BEA WebLogic Server™®

Understanding the WebLogic Diagnostic Service

Version 9.0 BETA
Document Revised: December 15, 2004
Part Number:

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Introduction and Roadmap

2. Overview of the WebLogic Diagnostic Service

3. WebLogic Diagnostic Service Architecture

D. Configuring and Using Application-Level Instrumentation

E. Accessing Data Online and Offline

F. WebLogic Scripting Tool Examples

Index

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide and the audiences to which it is addressed—*Understanding the WebLogic Diagnostic Service*.

Document Scope

This document describes the functionality and architecture of the WebLogic Diagnostic Service.

Document Audience

This document is written for the following audiences:

- **System administrators**—System Administrators manage and administer the server and applications in the production environment and need to rapidly isolate and diagnose problems. Typically, system administrators use the WebLogic Diagnostic Service to configure diagnostic modules that define the performance and fault information that is to be collected from a running server. System Administrators can also add application-level

instrumentation after the application is built, that is, during deployment process. In addition to configuring the diagnostic data to be collected, system administrators configure the following WebLogic Diagnostic Service components:

  - Watches that are to analyze that information
  - Notification listeners and alarms that fire when watches evaluate to true
  - The archive that is to persist the information to permanent storage
  - The server image capture component that captures server state if the server fails

- **Application Developer**—Developers use the Diagnostic Service to add diagnostic instrumentation to the server and applications that run on the server. Using the instrumentation component, developers can selectively add diagnostic code to specific locations in server and application code. When activated, the diagnostic code monitors performance of the server and applications running on it and generates data that can be used to diagnose problems.
- **Independent software vendor (ISV)—**ISVs can used the WebLogic Diagnostic Service to develop diagnostic tools to collect and analyze diagnostic data from a running server and to add custom diagnostic instrumentation to existing applications.

It is assumed that readers are familiar with Web technologies and the operating system and platform where BEA WebLogic Server® is installed.

Guide to This Document

This document is organized as follows:

- This chapter, "Introduction and Roadmap," introduces the organization of this guide and the audiences to which this guide is addressed.
- Chapter 2, "Overview of the WebLogic Diagnostic Service," provides an overview of the WebLogic Diagnostic Service. It describes the new monitoring and diagnostic features included in this release of WebLogic Server.
- Chapter 3, "WebLogic Diagnostic Service Architecture," describes the architecture of the WebLogic Diagnostic Service and the functions of and interactions between the components.
- Chapter 4, "Terminology," defines key terms that you will encounter throughout the monitoring and diagnostic documentation.

- Appendix A, "WebLogic Diagnostic Service Libraries" the WebLogic Diagnostic Service diagnostic monitor and diagnostic action libraries that ship with the BEA WebLogic Server.
- Appendix B, "Configuring the DyeInjection Monitor and Dye Filtering" describes how to use the DyeInjection Monitor and how to us dye filtering with diagnostic monitors.
- Appendix C, "Configuring the WebLogic Diagnostic Service System Resources" describes how WebLogic Diagnostic Framework (WLDF) components are configured in the new WebLogic Server configuration model.
- Appendix D, "Configuring and Using Application-Level Instrumentation" describes how to add diagnostic instrumentation code to WebLogic Server classes and applications running on the server.
- Appendix E, "Accessing Data Online and Offline" describes how to access data online and offline.
- Appendix F, "WebLogic Scripting Tool Examples" provides examples that demonstrate how to use the WebLogic Scripting Tool.

Related Documentation

- Administration Console Online Help

Diagnostic Service Samples and Tutorials

For WebLogic Diagnostic Service examples, see the following sections:

- "Configuring the WebLogic Diagnostic Service System Resources" on page C-1
- "Configuring and Using Application-Level Instrumentation" on page D-1
- "Accessing Data Online and Offline" on page E-1
- "WebLogic Scripting Tool Examples" on page F-1

New Diagnostic Features in This Release

The WebLogic Diagnostic Service is a new service in this release of WebLogic Server. It provides more diagnostic data, greater visibility into the server containers, broad instrumentation coverage of the server and server applications, cross-container tracing, and an access interface for diagnostic data acquisition.

The following topics describe key functionality of the WebLogic Diagnostic Service:

Visibility of and Access to Diagnostic Data

The WebLogic Diagnostic Service improves the quality and completeness of metrics, data events, and logging information available from the server. This includes both the ability to review existing metrics, data events, and logging information and to look for opportunities to expose new diagnostic data in areas of the server that were not previously exposed.

Logging

WebLogic Logging Service provides all logging events to the WebLogic Diagnostic Service in standard format. In previous releases of WebLogic Server, there were a number of independent logging capabilities, including the standard server and domain logs and access log. These logs relied on different implementations and thus did not benefit from the same services and facilities (such as rotation) as the standard log. These logging solutions are now unified with a single implementation that provides the same qualities of service for all server logging. You can configure the WebLogic Diagnostic Service to collect, analyze, and archive all events generated by the WebLogic Logging Services.

Custom Instrumentation

The WebLogic Diagnostic Service provides a flexible instrumentation component, referred to as a diagnostic monitor, for selectively adding diagnostic code to WebLogic Server and user-written applications running on it. It enables you to select locations in the server and application code at which such code is added and to add diagnostic actions, which are then executed at the selected locations. The combination of code locations and diagnostic actions constitute a diagnostic monitor.

Because the Instrumentation component enables you to add diagnostic code to your own applications, you can collect runtime data events and analyze them to gain a better understanding the state of your applications.

Using a system resource descriptor file, you can configure and deploy diagnostic monitors that generate data events. Using the Administration Console, WLST, or third-party tools, you can configure the WebLogic Diagnostic Service to collect, analyze, and archive data events generated by the diagnostic monitors.

Once the diagnostic monitors are configured and deployed on a running server, you can use the Administration Console, WLST, or third-party tools to dynamically and selectively enable or disable the execution of the diagnostic monitors while the server is running. Also, under certain circumstances, you can dynamically change the behavior of the diagnostic code executed at specific locations while the server is running by changing the diagnostic actions active at those locations.

Diagnostic Context

The WebLogic Diagnostic Service provides a diagnostic context that you can use in the recording of diagnostic events. The diagnostic context provides a means to reconstruct transactional events, as well as a means to correlate events based on the timing of the occurrence or on logical relationships. Using diagnostic context you can reconstruct or piece together a thread of execution from request to response.

WebLogic Server creates, initializes, and populates the diagnostic context when a request enters the system. Various diagnostic components, for example, the logging services and diagnostic monitors, use the diagnostic context to tag generated data events. Using the tags, the diagnostic data can be collated, filtered and correlated by the WebLogic Diagnostic Service and third-party tools.

The diagnostic context also makes it possible to generate diagnostic information only when contextual information in the diagnostic context satisfies certain criteria. This capability enables

you to keep the volume of generated information to manageable levels and keep the overhead of generating such information relatively low.

Request Dyeing

Most event collection is accomplished by designating key points in the application flow and then monitoring every request that passes through those points. However, it is sometimes preferable to capture only events for a single request or requests coming from a single client. This latter approach keeps the volume of information collected to a minimum and allows for easy isolation of events. The WebLogic Diagnostic Service supports the latter approach in that it allows you to mark, or dye, a particular request in such a way that the WebLogic Diagnostic Service can determine whether it should be monitored. The WebLogic Diagnostic Service marks requests when they enter the system by setting flags in the diagnostic context, discussed earlier.

Data Harvesting

The Harvester component can be configured to collect the harvestable data, or metrics, contained by server MBeans. Additionally, because the WebLogic Diagnostic Service enables you to harvest metrics from custom MBeans that you provide, you can also collect metrics from your own applications. Therefore, once your custom MBeans are written and registered with the appropriate MBean server, you can use the Administration Console or the WebLogic Scripting Tool (`weblogic.WLST`) to configure the WebLogic Diagnostic Service to collect, analyze, and archive the metrics that the server MBeans and the custom MBeans contain. Third-party vendors may also develop tools for the same purpose.

Server Image Capture

The Server Image Capture component of the WebLogic Diagnostic Service creates a diagnostic snapshot, or dump, from the server, which you can provide to support personnel for post-failure analysis. You can use this component to capture images on-demand or automatically. The most common sources of server state are captured, such as configuration, Log Cache, WorkManager, JNDI state, and Harvestable data. The Server Image Capture component captures and combines the images produced by the different server subsystems into a single server image file. In addition to capturing the most common sources of server state, this component captures images from all the server subsystems including, for example, images produced by the JMS, JDBC, EJB, and JNDI subsystems.

The Server Image Capture component also includes a First-Failure Notification feature. When a server makes the transition into a failed state, this feature automatically triggers a notification that

in turn triggers the creation and capture of a diagnostic image of server state. Without this feature, there would be no historical record of state of the server when it failed. In addition to ensuring that a historical record is produced, this feature enables system administrators to configure an image-lockout period (a time period that inhibits subsequent image captures) to limit the number of diagnostic images that are captured in a given period of time. When a server fails and recovers multiple times in a short period of time—such as may happen due to storm-related power failures—the image-lockout period prevents the server from repeatedly capturing and persisting diagnostic images that are very similar in content and that unnecessarily use up system resources, such as disk space.

Watches and Notifications

The Watches and Notifications component enables system administrators to configure watches that are capable of detecting specific conditions and triggering notifications. You can configure watches to analyze log records, data events, and harvested metrics. Additionally, you can configure watches to trigger an image capture and different types of notification listeners, including Simple Mail Transfer Protocol (SMTP), Simple Network Management Protocol (SNMP), Java Management Extensions (JMX), and Java Message Service (JMS). The notification listeners communicate messages that a particular watch has triggered, thus enabling the appropriate corrective action be taken.

In many cases, customers have developed software for the trapping, routing, and handling of events from systems within their enterprise. In these cases, the customer may desire a tighter integration than they can achieve by catching traps or sending e-mail when the events occur. Using the Watches and Notifications component, customers can analyze custom application events and automatically generate notifications to alert system administrators, provided the watch rules are based on the supported types of watches. This capability enables a tighter integration with legacy monitoring frameworks in larger customer environments.

The support for additional notification mediums is a beneficial enhancement to the diagnostic capabilities of WebLogic Server. Using the extended JMX services provided in previous releases of WebLogic Server, notifications could be transmitted across local and remote WebLogic servers. For applications that adhere to the JMX specification, support for this type of notification transport is sufficient. However, in many situations it is advantageous to transport notifications across other mediums. For example, in a distributed environment in which multiple WebLogic Server instances (or even non-WebLogic Server instances) are present, it is more appropriate to transport notifications across the wire through an existing protocol that the underlying receiver/sender can more easily accommodate. To satisfy this requirement the WebLogic

Diagnostic Service provides custom notification listeners that can propagate notifications through mediums such as SNMP, SMTP, JMX, and JMS.

Data Archiving

The availability of historical data is usually critical to successfully diagnosing system faults. Therefore, there is a need to capture data and archive it for future access, thereby, creating a historical archive. In the WebLogic Diagnostic Service, the Archiver component meets this need in that is persists all data to permanent storage. The Archiver persists all data events, log records, and metrics collected by the WebLogic Diagnostic Service from server and applications running on it and makes that data available for review. While the Archiver enables you to access diagnostic data in on-line mode (that is, on a running server), you can also access archived data in off-line mode using other file system mechanisms..

Data Accessing

The Data Accessor component provides access to all the data collected by the WebLogic Diagnostic Service, including data events, log records, and harvested metrics, as well as information about the data, such as log files. Using the Data Accessor component, you can access and view historical data. You can access archived data in the on-line mode (that is, on a running server) and off-line mode (that is, after the server shuts down). In the on-line mode, you can use the Administration Console, WLST, or third-party tools to access the archived data, however, in the off-line mode, you must use WLST or other file system mechanisms to access the archived data. System Administrators can access historical data for a server that has shut-down. Further, if the data is stored in a database, you can access it with the tools available with the database without server intervention.

Ease of Development and Integration of Third-Party Analytic Tools

Although no analytic tools are included in this release of WebLogic Server, the WebLogic Diagnostic Service makes it easier for customers and third-party diagnostic tools vendors to develop them. Specifically, the WebLogic Diagnostic Service provides standard integration capabilities for third-party monitoring and analytic tools. This feature includes the following capabilities:

- Standard interfaces and patterns of integration—Third-party tools and clients can collect data in a manner similar to the capability of the WebLogic Sever Administration Console. This capability enables the following:

  - The ability to develop third-party monitoring tools with well-defined publishing interfaces and to integrate those tools with the WebLogic Diagnostic Service
  - The ability of a user-developed administration console to consume and display data
  - The ability of third-party tools to collect data and consume events via JMX, JMS, SMTP, and SNMP

- Historical archive of diagnostic data—The data collected by the WebLogic Diagnostic Service is archived so that 1) the Administration Console and third-party tools can access historical data, and 2) file system mechanisms can have off-line access to historical data for a server that has shut down.

Further, the WebLogic Diagnostic Service is compatible with existing diagnostic capabilities and tools developed by BEA partners. For example, the WebLogic Diagnostic Service is compatible with BEA partners' existing instrumentation capabilities. Rather than limiting the capabilities of BEA partners or rendering their current tools obsolete, the WebLogic Diagnostic Service makes it easier for them to integrate their tools, while BEA Systems takes ownership of the integration interfaces and provides partners with support, documentation, and an improved quality of service.

