` Javadoc.

To set a filter, implement a class that extends `org.apache.log4j.filter` and adds the filter to the Appender, invoke the `app.addFilter(Filter newFilter)` method.

Listing 4-5 provides an example class that does the following:

- Publishes messages of the `WARNING` level or higher in the server log.
- Publishes messages of the `INFO` level or higher to standard out.
- Rejects `INFO` messages from the HTTP subsystem.

Listing 4-5 Example: Setting a Log4j Level and Filter

```
package weblogic.logging.examples;

import java.util.Enumeration;
import org.apache.log4j.AppenderSkeleton;
import org.apache.log4j.Logger;
import org.apache.log4j.spi.Filter;
import org.apache.log4j.spi.LoggingEvent;
import weblogic.logging.LoggerNotAvailableException;
import weblogic.logging.NonCatalogLogger;
import weblogic.logging.Severities;
import weblogic.logging.log4j.AppenderNames;
import weblogic.logging.log4j.Log4jLoggingHelper;
import weblogic.logging.log4j.WLLog4jLevel;
import weblogic.logging.log4j.WLLog4jLogEvent;

/**
 * This class sets a level and filter on a Log4j Appender.
 */
public class Log4jFilterExamplesStartup {
  public static void main(String[] args) {
    try {
      System.out.println("Invoked the log4j filter example startup class");
      Logger logger = Log4jLoggingHelper.getLog4jServerLogger();
      Enumeration e = logger.getAllAppenders();
      while (e.hasMoreElements()) {
        AppenderSkeleton app = (AppenderSkeleton) e.nextElement();
        String name = app.getName();
        if (name == null) continue;

        if (name.equals(AppenderNames.LOG_FILE_APPENDER)) {
          // Set the threshold level of messages going to the log file to WARNING
          // This will result in NOTICE, INFO, DEBUG, and TRACE messages being
suppressed from going to the server log file
          app.setThreshold(WLLog4jLevel.WARN);
          System.out.println("Set WARNING level on the log file appender");
        } else if (name.equals(AppenderNames.STDOUT_APPENDER)) {
          // Set level to INFO on the stdout filter
          app.setThreshold(WLLog4jLevel.INFO);
          // First clear the existing filters on the appender
          app.clearFilters();
          // Add a filter to block INFO messages from the HTTP subsystem
          app.addFilter(new MyFilter());
        }
      }

      // Now test the filter
      NonCatalogLogger nc = new NonCatalogLogger("MyFilterTest");
      nc.info("INFO messages will not be published to the file but to stdout");
      nc.warning("WARNINFG messages will be published to the file and stdout");


    } catch(LoggerNotAvailableException lex) {
      System.err.println("Log4j logger is not available on this server
    }
  }

  /**
   * Deny messages from the HTTP subsystem of level INFO
   */
  private static class MyFilter extends Filter {
    public int decide(LoggingEvent event) {
      if (event instanceof WLLog4jLogEvent) {
        WLLog4jLogEvent wlsEvent = (WLLog4jLogEvent)event;
        if (wlsEvent.getSubsystem().equals("HTTP")
          && wlsEvent.getSeverity() == Severities.INFO) {
        return DENY;
      }
    }
    return ACCEPT;
  }
 }
}
```

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`.

The following sections describe creating and subscribing a message handler:

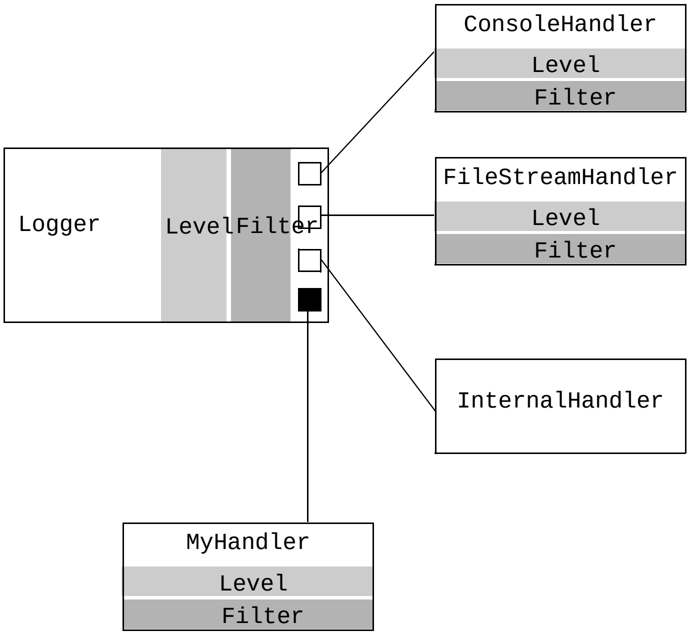
- "Overview of Message Handlers" on page 5-1
- "Creating and Subscribing a Handler: Main Steps" on page 5-3
- "Example: Subscribing to Messages in a Server JVM" on page 5-4
- "Example: Implementing a Log4j Appender Class" on page 5-9
- "Comparison of Java Logging Handlers with JMX Listeners" on page 5-12

For more information about WebLogic Server loggers and handlers, see "The Role of Logger and Handler Objects" on page 4-2.

Overview of Message Handlers

WebLogic Server instantiates and subscribes a set of message 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).

Figure 5-1 Subscribing a Handler



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.

Note: When creating your own message handlers, be careful to avoid executing custom code which runs in the WebLogic Server process before the server initialization has completed and the server has come to a running state. In some cases, custom code can interfere with server services which are being initialized. For example, custom log handlers that make an outbound RMI call which use the `PortableRemoteObject` before the IIOP server service is initialized, can cause server startup to fail.

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. See “Setting a Filter for Loggers and Handlers” on page 4-7.

6. 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 attach a handler to the server `Logger` object.
  - `getDomainLogger` if the current context is the Administration Server and you want to attach a handler to the domain `Logger` object.

    `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.

7. In this class, invoke the `Logger.addHandler(Handler myHandler)` method.
8. Optional. Invoke the `Logger.setFilter(Filter myFilter)` method to set a filter.

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 5-7.

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, see 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, see 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 illustrates the steps described in this section.

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;

   private PreparedStatement stmt = null;

   public MyJDBCHandler() throws NamingException, SQLException {

      InitialContext ctx = new InitialContext();
      DataSource ds = (DataSource)ctx.lookup("myPoolDataSource");
      con = ds.getConnection();
      PreparedStatement stmt = con.prepareStatement
      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 {
          ("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 `Logger` 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, see "Use Custom Classes to Configure Servers" 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();
    }
}
```

Example: Implementing a Log4j Appender Class

The example `Appender` class in Listing 5-3 connects to a JDBC data source and issues SQL statements that insert messages into a database table:

1. Extends `AppenderSkelton`.
2. Constructs a `javax.naming.InitialContext` object and invokes the `Context.lookup` method to look up a data source named `MyDataSource`.
3. Invokes the `javax.sql.DataSource.getConnection` method to establish a connection with the data source.
4. Implements the `append(LoggingEvent event)` method. The method does the following:
    a. Casts each `LoggingEvent` object that it receives as a `WLLog4jLogEvent`.
    b. Uses `WLLog4jLogEvent` methods to retrieve data from the messages.

       For more information about `WLLog4jLogEvent` methods, see the `WLLog4jLogEvent` Javadoc.
    c. Creates a SQL `prepareStatement` and executes the database update whenever a logging event arrives.

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

Table 5-2 Schema for Database Table in Log4j Appender Example

Name	Null?	Type
SERVERNAME		CHAR(30)
MSGID		CHAR(20)
SEVERITYLEVEL		CHAR(20)
LOGGERNAME		CHAR(100)
MESSAGE		VARCHAR(2048)
TIMESTAMP		LONG

5. Implements the `close` method to close the connection with the data source.

Listing 5-3 illustrates the steps described in this section.

Listing 5-3 Example: Log4j Appender Examples Startup