Standards Support

The WebLogic Diagnostic Service supports the following industry standards:

- Java Message Service (JMS), 1.1
- Java Management Extensions (JMX), 1.2
- JSR 77—JSR (Java Specification Request) 77 defines a statistics provider model for J2EE components.
- Log4j—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/. For information on how to use Log4j with WebLogic Server, see How to Use Log4j with WebLogic Logging Services.
- Simple Mail Transfer Protocol (SMTP)
- Simple Network Management Protocol (SNMP)

CHAPTER 2

Overview of the WebLogic Diagnostic Service

The following sections provide and overview of the WebLogic Diagnostic Service:

What Is the WebLogic Diagnostic Service?

The WebLogic Diagnostic Service is a monitoring and diagnostic service that runs within the WebLogic Server process and participates in the standard server life cycle. This service enables you to create, collect, analyze, archive, and access diagnostic data generated by a running server and the applications deployed within its containers. This data provides insight into the runtime performance of servers and applications and enables you to isolate and diagnose faults when they occur.

The WebLogic Diagnostic Service provides a set of standardized application programming interfaces (APIs) that enable dynamic access and control of diagnostic data, as well as improved monitoring that provides visibility into the server. The interfaces are standardized to facilitate future enhancement and for easier integration of third-party tools, while maintaining the integrity of the server code base. Further, through improvements in WebLogic Server monitoring and diagnostic capabilities, the service reduces customer total cost of ownership and support costs. The service is well suited to the server and the server's stack product components and targets operations and administrative staff as primary users.

In previous releases of WebLogic Server, access to diagnostic data by monitoring agents—which were developed by customers or third-party tools vendors—was limited to JMX attributes, and

changes to monitoring agents required server shut down and restart. However, the WebLogic Diagnostic Service enables dynamic access to server data through standard interfaces and the volume of data accessed at any given time can be modified without shutting down and restarting the server.

Table 2-1 provides a comparison of diagnostic functionality supported in previous releases of WebLogic Server (releases 8.1 and earlier) and this release of WebLogic Server (release 9.0).

Table 2-1 Functionality Comparison of WebLogic Server Releases

Feature Description	Supported in previous releases?	Supported in this release?
A WebLogic Administration Console, which provides the ability to monitor: • Server, domain, and cluster-level statistics • JMS, JDBC, Security, and JTA services	Yes	Yes
A WebLogic Administration Console, which provides full featured data-acquisition and management. The console supports the following features: • Improved data sources for containers and applications • Collection and historical archive of data • An improved data access interface • A built-in monitoring capability with a flexible watch and notification configuration model • Request dyeing and context tracing for end-to-end correlation of diagnostic data	No	Yes
The `WebLogic.Admin` command-line utility, which provides: • Command-line access to data and state • `Ping`, `GetState`, and filtered access to server logs • Thread dumps for deep diagnosis	Yes	Deprecated in this release.
Support for Simple Network Management Protocol (SNMP), which provides full access to the JMX and Log interfaces with support for: • Polling and traps • Gauge monitors, string monitors, and counters • Changing of Monitor attributes	Yes	Yes

Table 2-1 Functionality Comparison of WebLogic Server Releases (Continued)

Feature Description	Supported in previous releases?	Supported in this release?
Support for the following standards: • Simple Mail Transfer Protocol (SMTP) • Java Message Service (JMS) • JSR 77 (Java Specification Request) • Log4j	No	Yes
JMX programming that enables: • The use of Java programs and scripts • Access to the full server configuration interface • Qualities of service, such as strongly-typed interfaces	Yes	Yes
• A JMX services interface that enables the WebLogic Diagnostic Service to be controlled dynamically.	No	Yes
The WebLogic Scripting Tool (`weblogic.WLST`) that supports the functions available through the WebLogic Administration Console plus additional functions.	No	Yes
A unified framework, referred to as the WebLogic Diagnostic Framework, that centralizes and enhances diagnostic functions as follows: • Significantly expands the data set for greater visibility • Enables dynamic, non-disruptive control of data collection • Enables data analysis by means of configured watches, which, when triggered, fire notifications • Provides more efficient data access to historical data through an improved interface	No	Yes

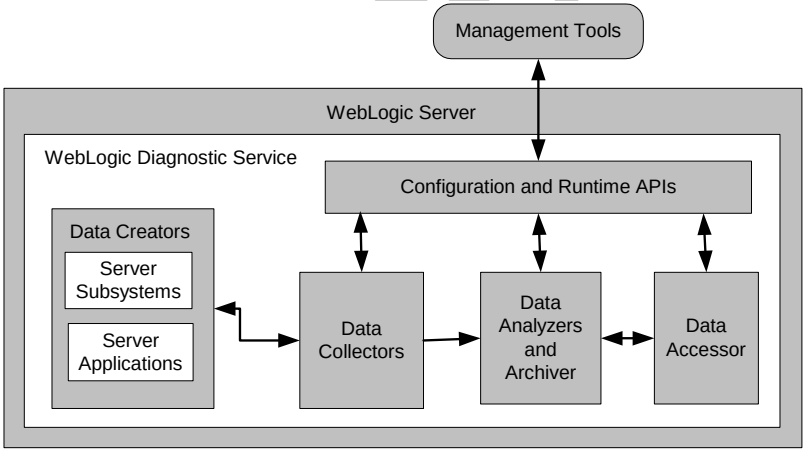
The WebLogic Diagnostic Service - A Simple Overview

The WebLogic Diagnostic Service enables you to perform the following tasks:

- Configure and deploy diagnostic monitors to generate diagnostic data events.
- Configure harvesting of metric data from WebLogic Server MBeans and custom MBeans.
- Configure watches to trigger notifications based on metric data, data events, server log data.
- Write and deploy custom MBeans that contain harvestable data that can be collected.
- Configure logging services to generate log records.
- Configure and deploy diagnostic modules to managed servers to collect, analyze, archive, and access diagnostic data.
- Configure diagnostic server image captures and initiate image captures on demand.

This WebLogic Diagnostic Service collects diagnostic data from WebLogic Server subsystems and applications deployed within the server containers (see Figure 2-1).

Figure 2-1 WebLogic Diagnostic Service - A Simple View



Each component performs a critical function in providing useful and manageable diagnostic data:

- The Configuration and Runtime application programming interfaces (APIs) enable you to use management tools, such as the WebLogic Server Administration Console, the

WebLogic Scripting Tool (`weblogic.WLST`), and third-party tools, to configure and monitor the WebLogic Diagnostic Service runtime. Using a management tool, you can configure a diagnostic module on an Administrative server and then deploy that module to one or more managed servers or clusters of managed servers. Once the diagnostic module is deployed, you can use the management tool to modify the diagnostic configuration on a running server without having to shut down and restart the server.

- Data Creators generate diagnostic data and make it available to the Data Collectors for collection.
- Data Collectors:
  - Format the diagnostic data specifically configured for collection and pass it to the Data Analyzers and Archiver component. Some data, such as data events, does not require formatting and is simply forwarded.
  - If the server fails, create an image of server state and persist it to permanent storage.
- The Data Analyzers and Archiver component:
  - Using configured watches, analyzes all the data it receives and, when a watch is triggered, fires configured notifications, alarms, and server image captures.
  - Makes historical data available to the Data Accessor component and archives the data received to permanent storage.
- The Data Accessor component enables you to view historical data. While you can view historical data in either on-line or off-line mode, off-line access is restricted to the local machine, that is, the machine on which the managed server was running.

The following sections provide more information on the WebLogic Diagnostic Service:

- "Configuration and Runtime APIs" on page 2-5
- "Data Creators" on page 2-6
- "Data Collectors" on page 2-7
- "Data Analyzers and Archiver" on page 2-8
- "Data Accessor" on page 2-9

Configuration and Runtime APIs

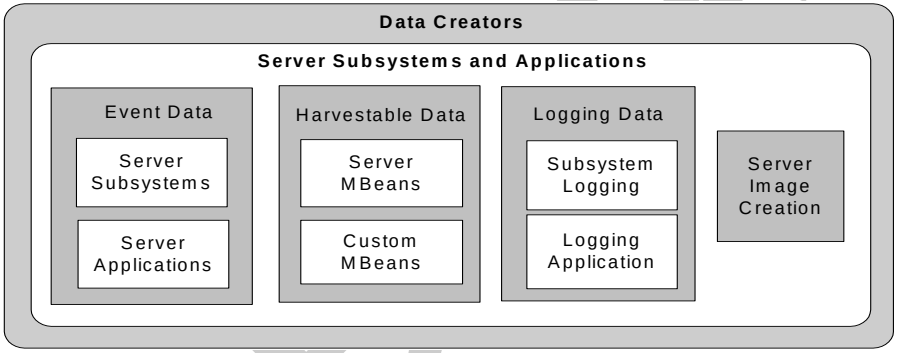
The Configuration and Runtime APIs enable you to perform configuration and monitoring tasks on the WebLogic Diagnostic Service. There are two sets of APIs: configuration APIs and runtime

APIs. The configuration APIs enable you to configure and control the runtime behavior of the WebLogic Diagnostic Service components. The runtime APIs enable you to monitor the runtime state and the operations defined for the different components of the WebLogic Diagnostic Service. Both the configuration and the runtime APIs are exposed as MBeans. You can access the APIs, using the WebLogic Server Administration Console, the WebLogic Scripting Tool, third-party tools, and programmatically.

Data Creators

Data Creators generate the diagnostic data that is used to diagnose problems. The WebLogic Diagnostic Service generate four types of diagnostic data: Event Data, Harvestable Data, Logging Data, and Server Image Creation (see Figure 2-2).

Figure 2-2 Data Creators Components



- Event Data—Diagnostic monitors instantiate actions at well defined points in the flow of execution of server code and application code. The diagnostic monitors generate events and send them to the Data Collector. The WebLogic Diagnostic Service has an instrumentation component that enables you to add diagnostic monitors at locations in server subsystems and applications running on it.
- Harvestable Data—Harvestable data can be created by server MBeans and user-written custom MBeans. By default, server MBeans produce harvestable data. Additionally, users can write custom MBeans to generate additional harvestable data for their own applications. You can configure the WebLogic Diagnostic Service to access and harvest the harvestable data.
- Logging Data—The logging services generate log records, which are presented to the Data Collectors for collection. The WebLogic Logging Service provides log messages that were

generated by servers, server subsystems, and J2EE applications that run on WebLogic Server server or client JVMs.

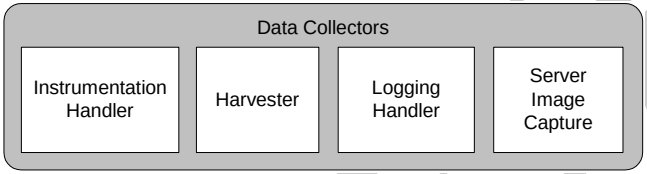
- Server Image Creation—The Server Image Creation component creates diagnostic snapshots, or dumps, of server state and provides them to the Data Collectors.

Data Collectors

Data Collectors gather diagnostic data from the Data Creators. Data Collectors consists of four components: Instrumentation Handler, Harvester, Logging Handler, and Server Image Capture (see Figure 2-3).

Each of these components provides a different type of diagnostic data.

Figure 2-3 Data Collectors Components



- Instrumentation Handler—The Instrumentation Handler handles events data generated by diagnostic monitors in the server subsystem and in applications running on the server.
- Harvester—The Harvester collects harvestable data from Server MBeans and, if provided, user-written custom MBeans. Using the Administration Console, WLST, or third-party tools, you can configure the Harvester to collect the data, reformat it, and forward it to the Data Analyzers and Archiver component. You can also configure the sampling rate, which determines the time interval, in seconds, between MBean data accesses.
- Logging Handler—The Logging Handler handles log records generated by the WebLogic Logging Service. Using the Administration Console, WLST, or third-party tools, you can configure the Logging Handler to send the log records to the Data Analyzers and Archiver component.
- Server Image Capture—The Server Image Capture component captures the images produced by the Server Image Creation component and packages them into a single file. This file is intended to be used by support personnel for post-failure analysis. Using the Administration Console, WLST, or third-party tools, you can configure the image destination directory, the time allowed to complete image captures, and the image capture

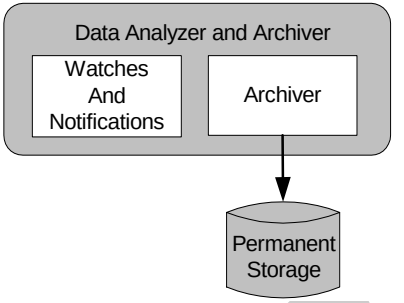
lockout period (the required elapsed time between image captures). You can also initiate server image captures on demand.

The Server Image Capture component is an invaluable tool for diagnosing server failures. By automatically capturing diagnostic images when the server enters a failed state, this component ensures that the diagnostic data needed to diagnose server failures is never lost.

Data Analyzers and Archiver

The Data Analyzers and Archiver processes, analyzes, and archives the diagnostic data received from the Data Collectors. The Data Analyzers and Archiver consists of two components: the Watches and Notifications and the Archiver (see Figure 2-4).

Figure 2-4 Data Analyzers and Archiver Components



The following section describe these components:

- "Watches and Notifications" on page 2-8
- "Archiver" on page 2-9

Watches and Notifications

The Watches and Notifications component enables you to configure automated watches that analyze specific diagnostic state and send notifications when the watches are triggered. The Watches and Notifications component processes and analyzes the data events, log records, and metrics that it receives from the Data Collectors. This component compares the diagnostic data to watches.

The watches are user-written, watch-rule expressions. Using an expression language, you can construct watch-rule expressions that specify relationships among multiple attributes of multiple MBeans. Watches can be composed of a number of watch rules.

You can configure notifications for each watch. Once configured, notifications fire when a specific watch evaluates to true. You can configure notifications to fire through four different listener mediums: SNMP, JMS, JMX, and SMTP. And, you can configure multiple notification listeners for each watch. You can also configure a watch to initiate a server image capture when the watch evaluates to true. You can configure watches, notification mediums, and alarms, using the Administration Console, the WebLogic Scripting Tool, third-party diagnostic tools, and programmatically.

Archiver

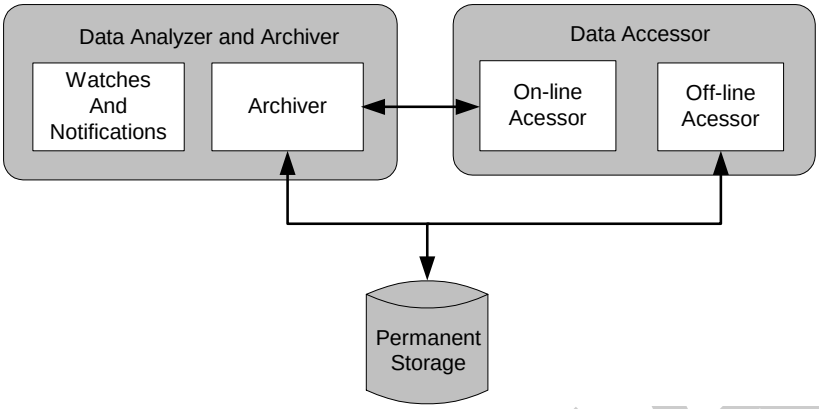
The Archiver processes all diagnostic data presented to it by the Data Collectors and archives it to permanent storage for future access, thereby, creating a historical archive. All data events, log records, and harvested metrics are archived. The Archiver instantiates the archive on a per-server basis. You can configure the Archiver to persist data to a file or to a JDBC database.

System administration tasks are minimal with respect to the Archiver. You can configure the diagnostic data archive type and the name of the data storage mechanism in which the data is stored. The data store can be either the directory where the diagnostic data is to be maintained or the JDBC data source. You can also use the Archiver to check how long the server takes to archive data and thereby assess the demands the Archiver places on system resources. You can use programmatic access, the Administration Console, the WebLogic Scripting Tool, or third-party tools to configure and monitor these settings.

Data Accessor

With the exception of the server image, the Data Accessor provides access to all the data collected by the WebLogic Diagnostic Service, including logs records, data events, and harvested metrics (see Figure 2-5).

Figure 2-5 Data Accessor Components



For ease of access, the Data Accessor provides a single access point and supplies data on demand. Therefore, you can access diagnostic data in a production environment on an as needed basis. Further, the Data Accessor enables you to look up data by type, by component, and by attribute. It permits you to perform time-based filtering and, when accessing events, filtering by severity, source, and content. The Archiver component also maintains different types of diagnostic data in separate data stores that is modeled as tabular format with rows and columns. Therefore, you can also use the Data Accessor to access diagnostic data in tabular format.

The Data Accessor consists of two components: an On-line Accessor and an Off-line Accessor.

- On-Line Accessor—On a running server, users can access the diagnostics information exposed by the Data Accessor using the Administration Console, the WebLogic Scripting Tool, or a third-party tool. You can use the On-line Accessor to get the archived historical data. The On-line Accessor accesses historical data through the Data Analyzers and Archiver component (see Figure 2-5).
- Off-Line Accessor—Should a server shut down, you can access historical data, such as archived metrics and log and event files, through the Off-line Accessor. The Off-line Accessor accesses the data in permanent storage directly and supports the use of file system access tools to access and analyze diagnostic data. The only limitation to off-line access is that the archived data can be accessed only from the machine on which the server was running.

CHAPTER 3

WebLogic Diagnostic Service Architecture

This section covers the following topics:

What is the WebLogic Diagnostic Framework?

The WebLogic Diagnostic Framework (WLDF) is the foundation upon which the WebLogic Diagnostic Service is built. The framework exhibits the following characteristics:

- **Is unified**—The framework pulls all diagnostics features and functionality together under one subsystem and is flexible so as to support future innovations, such as system analysis and feedback control that is required for adaptive systems management.
- **Enables incremental functionality enhancements**—The framework enables a planned, phased rollout. It provides the most valuable core functionality in its initial release and enables functionality to be added to future releases in a controlled and reasonable way.
- **Is extensible**—The framework is extensible is two respects: 1) it allows for expansion for both the data and functionality available in the initial core offering, and 2) it enables application developers and tools vendors to easily integrate and extend the framework with advanced tools.
- **Enables dynamic changes on a running server**—The framework enables activation and control of all key diagnostic features dynamically in a running server environment such

that the most significant diagnostic data is provided to users. And, the diagnostic data is provided to users in a state that allows for thorough and timely diagnostic investigation.

- **Minimizes performance overhead**—The framework introduces minimal overhead when inactive and reasonable overhead when significant diagnostic data collection is in effect.
- **Is not participatory**—The framework is not involved in the handling of application requests and has no opportunity to influence or modify request state. Rather, it exists beside the application request and observes and records diagnostic information for problem diagnosis.
- **Does not mandate diagnostic data consumption tools**—The framework focuses on capturing and providing diagnostic data. Therefore, the framework does not contain or mandate any of the tools—whether scripting-based or visual—that may be used to collect, analyze, archive, or access the diagnostic data.

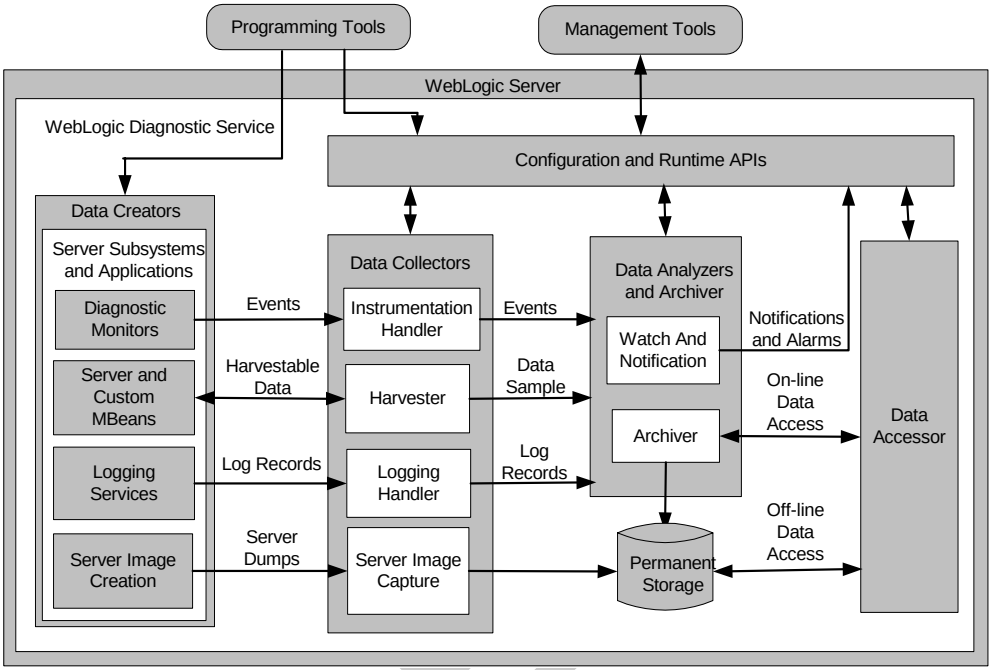
The framework components that are configurable have these attributes:

- Operate at the server level and are only aware of server scope
- Exist entirely within the server process and participate in the standard server lifecycle
- Are stored on a per-server basis

How the WebLogic Diagnostic Service Works

The WebLogic Diagnostic Service partitions diagnostic data creation and processing into four major components: Data Creators, Data Collectors, Data Analyzers and Archiver, and Data Accessor (see Figure 3-1). To configure and monitor these components, configuration and runtime APIs are provided and programming tools are supported.

Figure 3-1 WebLogic Diagnostic Service Components



This section covers the following topics:

Programming Tools

The WebLogic Diagnostic Service enables you to perform the following tasks programmatically:

- Use the diagnostic descriptors in the application archive to configure application-level diagnostic monitors.
- Use the WebLogic Diagnostic Framework system resource descriptor to configure server-level diagnostic monitors that generate events data. You can then configure the Instrumentation Handler to collect that data.
- Use JMX to create custom MBeans that contain harvestable data. You can then configure the Harvester to collect that data.
- Write Java programs that perform the following tasks:
  - Capture notifications using JMX listeners
  - Access data through the Data Accessor

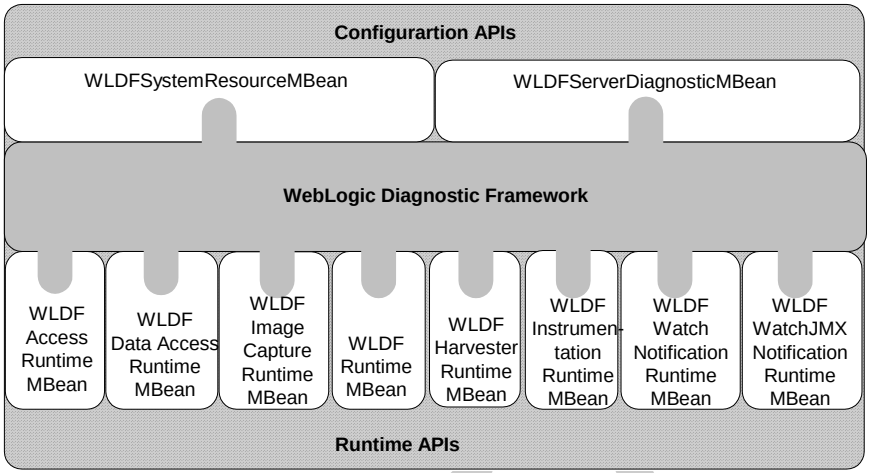
For information on using JMX with the WebLogic Diagnostic Service, see *Developing Manageable Applications with JMX*.

Configuration and Runtime APIs

You use the configuration and runtime APIs to configure and monitor the WebLogic Diagnostic Service. The APIs enable you to access the configurable components of the WebLogic Diagnostic Service. Using the APIs, you can configure, activate, and deactivate data collection, configure watches, notifications, alarms, and server image captures, and access data.

Figure 3-2 illustrates the interaction between the Configuration and Runtime APIs and the WebLogic Diagnostic Service. You use the configuration APIs to configure and control the runtime behavior of the WebLogic Diagnostic Service. You use the runtime APIs to monitor the runtime state and the operations defined for the different components of the service. Both the configuration and the runtime APIs are exposed as MBeans. You can use administrative clients such as the WebLogic Server Administration Console, the WebLogic Scripting Tool (`weblogic.WLST`), or diagnostic tools provided by third-party vendors to access these APIs.

Figure 3-2 Configuration and Runtime APIs



This section covers the following topics:

- "Configuration APIs" on page 3-5
- "Runtime APIs" on page 3-7

Configuration APIs

The Configuration APIs define interfaces that are used to configure the following WebLogic Diagnostic Service components (see Figure 3-3):

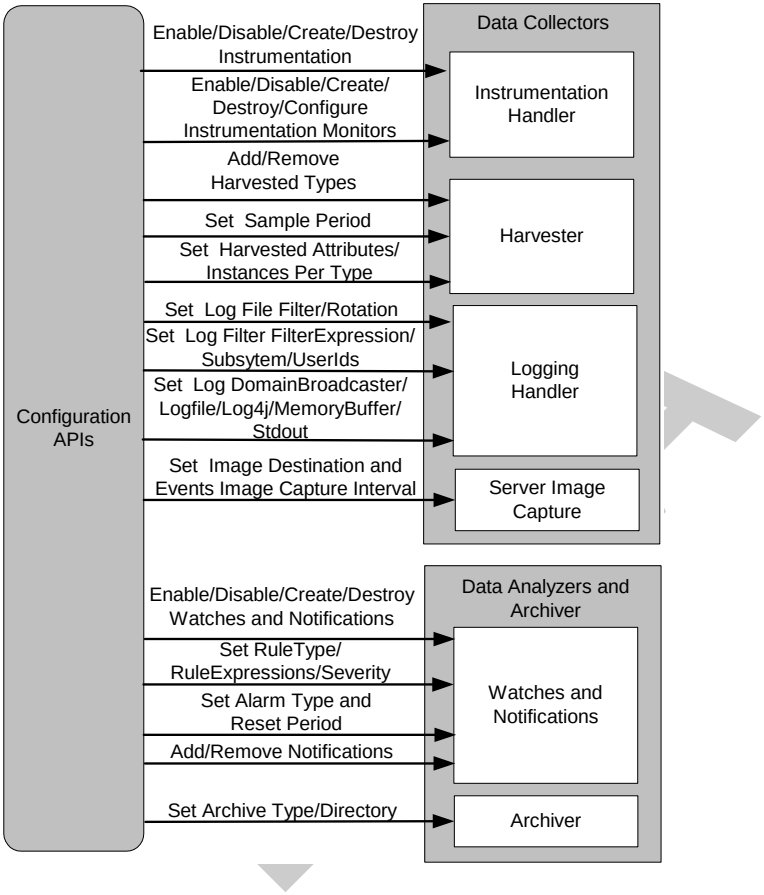
- Data Collectors—You can use the configuration APIs to configure and control Instrumentation, Harvesting, and Server Image Capture.
  - For the Instrumentation Handler, you can enable, disable, create, and destroy server-level instrumentation and instrumentation monitors.