```
package weblogic.logging.examples;

import java.util.Enumeration;
import org.apache.log4j.AppenderSkeleton;
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.Logger;
import org.apache.log4j.spi.Filter;
import org.apache.log4j.spi.LoggingEvent;
import weblogic.logging.LoggerNotAvailableException;
import weblogic.logging.NonCatalogLogger;
import weblogic.logging.Severities;
import weblogic.logging.log4j.AppenderNames;
import weblogic.logging.log4j.Log4jLoggingHelper;
import weblogic.logging.log4j.WLLog4jLevel;
import weblogic.logging.log4j.WLLog4jLogEvent;
import org.apache.log4j.jdbc.JDBCAppender;
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.InitialContext;
import weblogic.logging.log4j.WLLog4jLogEvent;
import weblogic.logging.Severities;

/**
 * This class sets up a Log4j Appender as a listener to the
 * Server Logger for log events.
 */
public class Log4jAppenderExampleStartup {
  public static void main(String[] args) {
    try {
      System.out.println("Invoked the appender example startup class");
      Logger serverLogger = Log4jLoggingHelper.getLog4jServerLogger();
      // Configure the JDBC appender
      MyJDBCAppender jdbcAppender = new MyJDBCAppender();

      // Now add the JDBC appender to the server logger
      serverLogger.addAppender(jdbcAppender);

      // Now test the filter
      NonCatalogLogger nc = new NonCatalogLogger("MyAppenderTest");
      nc.info("Test INFO message");
      nc.warning("Test WARNING message");
    } catch(Exception ex) {
      System.err.println("Init failure " + ex.getMessage());
      ex.printStackTrace();
    }
  }

  private static class MyJDBCAppender extends AppenderSkeleton {

    private Connection connection;
    private java.sql.PreparedStatement stmt;

    public MyJDBCAppender() throws javax.naming.NamingException, SQLException {
      InitialContext ctx = new InitialContext();
      javax.sql.DataSource ds
          = (javax.sql.DataSource) ctx.lookup ("MyDataSource");
      connection = ds.getConnection();
      // Table schema creation SQL command
      // Create table SERVER_LOG (server_name char(30),msg_id char(20),
severity_level char(20),logger_name char(100),message varchar(2048),timestamp
long);
      stmt = connection.prepareStatement("INSERT INTO SERVER_LOG VALUES (?, ?,
?, ?, ?, ?)");
            stmt.setEscapeProcessing(true);
            connection.setAutoCommit(true);
        }
        // Override execute method
        public void append(LoggingEvent event) {
        WLLog4jLogEvent wlsEvent = (WLLog4jLogEvent) event;
        try {
          stmt.setString(1, wlsEvent.getServerName());
          stmt.setString(2, wlsEvent.getId());
          stmt.setString(3,
Severities.severityNumToString(wlsEvent.getSeverity()));
          stmt.setString(4, wlsEvent.getSubsystem());
          stmt.setString(5, wlsEvent.getMessage().toString());
          stmt.setLong(6, wlsEvent.getTimestamp());
          stmt.executeUpdate();
        } catch (SQLException e) {
          System.err.println(e.toString());
        }
      }

      public boolean requiresLayout() {
        return false;
      }

      public void close() {
        try {
          stmt.close();
          connection.close();
        } catch(SQLException sqlex) {
          System.err.println("Error closing JDBC appender");
          sqlex.printStackTrace();
        }
      }
    }
  }
```

Comparison of Java Logging 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 also use Java Logging handlers to receive (subscribe to) log messages.

While both techniques—Java Logging handlers and JMX listeners—provide similar results, the Java Logging 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, see the Sun API documentation for `Formatter`: http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/Formatter.html.

In addition, the Java Logging `Handler` APIs are easier to use and require fewer levels of indirection than JMX APIs. For example, the following lines of code retrieve a Java Logging `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 5-4. The code looks up the `MBeanHome` interface, looks up the `RemoteMBeanServer` interface, looks up the `LogBroadcasterRuntimeMBean`, and then registers the listener.

Optimally, you would use Java Logging handlers to subscribe to log messages on your local machine and JMX listeners to receive log messages from a remote machine. If you are already using JMX for monitoring and you simply want to listen for log messages, not to change their formatting or reroute them to some other output, use JMX listeners. Otherwise, use the Java Logging handlers.

Listing 5-4 Registering a JMX Listener

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

//domain variables
String url = "t3://localhost:7001";
String serverName = "Server1";
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);
    }
}
```