    Note: The configuration APIs do not support configuration of application-level instrumentation. However, configuration changes for application-level instrumentation can be effected using Java Specification Request (JSR) 88 APIs.
  - For the Harvester, you can add and remove types to be harvested, specify which attributes and instances of those types are to be harvested, and set the sample period for the harvester.

  - For the Logging Handler, you can set the file and rotation parameters for log file, set the filter expression for the log filter, and set the log file, broadcaster, Log4j, memory buffer, and standard out parameters for the log.
  - For the Server Image Capture, you can set the name and path of the directory in which the image capture is to be stored and the events image capture interval, that is, the time interval during which recently archived events are captured in the diagnostic image.

- Data Analyzers and Archiver—You can use the configuration APIs to configure watches, notifications and alarms. You can enable, disable, create, and destroy watches and notifications. You can also use the configuration APIs to:
  - Set the rule type, watch-rule expressions, and severity for watches
  - Set alarm type and alarm reset period for notifications
  - Configure a watch to trigger a server image capture
  - Add and remove notifications from watches
  - Set the archive type and the archive directory

Figure 3-3 Configuration Capabilities of the Configuration APIs



Runtime APIs

The runtime APIs define interfaces that are used to monitor the runtime state of the different WebLogic Diagnostic Service components. Instances of these APIs are instantiated on instances of individual managed servers. These APIs are defined as runtime MBeans so JMX clients can easily access them.

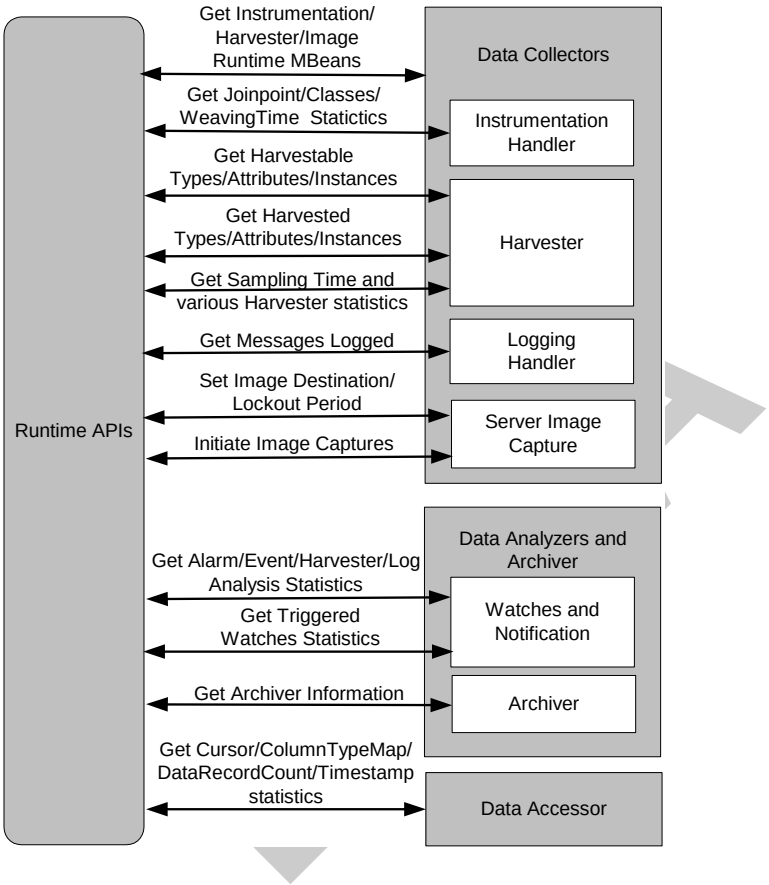
The Runtime APIs encapsulate all other runtime interfaces for the individual WebLogic Diagnostic Service components. These APIs are included in the `weblogic.management.runtime` package.

You can use the runtime APIs to monitor the following WebLogic Diagnostic Service components (see Figure 3-4):

- Data Collectors—You can use the runtime APIs to monitor the Instrumentation Handler, the Harvester, the Logging Handler, and the Server Image Capture components.
  - For the Instrumentation Handler, you can monitor the instrumentation MBean, joinpoint count statistics, the number of classes inspected for instrumentation monitors, the number of classes modified, and the time it takes to inspect a class for instrumentation monitors.
  - For the Harvester, you can query the set of harvestable types, harvestable attributes, and harvestable instances (that is, the instances that are currently harvestable for specific types). And, you can also query which types, attributes, and instances are currently configured for harvesting. The sampling interval and various runtime statistics pertaining to the harvesting process are also available.
  - For the Logging Handler, you can get the messages logged by the log broadcaster for JMX notifications.
  - For Server Image Capture, you can specify the destination and lockout period for diagnostic images and initiate image captures.
- Data Analyzers and Archiver—You can use the runtime APIs to monitor the Watches and Notifications and Archiver components.
  - For Watches and Notifications, you can reset watch alarms, attach JMX notification listeners, and monitor statistics about watch-rule evaluations and watches triggered, including information about the analysis of alarms, events, log records, and harvested metrics.
  - For the Archiver, you can monitor information about the archive, such as file name.
- Data Accessor—You can use the runtime APIs to retieve the diagnostic data persisted in the different archives. The runtime APIs also support data filtering by allowing you to specify a query expression to search the data from the underlying archive.
- You can monitor information about column type maps (a map relating column names to the corresponding type names for the diagnostic data), statistics about data record counts and timestamps, and cursors (cursors are used by clients to fetch data records).

For information on how to perform configuration and monitoring tasks using the WebLogic Scripting Tool, see *WebLogic Scripting Tool*.

Figure 3-4 Monitoring Capabilities of the Runtime APIs



Data Creators

Diagnostic data is created by a number of WebLogic Server subsystems. Programmatically you can also add code to server subsystems and your own applications so as to generate additional diagnostic data.

Diagnostic data describes the state of a running server and the applications running on that server. As illustrated in Figure 3-1, the WebLogic Diagnostic Service receives diagnostic data from four different types of data creators: Diagnostic Monitors, Server and Custom MBeans, Logging Services, and Server Image Creation.

This section covers the following topics:

Diagnostic Monitors

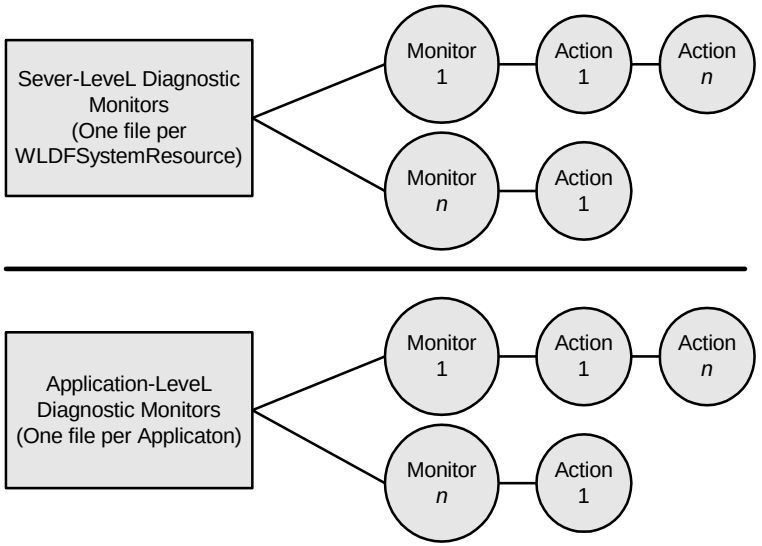
The WebLogic Server product includes server-level diagnostic monitors that generate data events that are provided to the Data Collectors for collection (see Figure 3-1). The WebLogic Diagnostic Service also provides an Instrumentation component that enables you to configure and deploy server-level and application-level monitors which are available in the monitor library. In addition, it allows you to define and configure your own custom monitors which can be used at application level. These monitors generate additional data events and provide them to the Data Collectors for collection.

Diagnostic monitors identify the location in the program that they are to affect and the diagnostic action that is to be executed at that location. The locations where diagnostic code is added are called diagnostic joinpoints. The Instrumentation component enables you to identify such diagnostic joinpoints with an expression in a generic manner. A set of joinpoints, identified by such a generic expression, is called a pointcut. Some diagnostic monitors can only be applied to server classes while others can only be applied to applications. Therefore, the instrumentation component makes a distinction between monitors that are scoped to WebLogic Server classes and those that are scoped to application (non-WebLogic Server) classes.

To enable you to configure and deploy diagnostic monitors, the Instrumentation component supports use of a system resource descriptor file. Using descriptors, you can add diagnostic monitors to a server's subsystems and the applications running on the server. The descriptors enable you to define specific locations in the server and application code at which the specified diagnostic monitors are executed. Two levels of descriptors are supported: server-level and application-level.

Figure 3-5 shows the hierarchy of diagnostic monitors. Notice that at each level, server and application, the descriptor file can include multiple monitors and each monitor can have multiple actions defined for it.

Figure 3-5 Hierarchy of Diagnostic Monitors



Because diagnostic monitors are activated at server startup, you can tune the WebLogic Diagnostic Service at a customer's site according to specific needs. Once the server is started and a diagnostic monitor is activated, you can dynamically disable and enable the execution of the diagnostic monitor while the server is running. Thus, you have the option of activating the diagnostic monitors only when problems need to be diagnosed. Further, under certain circumstances, you can change the behavior of the diagnostic monitors while the server is running.

The WebLogic Diagnostic Service supports three categories of diagnostic monitors: standard, delegating, and custom (see Table 3-1). All Diagnostic monitors have two components: 1) the pointcuts used by the monitors—which identify the locations in the code where diagnostic code is to be added, and 2) the business logic, or the diagnostic actions—which specify the diagnostic activity that takes place at those locations. As described in Table 3-1, a standard monitor uses a fixed pointcut and provides a fixed diagnostic action. Thus, both components, the pointcut and action, are preset in standard monitors and cannot be changed. Delegating monitors use a fixed pointcut, but you can configure their diagnostic action. Custom monitors allow a you to configure a pointcut and the action(s) to be used with it

Note: You can only create custom monitors with application instrumentation, not with server instrumentation.

Table 3-1 Diagnostic Monitor Descriptions

Monitor Category	Pointcut	Diagnostic Action
Standard Monitor	Fixed	Fixed
Delegating Monitor	Fixed	Configurable
Custom Monitor	Configurable	Configurable

This section covers the following topics:

Standard Monitors

The standard monitors provided in the diagnostic monitor library perform specific diagnostic actions at specific pointcuts. In other words, they identify the locations in the program that will be affected by the instrumentation and the actual diagnostic functionality that is performed at those locations. Once instrumented and activated on a running server, you can dynamically enabled or disabled standard monitors while the server is running. However, you cannot change the behavior of standard monitors. Each standard monitor is known by its type. A standard monitor of a particular type can be applied to more than one level, or scope, server or application, however, within a scope, you can only apply a standard monitor of a particular type once. At runtime, active standard monitors are identified by their scopes and types.

When standard monitors are disabled at runtime, they introduce minimal performance overhead. To remove a standard monitor completely from an application scope, it is necessary to remove it from the configuration and redeploy the application. To remove a standard monitor completely from a server, it is necessary to remove it from the configuration and restart the server.

Delegating Monitors

Delegating monitors provide more flexibility than standard monitors. Similar to standard monitors, delegating monitors can be enabled or disabled at runtime without restarting the server

or redeploying applications. However, unlike standard monitors, you can change the behavior of delegating monitors behavior by changing the set of diagnostic actions associated with them. You can effect such changes in diagnostic behavior without restarting the server or redeploying the application. Moreover, you can attach multiple diagnostic actions to delegating monitors so that divergent diagnostic functions can be executed at the same locations, or pointcuts, without affecting each other. When you attach multiple actions to a delegating monitor, the actions are executed in the same order in which they were registered with the monitor.

Implementation of delegating monitors may have certain structural requirements. For example, some monitors may affect locations at method entries. Others may affect method entries as well as method exits. Consequently, delegating monitors may require that they be used with only certain types of actions from the action library, and delegating monitors must be compatible with their associated actions.

Delegating monitors work on certain standard pointcuts such as J2EE pointcuts for applications. When delegating monitors are disabled at runtime, they introduce minimal performance overhead. In order to remove them completely from an application scope, it is necessary to remove them from the configuration and redeploy the application. To remove delegating monitors completely from server scope, it is necessary to remove them from the configuration and restart the server.

Custom Monitors

Standard and delegating monitors work on specific standard pointcuts identified during their development and the pointcuts cannot be changed. They may satisfy most of your diagnostic requirements. However, there may be situations when you need to specify other locations in applications where you want to add diagnostic code. Custom monitors satisfy this need.

Custom monitors provide the means to execute diagnostic actions at specific locations in application code. These locations can be identified while configuring custom monitors by specifying appropriate pointcuts. As part of defining a custom monitor, the following information is required:

- Name—Display name of the custom monitor, which is used for control purposes.
- Location—The relative locations where the diagnostic code is to be added. Possible values are “before”, “after”, and “around”.
- Pointcut—An expression, which identifies the set of locations in the applications that are affected.
- Action—The diagnostic action that is executed at the pointcut.

The pointcut syntax identifies a method execution location, or a method call-site location. The pointcut syntax is described in *Configuring the WebLogic Diagnostic Service (This document is to be supplied)*.

Once activated, custom monitors can be enabled or disabled on a running server. When disabled, they introduce minimal performance overhead. In order to remove them completely, it is necessary to remove them from the configuration and redeploy the application. Similarly, a newly configured or updated custom monitor takes effect only after the application is redeployed.

A custom monitor accepts only actions of compatible types. Depending on their location, an attribute specific subset of actions can be used with custom monitors.

Note: You can only use custom monitors to instrument your applications The WebLogic Diagnostic Service does not allow custom monitors to be used to instrument WebLogic classes on servers.

To assist you in adding diagnostic monitors to server and application code, libraries of diagnostic monitors and actions are provided in the WebLogic Server product distribution kit. For a description of the diagnostic monitors and diagnostic actions provided in the libraries, see Appendix A, "WebLogic Diagnostic Service Libraries."

Diagnostic Actions

Delegating and custom monitors do not provide any diagnostic functionality of their own. The diagnostic functionality is provided by the diagnostic actions, which you attach to them. A library of diagnostic actions is provided with the WebLogic Server product. You can use these diagnostic actions with the delegating monitors provided in the diagnostic monitor library and any custom monitors that you may write.

Depending on the functionality of a diagnostic action, it may need a certain environment to do its job. Such an environment is provided by the delegating or custom monitor to which the diagnostic action is attached. Therefore, diagnostic actions can be used only with compatible monitors. To ensure that there is no mismatch between diagnostic monitors and diagnostic actions, diagnostic actions are classified by their types, which are then used by the WebLogic Diagnostic Service to determine compatibility.

To assist you in adding diagnostic monitors to server and application code, libraries of diagnostic monitors and actions are provided in the WebLogic Server product distribution kit.

Dye Filtering and Dye Mask

The diagnostic activity of an action is executed only if certain filtering conditions are met based on the dye vector in the diagnostic context. The dye vector is typically set by the

DyeInjectionMonitor diagnostic monitor when the request enters the server. The dye vector carries request characteristics as the request progresses through its execution path. If dye filtering for the diagnostic monitor is disabled, its business logic is unconditionally executed. Otherwise, the current dye vector in the diagnostic context must be consistent with the dye mask configured with the monitor. In other words, the business logic of the monitor will be executed only if the dye vector in the diagnostic context satisfies the following condition:

(dye_mask & dye_vector == dye_mask)

By properly configuring and deploying the DyeInjectionMonitor and configuring the dye mask of the monitors, you can configure diagnostic monitors to execute only during requests of special interests. For example, you could configure a test request to fire for a test machine and have its progress analyzed. This approach facilitates quick analysis of diagnostic data. It also ensures that other requests are not slowed down by diagnostic activity.

Server MBeans and Custom MBeans

WebLogic Server MBeans, which are provided with the WebLogic Server product, provide metrics for discrete data points in WebLogic Server that can be harvested by the Harvester (see Figure 3-1). Users can also write custom MBeans to generate metrics for harvesting. In order to be visible to the Harvester, custom MBeans must be registered with the local WebLogic runtime MBean Server.

Logging Services

Logging Services generate log records that are produced by WebLogic Logging Services (see Figure 3-1).

Note: You can only use the Data Accessor to access server-side log records. The WebLogic Diagnostic Service does not support access to client-side log records through the Data Accessor.

WebLogic Logging Services provide facilities for writing, viewing, filtering, and listening for log messages. The log messages are generated by servers, server subsystems, and J2EE applications that run on WebLogic Server client or server JVMs. Additionally, you can create your own catalog of log messages. The log messages are collected by the WebLogic Diagnostic Service as events. For more information about WebLogic Logging Services, see "Understanding WebLogic Logging Services" in *Configuring Log Files and Filtering Log Messages*.

Server Image Creation

The Server Image Creation component (see Figure 3-1) creates a diagnostic snapshot, or dump of server state whenever the server enters a failed state. This component creates separate images of the most common sources of server state, such as configuration, Log Cache, WorkManager, Harvestable data, and server subsystems, such as JNDI, JDBC, and EJB. As the separate images are created, they are presented to the Server Image Capture component for collection.

Data Collectors

The Data Collectors component consists of four sub-components (see Figure 3-1): Instrumentation Handler, Harvester, Logging Handler, and Server Image Capture. Each of these components collects a different type of diagnostic data.

This section covers the following topics:

Instrumentation Handler

The Instrumentation Handler receives all data events generated by the diagnostic monitors. The Instrumentation Handler uses a data-push model to collect events data. Using the data-push model, the diagnostic monitors send, or push, events data to the Instrumentation Handler as the events are generated. The handler then forwards all the data events that are configured for collection as they occur, without modification, to the Data Analyzers and Archiver.

Harvester

The Harvester accesses and retrieves the harvestable metrics that are configured for collection by the WebLogic Diagnostic Service. The Harvester uses a data-pull model to collect harvestable data. Using the data-pull model, the MBeans make harvestable data available to the Harvester, but the Harvester accesses the MBeans to collect, or pull, the data. The Harvester then reformats the data into a collective data sample and sends the sample to the Data Analyzers and Archiver.

Logging Handler

The Logging Handler receives all log records generated by the WebLogic Logging Services. The Logging Handler uses the data-push model to collect log records. Using the data-push model, the WebLogic Logging Service sends, or pushes, the log records to the Data Collectors as they are generated. The Logging Handler then forwards all the log records that are configured for collection as they occur, without modification, to the Data Analyzers and Archiver.

Server Image Capture

The Server Image Capture component receives a diagnostic image from the server whenever the server enters a failed state. When configured by the WebLogic Diagnostic Service to capture server images, the Server Image Capture component combines those separate images into a single file, called a diagnostic image, and writes that file to permanent storage.

The Server Image Capture component consists of these sub-components: Image Manager, First Failure, Image Sources, and Image Data (see Figure 3-6).

This section covers the following topics:

Figure 3-6 Server Image Capture Components

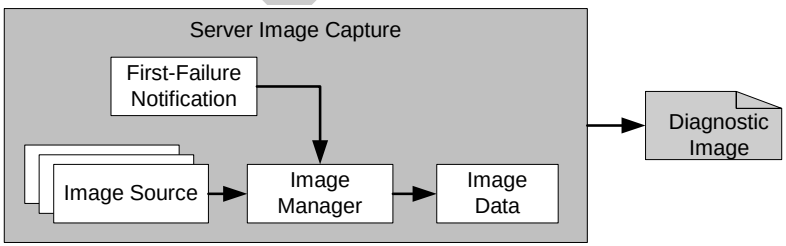


Image Manager

The Image Manager manages image sources and the creation of diagnostic images. The Image Manager is the singular and central resource for all diagnostic image handling. It is a singleton in each process in which it is utilized. The image capture functionality may be invoked by a First-Failure Notification, by a configured watch notification, on-demand by a direct API call to the WLDFImageRuntimeMBean, or by push button functionality provided in third-party diagnostic tools.

To create a diagnostic image the Image Manager invokes each image source that has been registered. If a particular image source fails, the Image Manager notes this failure and continues processing the remaining image sources. The Image Manager adds image data (summary information about the image) to the image capture. Additionally, it provides a means of modifying the configured destination of the output image artifact. This value is persisted and survives server restart. If the Image Manager is unable to write to the specified directory or fails during writing, it generates an exception. During initialization the Image Manager tests the configured destination directory to determine whether it is write protected. If the test fails, an error message is logged.

To enable users to control the image capture function and manage performance, the Image Manager supports the following features:

- **Image Capture Timeout**—The image capture time-out is used by the Image Manager to determine whether an image source has spent too much time creating its image. If the time-out period is exceeded, the Image Manager sends a time-out signal to the Image Source. The source is not required to react immediately to the time-out signal, however, any work completed after the time-out signal is sent may be disregarded by the Image Manager. Further, the Image Manager may choose to close the output stream provided to the image source at any time after a time-out call is made. This allows the Image Manager to gracefully handle Image Sources that may be compromised by the current failure state or that are exceeding defined performance requirements.

- **Image Capture Lockout Period**—Because the creation of a diagnostic image may be costly in terms of machine resources, it should be a task that is executed infrequently. Creating an image for each error detected is neither efficient nor more effective than a single image created on the first failure. Therefore, the WebLogic Diagnostic Service imposes a lockout period by default that prohibits the Image Manager from creating a secondary image until after some period of time. Capture requests that occur during a lockout result in a CaptureLockoutException, which can be safely ignored unless in fact there is proof that no such image has been created. This functionality prevents a cascade of errors generating many similar diagnostic images. If an image capture fails for any reason

(including input/output problems or a fatal error in a registered image source) the lockout is not set and the Image Manager is immediately ready to handle another capture request.

- **Log Messages**—The Image Manager logs the beginning of the diagnostic image capture and the completion of the capture as two distinct events. The completion log message includes the time elapsed during image creation. Using the Watch and Notification capabilities of the WebLogic Diagnostic Service, the Log Messages feature makes it possible to watch for these messages and create a notification (for example, using e-mail) that an image capture has occurred.

First-Failure Notification

First-Failure Notification is a mechanism that detects when a server is making the transition into a failed state and triggers the creation of a diagnostic image to capture the state of the server. Because a server may fail multiple times in a short period of time (for example, due to power failures), once the First-failure Notification mechanism triggers, it is prohibited from triggering again until a lockout period expires. The lockout period is configurable.

Image Sources

The format and content of image sources are pre-determined by WebLogic Server product development. Image sources stream their diagnostic image content out to an output stream. Each image source registers itself with the Image Manager. The image sources that contribute output to the diagnostic image include the following:

- WorkManager state
- Configuration state
- Defined deployment plans
- Plug-in configuration (if available)
- Log Cache (recently logged messages for all logs)
- State of all configured Harvestable instances and their attributes
- JNDI dump
- JRockit JRA output file
- JVM state (thread dump, memory information, etc.)
- Native info (like file descriptor counts)

Image Data

The Image Manager generates descriptive data in each diagnostic image. The data includes the following:

- Creation date and time of the image
- Source of the capture request
- Name of each Image Source included in the image and the time spent processing each of those Image Sources
- JVM and operating system information, if available
- Command-line arguments, if available
- Networking muxer version, if available
- WebLogic Server version, including patch and build number information

Diagnostic Image

A captured image results in a single file for the entire server which is uniquely named. Every diagnostic image produced has a unique name so multiple diagnostic images may exist, that is, existing diagnostic images are not overwritten by the creation of a new one.

Each diagnostic image is named as follows:

```
diagnostic_image_domain_server_yyyy_MM_dd_HH_mm_ss
```

If for some reason, and this is an extremely unlikely case (given that image creation should take more than one second), a new image defines a name that is already in use, it results in a log message indicating the collision and a diagnostic image is generated with the same name with an additional sequence number appended as follows:

```
diagnostic_image_domain_server_yyyy_MM_dd_HH_mm_ss_#
```

Where # represents the next available sequential number after all existing files with the same initial image name.

Note: There is no limit to the number of diagnostic images that can be written to disk.

Data Analyzers and Archiver

The Data Analyzers and Archiver component consists of two sub-components: Watches and Notifications and Archiver (see Figure 3-1):

This section covers the following topics:

Watches and Notifications

The Watches and Notifications component enables you to monitor metrics, instrumentation events, and log messages. You can write watch rules that specify relationships among multiple attributes of multiple MBeans and also configure notifications to fire once the specified rules are satisfied. Further, this component enables users to configure notifications to fire through disparate transportation mediums, such as SNMP, JMS, JMX, and SMTP.

A watch is the primary entity that captures a user's specific watch configuration. A watch encapsulates all of the configuration attributes, including the name of the watch, the watch-rule expression, the alarm settings, and the notifications listeners that are fired when a watch evaluates to true. Users can dynamically enable and disable watches and add or remove notification listeners for specific watches without shutting down and restarting the server.

You can also configure watches as alarms. If a watch evaluates to true and is configured as an alarm, then the watch is not evaluated again until the alarm is reset. You can configure an alarm to reset automatically (when a specified amount of time has expired) or manually using the Administration Console or the WebLogic Scripting Tool. Alarms are in-memory so they are also reset wherever the server reboots.

You can write watch rules that analyze data events, harvested metrics, and log records that are of the ServerLog type.

Note: You can configure watches to analyze ServerLog data, however, you cannot configure watches to analyze other types of log data, such as HTTP access log data, Domain log data, and so on.

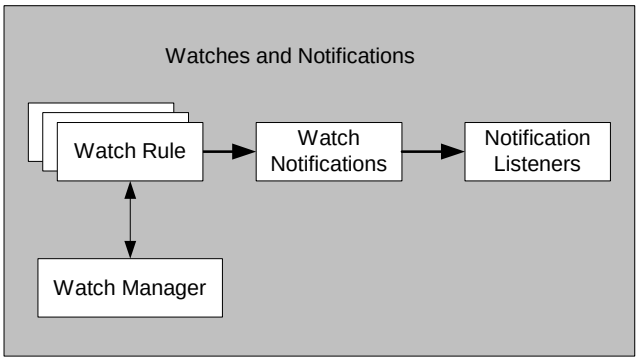
Watches can be composed of a number of watch rules and multiple notification listeners may be configured for each watch. By default, every watch logs an event in the server log.

Figure 3-7 illustrates the sub-components of Watches and Notifications.

This section covers the following topics:

- “Notification Listeners” on page 3-24

Figure 3-7 Watch and Notification Components



Watch Rule

A watch rule is a query expression that can be evaluated to true or false. A watch-rule expression encapsulates all information that is necessary for specifying a particular rule. While a watch-rule expression uses the same syntax as the Accessor component, it has different variables for different types of rule expressions. The watch-rule expression allows you to specify attributes, operators, and values. There are three types of watch-rule expressions:

- Expressions that refer to instrumentation events
- Expressions that refer to harvestable data metrics
- Expressions that refer to log events

You specify the type of watch-rule expression when you create the watch. A watch-rule expression may include any of the following:

- Variable name (Differs based upon the type of rule expression)
- Threshold value (`String, Integer, Long, Double, Boolean constant`)
- Operator (`<, >,<=, >=, =, !=, AND, OR, NOT, MATCHES, LIKE`)

Listing 3-1 shows an example of a watch-rule expression that analyzes harvested data.

Listing 3-1 Watch-rule Expression for the Harvested Data

```
${mydomain:Name=myserver,Type=ServerRuntime//SocketsOpenedTotalCount} >= 1
```

Watch Manager

The Watch Manager manages the Watch and Notification component. It manages watch configurations, evaluates watches, and manages watch alarms. Specifically, the Watch Manager performs the following tasks:

- Maintains a list of all watches and the state of each watch (enabled, disabled, alarm).
- When called by the Harvester after sampling is complete, evaluates watches that refer to harvester attributes and calls back into the harvester to access harvested data from the current sample.
- When called by the Watch Log Handler, evaluates watches that refer to log event attributes. The Watch Manager queues log events and evaluates them asynchronously.
- Maintains alarms and resets alarms either automatically or manually.

Watch Notifications

When a watch is triggered, watch notifications are passed to notification listeners that contain details about a watch, such as the watch name and watch type. Watches can also trigger a server image when they evaluate to true. Typically, watch notifications contain the following information:

- Watch name (the name of the watch that evaluated to true)
- Watch-rule type (either harvested, log record, or event)
- Watch-rule expression (the expression that evaluated to true)
- Time the watch evaluated to true (formatted, locale specific)
- Watch severity level (uses same levels as WebLogic Server log events)
- Watch alarm type (AutomaticReset or ManualReset)
- Watch alarm reset period (for AutomaticReset alarm types only)
- Watch domain name

- Watch server name

Watch notifications also contain payload information about the event that triggered the watch. This payload information is specific to the type of watch notification. Three types of watch notifications are supported:

- Log Watch Notification—The payload contains the attributes of the LogEntry.
- Instrumentation Watch Notification—The payload contains the data record items.
- Harvester Watch Notification—The payload contains the names and values of the harvested metrics that caused the watch rule to fire.

The payload information contains constants or variables depending on the type. For the log and instrumentation Watch Notifications, the payload information consists of constants and is contained in a set of key/value pairs in the `WatchNotification` implementation class. For the complete list of key/value pairs supported, see the `weblogic.diagnostics.watch.WatchNotification` class. For the Harvester Watch Notification, the payload consists of metric data and contains the MBean type, the name of the attribute, and the attribute's value.

Notification Listeners

Notification listeners provide an appropriate implementation for a particular transport medium. For example, SMTP notification listeners provide the mechanism to establish an SMTP connection with a mail server and trigger an e-mail with the notification instance that it receives. JMX, SNMP, JMS and other types of listeners provide their respective implementations as well.

Note: You can develop plug-ins that propagate events generated by the WebLogic Diagnostic Service using transport mediums other than SMTP, JMX, SNMP, or JMS. To do so, use the JMX NotificationListener Interface to implement an object and then propagate the notification according to the requirements of the selected transport medium.

Table 3-2 describes each notification listener type that is provided with WebLogic Server product and the relevant configuration settings for each type.

Table 3-2 Notification Listener Types

Notification Medium	Description	Configuration Parameter Requirements
JMS	Propagated via JMS Message queues or topics.	Required: Destination JNDI name. Optional: Connection factory JNDI name (use the default JMS connection factory if not present).
JMX	Propagated via standard JMX notifications.	None required. A runtime MBean is automatically constructed to broadcast the notifications. The name of this MBean is derived in a predictable way from the provided notification name.
SMTP	Propagated via regular e-mail.	Required: MailSession JNDI name and Destination e-mail. Optional: subject and body (if not specified, use default)
SNMP	Propagated via SNMP traps and the WebLogic Server SNMP Agent.	None required, but the SNMPTrapDestination MBean must be defined in the WebLogic SNMP agent.

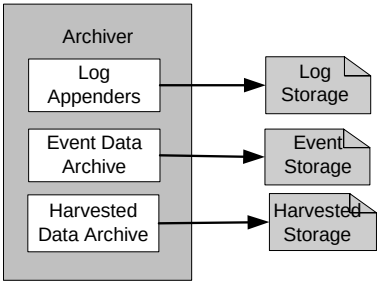
Note: By default, all notifications fired from watch rules are stored in the server log file in addition to being fired through the configured medium.

Archiver

The Archiver persists all data events, harvested data, and log records collected by the Data Collectors to permanent storage and makes the archived data available for historical review. An archive is instantiated on a per-server basis and logically consists of the following components: Log Appenders, an Event Data Archive, and a Harvested Data Archive (see Figure 3-8). You can configure the Archiver to archive data to a file or to a JDBC database.

Note: The WebLogic Diagnostic Service only supports persistence of data events and harvested data to JDBC databases. It does not support persistence of log records to JDBC databases.

Figure 3-8 Archiver Components

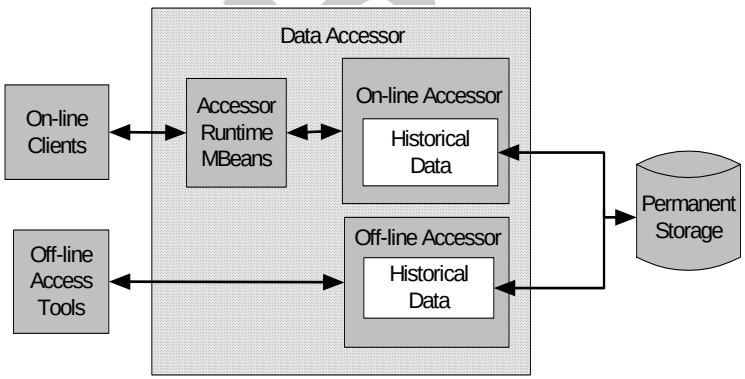


Each Archiver component serves a specific function. Traditional logging information, which is included in the server log for use by operators and administrators, is persisted through the standard log appenders. New event data that is intended for system consumption, and that may be compacted and needs not be readable by users, is persisted in an event store by the Event Data Archive. Harvested data is persisted into a data store using a Harvested Data Archive.

Data Accessor

The Data Accessor component enables users to access to all the data collected by the Data Collectors, including log records, data events, and harvested metrics, as well as information about the data, such as log files (see Figure 3-9).

Figure 3-9 Data Accessor Components



The Data Accessor component only provides access to diagnostic information for the server on which it is running. You can use the Data Accessor to do data lookups by type, by component,

and by attribute. You also use it do time-based filtering and, when accessing events, filtering by severity, source, and content. Since the WebLogic Diagnostic Service maintains different kinds of diagnostic information in separate physical data stores that can be modeled as tabular data, you can also access diagnostic data in tabular form.

The Data Accessor provides a single access point to clients and supplies data to such clients on demand. Therefore, you can use clients to access diagnostic data in a production environment on an as needed basis.

The Data Accessor provides both on-line and off-line access to diagnostic data.

On-line mode: When the server is running, historical data, such as archived metrics, log records, and data events files, is accessible through the On-line Accessor. The On-line Accessor allows you to use of the WebLogic Server Administration Console, a scripting utility such as the WebLogic Scripting Tool, or an external third-party diagnostic tool to access diagnostic data. The data is exposed to on-line clients by two Accessor runtime MBeans: the WLDFAccessRuntimeMBean and the WLDFDataAccessRuntimeMBean.

- The WLDFAccessRuntimeMBean is used to get the logical names of the available data stores on the server and to look up a WLDFDataAccessRuntime MBean to access the data from a specific log based on its logical name. The different data stores are uniquely identified by their logical names.
- The WLDFDataAccessRuntime MBean is used to retrieve data stores based on a search condition, or query. Optionally, a time interval can be specified with the query so as to retrieve data records within a specific time duration only. This MBean provides meta-data about the columns of the data set and the earliest and latest timestamp of the records in the underlying data store.

Off-line mode: When a server shuts down, historical data, such as archived metrics, log records, and data events files, is accessible through the Off-line Accessor. The Off-line Accessor supports the use of file system access tools to access and analyze diagnostic data. The only limitation to off-line access is that the data can be accessed only from the machine on which the server was running.

Note: You can use the `exportDiagnosticData` command provided by the WebLogic Scripting Tool (`weblogic.WLST`) to access historical data in the off-line mode. For more information on the `exportDiagnosticData` command, see Diagnostic Command in *WebLogic Scripting Tool*.

This section covers the following topics:

- "Data Accessor Query Language" on page 3-28
- "Data Accessor Code Examples" on page 3-32

Access to Data Stores

The Data Accessor provides information regarding available diagnostic information from other WebLogic Server Diagnostic Service components. Captured information is segregated into logical data stores that are separated by the types of diagnostic data. For example, Server logs, HTTP logs, and harvested metrics are captured in separate data stores. The Accessor component enables users to discover such data stores, determine the nature of the data they contain, and access their data selectively using a query.

Data stores are tabular in format. Each record in the table represents a datum. Columns in the table describe various characteristics of the datum. Different data stores may have different columns. However, most data stores have some shared columns, such as the time when the data item was collected.

The information about the data includes:

- A list of supported data store types, for example: `SERVER_LOG`, `HTTP_LOG`, and `HARVESTED_DATA`
- A list of available data stores
- The layout of each data store (information that describes the columns in the data store)

Data Accessor Query Language

The Data Accessor component enables users to access data from different data stores. To query data from these data stores, the Data Accessor provides a query language. The query is specified as a string in the query language defined by the accessor. Although different data stores may have different layouts in terms of the attributes (columns), they are all modeled as tabular data. All records from the data store that match the specified query are returned. By specifying appropriate constraints in the query, a client can filter the data in different ways and thus control the content and volume of the information returned.

The following topics provide additional information:

- "Query Language Syntax" on page 3-29
- "Logical Names" on page 3-31
- "Log Types" on page 3-31

Query Language Syntax

The query language syntax is a small and simplified subset of SQL syntax. It is used to specify the selection criteria to build boolean expressions based on the column names supported by the data store. Using the query language, you can specify relatively complex selection expressions, which typically appear in the `WHERE` clause of an `SQL SELECT` statement.

A query represents the search conditions that the records in the result set should meet. You can use the column names listed in Table 3-4 in a query to retrieve data. The column names included in a query are resolved for each row of data and added to the result set only if it satisfies the query conditions. An empty query, that is, a query that omits column names, returns all the entries in the log.

The query language supports Boolean operators: `AND`, `OR`, and `NOT` and the relational operators listed in Table 3-3.

Table 3-3 Relational Operators Supported by the Query Language

Relational Operator	Definition
<	Less than
>	Greater than
<=	Less than or equals
>=	Greater than or equals
=	Equals
!=	Not equals

Note: The relational operators do a lexical comparison for Strings. For more informaiton, see the documentation for the `java.lang.String.compareTo(String str)` method. Numeric constants are specified the same as in Java. Some examples of numeric constants are 2, 2.0, 12.856f, 2.1934E-4, 123456L and 2.0D. String constants are single quoted, for example '`Cat`', '`Dog`', and so on.

Listing 3-2 shows operator precedence in complex expressions.

Listing 3-2 Operator Procedure (Descending Order)

```
( )
NOT
=, <>, >, >=, <, <=, like
AND
OR
```

The query language also supports special operators for string matching: `LIKE and MATCHES`.

- The `LIKE` operator works like SQL `Like`, and the percent (`%`) and period (`.`) characters are used as wildcards.
- The `MATCHES` operator enables a regular expression to be matched against a `String` value.

An example of a query is:

```
SUBSYSTEM = 'Deployer' AND MESSAGE LIKE '%Failed%'
```

You can also nest queries by surrounding sub-expressions within parentheses, for example:

```
A=2 AND (C=3 OR D='Daddy')
```

You can build complex query expressions using sub-expressions involving variables, binary comparisons and other complex sub-expressions.

The query syntax also includes the `IN` operator for matching if the variable value exists in a predefined set. For example:

```
SUBSYSTEM IN ('A','B')
```

The query is executed against a specific data store. Thus, the `FROM` part of the query is implicit. When there is a match, all columns of matching rows are returned. Listing 3-3 shows an example of an API method invocation that includes a query to capture harvested attributes of the JDBC connection pool `MyPool`, within an interval between a `start_time` (inclusive) and an `end_time` (exclusive).

Listing 3-3 Query Example

```
WLDFDataAccessRuntimeMBean.retrieveDataRecords(timestampFrom, timeStampTo,
"TYPE='JDBCConnectionPoolRuntime'  AND NAME='MyPool'")
```

Also, you can enclose a variable name within `${}` if it includes special characters as in a MBean Object Name. For example:

```
${mydomain:Name=myserver,
      Type=ServerRuntime//SocketsOpenedTotalCount} >= 1
```

Note: Notice that the Object Name and the Attribute Name are separated by '//' in the Watch variable name.

The column names included in the query are resolved for each row of data and added to the result set only if it satisfies the query conditions. An empty query returns all the entries in the log.

Logical Names

The logical name for a data store must be unique. The logical name consists of the log type and, optionally, other identifiers separated by a forward-slash character (`/`). For example, the logical name of the server log data store is simply ServerLog, however, other log types, such HTTP access logs, may require more information. For HTTP access logs, there can be one log available for each of the virtual hosts deployed on the server. For the default WebServer, the logical name is `HTTPAccessLog`, whereas for virtual hosts, the logical name is `HTTPAccessLog/`*`virtualHostName`*.

Log Types

The log types supported by the Data Accessor are described in Table 3-4.

Table 3-4 Data Accessor Log Types

Log Type	Column Names
`ServerLog`	RECORDID, DATE, SEVERITY, SUBSYSTEM, MACHINE, SERVER, THREAD, USERID, TXID, CONTEXTID, TIMESTAMP, MSGID, MESSAGE.
`DomainLog`	Same as ServerLog
`JDBCLog`	Same as ServerLog
`WebAppLog`	Same as ServerLog
`HTTPAccessLog`	RECORDID, HOST, REMOTEUSER, AUTHUSER, TIMESTAMP, REQUEST, STATUS, BYTECOUNT
`HarvestedData Archive`	RECORDID, TIMESTAMP, DOMAIN, SERVER, TYPE, NAME, ATTRNAME, ATTRTYPE, ATTRVALUE

Table 3-4 Data Accessor Log Types

Log Type	Column Names
`EventsDataArchive`	RECORDID,TIMESTAMP, CONTEXTID, USERID, TXID, TYPE, DOMAIN, SERVER, SCOPE, MONITOR, FILENAME, LINENUM, CLASSNAME, METHODNAME, METHODDSC, ARGUMENTS, RETVAL, PAYLOAD, CTXPAYLOAD
`ConnectorLog`	RECORDID, LINE
`JMSMessageLog`	RECORDID, DATE, TXID, CONTEXTID, TIMESTAMP, NANOTIMESTAMP, JMSMESSAGEID, JMSCORRELATIONID, DESTINATION, EVENT, USERID, MESSAGECONSUMER, MESSAGE

Note: In the case of HTTP Access logs with extended format, the number of columns are user-defined.

Data Accessor Code Examples

For Data Accessor code examples, see "Accessing Data Online and Offline" on page E-1.

Terminology

Key terms that you will encounter throughout the diagnostic and monitoring documentation include the following:

artifact

Any resulting physical entity, or data, generated and persisted to disk by the WebLogic Diagnostic Service that can be used later for diagnostic analysis. For example, the diagnostic image file that is created when the server fails is an artifact. The diagnostic image artifact is provided to support personnel for analysis to determine why the server failed. The WebLogic Diagnostic Service produces a number of different artifacts.

context creation

If diagnostic monitoring is enabled, a diagnostic context is created, initialized, and populated by WebLogic Server when a request enters the system. Upon request entry, WebLogic Server determines whether a diagnostic context is included in the request. If so, the request is propagated with the provided context. If not, WebLogic Server creates a new context with a specific name (`weblogic.management.DiagnosticContext`). The contextual data for the diagnostic context is stored in the diagnostic context payload. Thus, within the scope of a request execution, existence of the diagnostic context is guaranteed.

context payload

The actual contextual data for the diagnostic context is stored in the Context Payload. See also context creation, diagnostic context, request dyeing.

data stores

Data stores are a collection of data, or records, represented in a tabular format. Each record in the table represents a datum. Columns in the table describe various characteristics of the datum. Different data stores may have different columns; however, most data stores have some shared columns, such as the time when the data item was collected.

In WebLogic Server, information captured by WebLogic Diagnostic Service is segregated into logical data stores, separated by the types of diagnostic data. For example, Server logs, HTTP logs, and harvested metrics are captured in separate data stores.

diagnostic action

Business logic or diagnostic code that is executed when a joinpoint defined by a pointcut is reached. Diagnostic actions, which are associated with specific pointcuts, specify the code to execute at a joinpoint. Put another way, a pointcut declares the location and a diagnostic action declares what is to be done at the locations identified by the pointcut. Diagnostic actions provide visibility into a running server and applications. Diagnostic actions specify the diagnostic activity that is to take place at locations, or pointcuts, defined by the monitor in which it is implemented. Without a defined action, a diagnostic monitor is useless.

Depending on the functionality of a diagnostic action, it may need a certain environment to do its job. Such an environment must be provided by the monitor to which the diagnostic action is attached; therefore, diagnostic actions can be used only with compatible monitors. Hence, diagnostic actions are classified by type so that their compatibility with monitors can be determined.

To facilitate the implementation of useful diagnostic monitors, a library of suitable diagnostic actions is provided with the WebLogic Server product.

diagnostic context

The WebLogic Diagnostic Service adds contextual information to all requests when they enter the system. You can use this contextual information, referred to as the diagnostic context, to reconstruct transactional events, as well correlate events based on the timing of the occurrence or logical relationships. Using diagnostic context you can reconstruct or piece together a thread of execution from request to response.

Various diagnostic components, for example, the logging services and diagnostic monitors, use the diagnostic context to tag generated data events. Using the tags, the diagnostic data can be collated, filtered and correlated by the WebLogic Diagnostic Service and third-party tools.

The diagnostic context also makes it possible to generate diagnostic information only when contextual information in the diagnostic context satisfies certain criteria. This

capability enables you to keep the volume of generated information to manageable levels and keep the overhead of generating such information relatively low. See also context creation, context payload, request dyeing.

diagnostic image

An artifact containing key state from an instance of a server that is meant to serve as a server-level state dump for the purposes of diagnosing significant failures. This artifact can be used to diagnose and analyze problems even after the server has cycled. Each diagnostic image contains a summary artifact that includes, at a minimum, the following elements:

- Creation date and time of the image
- Source of the capture request
- Name of each image source included in the image and the time spent processing each of those image sources
- Java Virtual Machine (JVM) and operating system information if available
- Command-line arguments if available
- Networking muxer version if available
- WebLogic Server version including patch and build number information

diagnostic module

A diagnostic module is the definition the configuration settings that are to applied to the WebLogic Diagnostic Service. The configuration settings determine what data is to be collected and processed, how the data is to be analyzed and archived, what notifications and alarms are to be fired, and the operating parameters of the server image capture component. Once a diagnostic module has been defined, or configured, it can be distributed to a running server where the data is collected.

Typically, diagnostic data is collected from a number of sources depending on the complexity of the system being monitored. Rather than collect data from all sources all the time—an approach that could result in massive amounts of data being collected and require significant system resources to collect and many person hours to evaluate—the ability to define a subset of sources to be monitored saves system and human resources. Further, the ability to define diagnostic modules enables the users to design the data collected to meet the needs of both the system and the environment in which it is being used. Clearly, the ability to define diagnostic modules can have a very positive effect on maintainability and productivity.

In WebLogic Server, a single diagnostic module can be defined and then targeted to one or more individual servers or clusters of servers. Only one diagnostic module can be targeted to a single server at a time.

diagnostic monitor

A diagnostic monitor is a unit of diagnostic code that defines 1) the locations in a program where the diagnostic code will be added and 2) the diagnostic actions that will be executed at those locations.

WebLogic Server provides a library of useful diagnostic monitors. Users can integrate these monitors into server and application classes. Once integrated, the monitors take effect at server startup for server classes and application deployment and redeployment for application classes.

diagnostic notification

The action that occurs as a result of the successful evaluation of a watch rule. The WebLogic Diagnostic Service supports four types of diagnostic notifications: Java Management Extensions (JMX), Java Message Service (JMS), Simple Mail Transfer Protocol (SMTP), and Simple Network Management Protocol (SNMP).

dye filtering

The process of looking at the dye mask and making the decision as to whether or not a diagnostic monitor should execute an action so as to generate a data event. Dye filtering is dependent upon dye masks. You must define dye masks in order for dye filtering to take place. See also dye mask, request dyeing.

dye mask

The entity that contains a predefined set of conditions that are used by dye filtering to determine whether or not a data event should be generated. You use system resource descriptors to configure the dye masks. See also dye filtering, request dyeing.

harvestable entities

A harvestable entity is any entity that is available for data consumption via the Harvester. Once an entity is identified as a harvestable resource, the Harvester can engage the entity in the data collection process.

Harvestable entities provide access to the following information: harvestable attributes, values of harvestable attributes, meta-data for harvestable attributes, and the name of the harvestable entity. See also harvestable data, harvested data, Harvester's configuration data set, MBean type discovery.

harvestable data

Harvestable data (types, instances, attributes) is the set of data that potentially could be harvested when and if it is configured for harvesting. Therefore, the set of harvestable data exists independent of what data is configured for harvesting and of what data samples are taken.

The `WLDFHarvesterRuntimeMBean` provides the set of harvestable data for users. For a description of the information about harvestable data provided by this MBean, see the description of the `weblogic.management.runtime.WLDFHarvesterRuntimeMBean` in the WebLogic Server MBean Reference.

The WebLogic Diagnostic Service only makes MBeans available as harvestable. In order for an MBean to be harvestable, it must be registered in the local WebLogic Server runtime MBean server. See also harvestable entities, harvested data, Harvester's configuration data set, MBean type discovery.

harvested data

A type, instance, or attribute is called harvested data if that data is currently being harvested. To meet these criteria the data must: 1) be configured to be harvested, 2) if applicable, it must have been discovered, and 3) it must not throw exceptions while being harvested.

When the configuration is loaded, the set of harvested items is the same as the set of configured items for WebLogic Server MBean types. Configured customer data is added as it is discovered. So the list of harvested instances for both WebLogic Server and customer types can grow as MBeans are added. And for customer data, the introduction of new instances can also cause the list of harvested types and attributes to grow. However, if the Harvester discovers that certain items cannot be harvested, they are removed, thereby, causing the list to shrink. If the Harvester configuration changes, the set of harvested data is recalculated.

The `WLDFHarvesterRuntimeMBean` provides the set of harvested data for users. The information returned by this MBean should be considered a snapshot of a potentially changing state. For a description of the information about harvested data provided by the this MBean, see the description of the `weblogic.management.runtime.WLDFHarvesterRuntimeMBean` in WebLogic Server MBean Reference. See also harvestable entities, harvestable data, Harvester's configuration data set, MBean type discovery.

Harvester's configuration data set

The set of data to be harvested as defined by the Harvester's configuration. The configured data set can contain items that are not harvestable and items that are not currently being harvested.

The set of harvestable MBeans comprises the set of harvestable instances. This set is dynamic in that it grows and shrinks as MBeans are registered with and removed from the MBean server. Since the set is dynamic, some data may be legitimately harvestable one moment and not harvestable the next. The dynamics of the set can also create situations where the configuration is legitimate, but the data is not available to verify the

configuration. The Harvester's validation is designed to be tolerant of these dynamic situations.

The Administration Console assists you in the configuration process by prompting you with lists of harvestable data that can be configured—such as harvestable types, instances of harvestable types, and harvestable attributes of harvestable types. The information listed is always complete for WebLogic Server MBeans data, but is discovered dynamically for custom MBeans. See also harvestable entities, harvestable data, harvested data, Harvester's configuration data set.

joinpoint

A well defined point in the program flow where diagnostic code can be added. The Instrumentation component allows identification of such diagnostic joinpoints with an expression in a generic manner.

pointcut

A well defined set of joinpoints, typically identified by some generic expression. Pointcuts identify joinpoints, which are well-defined points in the flow of execution, such as a method call or data variable access. The Instrumentation component provides a mechanism to allow execution of specific diagnostic code at such pointcuts. The Instrumentation component adds such diagnostic code to the server and application code.

MBean (managed bean)

A Java object that provides a management interface for an underlying resource. An MBean is part of Java Management Extensions (JMX).

In the WebLogic Diagnostic Service, MBean classes are used to configure the service and to monitor its runtime state. MBeans are registered with the MBean server that runs inside WebLogic Server. MBeans are implemented as standard MBeans which means that each class implements its own MBean interface.

MBean type discovery

For WebLogic Server entities, the set of harvestable types is known at system startup, but not the complete set of harvestable instances. For customer defined MBeans, however, the set of types can grow dynamically, as more MBeans appear at runtime. The process of detecting a new type based on the registration of a new MBean is called type discovery. MBean type discovery is only applicable to customer MBeans.

MBean type meta-data

The set of harvestable attributes for a type (and its instances) is defined by the meta-data for the type. Since the WebLogic Server model is MBeans, the meta-data is provided through MBeanInfos. Since WebLogic type information is always available, the set of

harvestable attributes for WebLogic Server types (and existing and potential instances) is always available as well. However, for customer types, knowledge of the set of harvestable attributes is dependent on the existence of the type. And, the type does not exist until at least one instance is created. So the list of harvestable attributes on a user defined type is not known until at least one instance of the type is registered.

It is important to be aware of latencies in the availability of information for custom MBeans. Due to latencies, the Administration Console cannot provide complete lists of all harvestable data in its user selection lists for configuring the harvester. The set of harvestable data for WebLogic Server entities is always complete, but the set of harvestable data for customer entities (and even the set of entities itself) may not be complete.

meta-data

Meta-data is information that describes the information the WebLogic Diagnostic Service collects. Because the service collects diagnostic information from different sources, the consumers of this information need to know what diagnostic information is collected and available. To satisfy this need, the Data Accessor provides functionality to programmatically obtain this meta-data. The meta-data made available by means of the Data Accessor includes: 1) a list of supported data store types, for example, `SERVER_LOG`, `HTTP_LOG`, `HARVESTED_DATA`, 2) a list of available data stores, and 3) the layout of each data store, that is, information about columns in the data store.

metrics

Monitoring system operation and diagnosing problems depends on having data from running systems. Metrics are measurements of system performance. From these measurements, support personnel can determine whether the system is in good working order or a problem is developing.

In general, metrics are exposed to the WebLogic Diagnostic Service as attributes on qualified MBeans. In WebLogic Server, metrics include performance measurements for the operating system, the virtual machine, the system runtime, and applications running on the server.

request dyeing

Requests can be dyed, or specially marked, to indicate that they are of special interest. For example, in a running system, it may be desirable to send a specially marked test request, which can be conditionally traced by the tracing monitors. This allows creation of highly focused diagnostic information without slowing down other requests.

Requests are typically marked when they enter the system by setting flags in the diagnostic context. The diagnostic context provides a number of flags, 64 in all, that can be independently set or reset.

Note: Only dyes 56-63 are available for use by applications. All other dye flags are reserved for use by WebLogic Diagnostic Service components and libraries.

The DyeInjectionMonitor can turn on these flags when the request enters the system based on its configuration and request properties. Thereafter, other diagnostic monitors can make use of these flags (dyes) to conditionally execute certain actions. For example, a diagnostic monitor can be configured to perform its diagnostic action only if the request originated from a specific address. See also context creation, context payload, diagnostic context.

system image capture

Whenever a system fails, there is need to know its state when it failed. Therefore, a means of capturing system state upon failure is critical to failure diagnosis. A system image capture does just that. It creates, in essence, a diagnostic snapshot, or dump, from the system for the express purpose of diagnosing significant failures.

In WebLogic Server, you can configure the WebLogic Diagnostic Service provides the First-Failure Notification feature to trigger system image captures automatically when the server experiences an abnormal shutdown. You can also implement watches to automatically trigger server image captures when significant failures occur and you can manually initiate server image captures on demand.

watch

A watch encapsulates all of the information for a watch rule. This includes the watch rule expression, the alarm settings for the watch, and the various notification handlers that will be fired once a watch rule expression evaluates to true.

weaving time

The time it takes while loading server and application classes to insert the diagnostic byte codes into server and application classes at well-defined locations. The diagnostic byte codes enable the WebLogic Diagnostic Service to take diagnostic actions. Weaving time affects both the load time for server-level instrumented classes and application deployment time for application-level classes.

APPENDIX A

WebLogic Diagnostic Service Libraries

The WebLogic Diagnostic Service libraries that ship with the BEA WebLogic Server product are discussed in the following sections:

Diagnostic Monitor Library

This section lists and describes the diagnostic monitors defined by the Diagnostic Monitor Library that is provided with the WebLogic Server product. The diagnostic monitors are broadly classified as server-scoped and application-scoped monitors. The former can be used to instrument WebLogic Server classes. The later can be used to instrument application classes. Except for the DyeInjection monitor, all monitors are delegating monitors, that is, they do not have a built-in diagnostic action. Instead, they delegate to actions attached to them to perform diagnostic activity.

All of the monitors are preconfigured with their respective pointcuts. However, the actual locations affected by them may vary depending on the classes they instrument. For example, the Servlet_Before_Service monitor adds diagnostic code at the entry of servlet or java server page (jsp) service methods at different locations in different servlet implementations.

For any delegating monitor, only compatible actions may be attached. The compatibility is determined by the nature of the monitor.

Table A-1 lists and describes the diagnostic monitors that can be used within server scope, that is, in WebLogic Server classes. For the diagnostic actions that are compatible with each monitor, see the Compatible Action Type column in the table.

Table A-2 lists the diagnostic monitors provided with the WebLogic Server product that can be used within application scopes, that is, in deployed applications. For the diagnostic actions that are compatible with each monitor, see the Compatible Action Type column in the table.

Table A-1 Diagnostic Monitors for Use Within Server Scope

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
DyeInjection	Before	Built-in	At points where requests enter the server.
Connector_Before_ Inbound	Before	StatelessAction	At entry of methods handling inbound connections.
Connector_Before_ Outbound	Before	StatelessAction	At entry of methods handling outbound connections.
Connector_Before_Work	Before	StatelessAction	At entry of methods related to scheduling, starting and executing work items.
Connector_Before_Tx	Before	StatelessAction	Entry of transaction register, unregister, start, rollback and commit methods.
Connector_After_ Inbound	Server	StatelessAction	At exit of methods handling inbound connections.
Connector_After_ Outbound	After	StatelessAction	At exit of methods handling outbound connections.
Connector_After_Work	After	StatelessAction	At exit of methods related to scheduling, starting and executing work items.
Connector_After_Tx	After	StatelessAction	At exit of transaction register, unregister, start, rollback and commit methods.
Connector_Around_ Inbound	Around	AroundAction	At entry and exit of methods handling inbound connections.

Table A-1 Diagnostic Monitors for Use Within Server Scope (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
Connector_Around_ Outbound	Around	AroundAction	At entry and exit of methods handling outbound connections.
Connector_Around_ Work	Around	AroundAction	At entry and exit of methods related to scheduling, starting and executing work items.
Connector_Around_Tx	Around	AroundAction	At entry and exit of transaction register, unregister, start, rollback and commit methods.

Table A-2 Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
Servlet_Before_Service	Before	StatelessAction	At method entries of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter
Servlet_Before_Session	Before	StatelessAction	Before calls to servlet methods: HttpServletRequest.getSession HttpSession.setAttribute/putValue HttpSession.getAttribute/getValue HttpSession.removeAttribute/ removeValue HttpSession.invalidate
Servlet_Before_Tags	Before	StatelessAction	Before calls to jsp methods: Tag.doStartTag Tag.doEndTag

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JNDI_Before_Lookup	Before	StatelessAction	Before calls to javax.naming.Context lookup methods Context.lookup*
JMS_Before_ TopicPublished	Before	StatelessAction	Before call to methods: TopicPublisher.publish
JMS_Before_ MessageSent	Before	StatelessAction	Before call to methods: QueSender send
JMS_Before_ AsyncMessageReceived	Before	StatelessAction	At entry of methods: MessageListener.onMessage
JMS_Before_ SyncMessageReceived	Before	StatelessAction	Before calls to methods: MessageConsumer.receive*
JDBC_Before_ GetConnection	Before	StatelessAction	Before calls to methods: Driver.connect DataSource.getConnection
JDBC_Before_ CloseConnection	Before	StatelessAction	Before calls to methods: Connection.close
JDBC_Before_ CommitRollback	Before	StatelessAction	J Before calls to methods: Connection.commit Connection.rollback
JDBC_Before_Statement	Before	StatelessAction	Before calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_Before_Execute	Before	StatelessAction	Before calls to methods: Statement.execute* PreparedStatement.execute*

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_Before_ SessionEjbMethods	Before	StatelessAction	At entry of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_Before_SessionEjb SemanticMethods	Before	StatelessAction	At entry of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_Before_SessionEjb BusinessMethods	Before	StatelessAction	At entry of all SessionBean methods, which are not standard ejb methods.
EJB_Before_ EntityEjbMethods	Before	StatelessAction	At entry of methods: EnitityBean.setEntityContext EnitityBean.unsetEntityContext EnitityBean.ejbRemove EnitityBean.ejbActivate EnitityBean.ejbPassivate EnitityBean.ejbLoad EnitityBean.ejbStore
EJB_Before_Entity EjbSemanticMethods	Before	StatelessAction	At entry of methods: EnitityBean.set* EnitityBean.get* EnitityBean.ejbFind* EnitityBean.ejbHome* EnitityBean.ejbSelect* EnitityBean.ejbCreate* EnitityBean.ejbPostCreate*
EJB_Before_EntityEjb BusinessMethods	Before	StatelessAction	At entry of all EnitityBean methods, which are not standard ejb methods.
MDB_Before_Message Received	Before	StatelessAction	At entry of methods: MessageDrivenBean.onMessage

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
MDB_Before_Set MessageDrivenContext	Before	StatelessAction	At entry of methods: MessageDrivenBean.setMessage DrivenContext
MDB_Before_Remove	Before	StatelessAction	At entry of methods: MessageDrivenBean.ejbRemove
JTA_Before_Start	Before	StatelessAction	At entry of methods: UserTransaction.begin
JTA_Before_Commit	Before	StatelessAction	At entry of methods: UserTransaction.commit
JTA_Before_Rollback	Before	StatelessAction	At entry of methods: UserTransaction.rollback
Servlet_After_Service	After	StatelessAction	At method exits of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter
Servlet_After_Session	After	StatelessAction	After calls to servlet methods: HttpServletRequest.getSession HttpSession.setAttribute/putValue HttpSession.getAttribute/getValue HttpSession.removeAttribute/ removeValue HttpSession.invalidate
Servlet_After_Tags	After	StatelessAction	After calls to jsp methods: Tag.doStartTag Tag.doEndTag

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JNDI_After_Lookup	After	StatelessAction	After calls to javax.naming.Context lookup methods: Context.lookup*
JMS_After_Topic Published	After	StatelessAction	After call to methods: TopicPublisher.publish
JMS_After_MessageSent	After	StatelessAction	After call to methods: QueSender send
JMS_After_ AsyncMessageReceived	After	StatelessAction	At exits of methods: MessageListener.onMessage
JMS_After_Sync MessageReceived	After	StatelessAction	After calls to methods: MessageConsumer.receive*
JDBC_After_Get Connection	After	StatelessAction	After calls to methods: Driver.connect DataSource.getConnection
JDBC_After_ CloseConnection	After	StatelessAction	After calls to methods: Connection.close
JDBC_After_Commit Rollback	After	StatelessAction	After calls to methods: Connection.commit Connection.rollback
JDBC_After_Statement	After	StatelessAction	After calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_After_Execute	After	StatelessAction	After calls to methods: Statement.execute* PreparedStatement.execute*

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_After_ SessionEjbMethods	After	StatelessAction	At exits of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_After_SessionEjb SemanticMethods	After	StatelessAction	At exits of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_After_SessionEjb BusinessMethods	After	StatelessAction	At exits of all SessionBean methods, which are not standard ejb methods.
EJB_After_EntityEjb Methods	After	StatelessAction	At exits of methods: EnitityBean.setEntityContext EnitityBean.unsetEntityContext EnitityBean.ejbRemove EnitityBean.ejbActivate EnitityBean.ejbPassivate EnitityBean.ejbLoad EnitityBean.ejbStore
EJB_After_EntityEjb SemanticMethods	After	StatelessAction	At exits of methods: EnitityBean.set* EnitityBean.get* EnitityBean.ejbFind* EnitityBean.ejbHome* EnitityBean.ejbSelect* EnitityBean.ejbCreate* EnitityBean.ejbPostCreate*
EJB_After_EntityEjb BusinessMethods	After	StatelessAction	At exits of all EnitityBean methods, which are not standard ejb methods.
MDB_After_Message Received	After	StatelessAction	At exits of methods: MessageDrivenBean.onMessage

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
MDB_After_SetMessage DrivenContext	After	StatelessAction	At exits of methods: MessageDrivenBean.setMessage DrivenContext
MDB_After_Remove	After	StatelessAction	At exits of methods: MessageDrivenBean.ejbRemove
JTA_After_Start	After	StatelessAdvice	At exits of methods: UserTransaction.begin
JTA_After_Commit	After	StatelessAdvice	At exits of methods: UserTransaction.commit
JTA_After_Rollback	After	StatelessAdvice	At exits of methods: UserTransaction.rollback
Servlet_Around_Service	Around	AroundAction	At method entry and exits of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter
Servlet_Around_Session	Around	AroundAction	Before and after calls to servlet methods: HttpServletRequest.getSession HttpSession.setAttribute/putValue HttpSession.getAttribute/getValue HttpSession.removeAttribute/ removeValue HttpSession.invalidate
Servlet_Around_Tags	Around	AroundAction	Before and after calls to jsp methods: Tag.doStartTag Tag.doEndTag

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JNDI_Around_Lookup	Around	AroundAction	Before and after calls to javax.naming.Context lookup methods Context.lookup*
JMS_Around_Topic Published	Around	AroundAction	Before and after call to methods: TopicPublisher.publish
JMS_Around_Message Sent	Around	AroundAction	Before and after call to methods: QueSender send
JMS_Around_Async MessageReceived	Around	AroundAction	At entry and exits of methods: MessageListener.onMessage
JMS_Around_Sync MessageReceived	Around	AroundAction	Before and after calls to methods: MessageConsumer.receive*
JDBC_Around_Get Connection	Around	AroundAction	Before and after calls to methods: Driver.connect DataSource.getConnection
JDBC_Around_Close Connection	Around	AroundAction	Before and after calls to methods: Connection.close
JDBC_Around_Commit Rollback	Around	AroundAction	Before and after calls to methods: Connection.commit Connection.rollback
JDBC_Around_ Statement	Around	AroundAction	Before and after calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_Around_Execute	Around	AroundAction	Before and after calls to methods: Statement.execute* PreparedStatement.execute*

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_Around_Session EjbMethods	Around	AroundAction	Before and after calls to methods: Statement.execute* PreparedStatement.execute*
EJB_Around_SessionEjb SemanticMethods	Around	AroundAction	At entry and exits of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_Around_SessionEjb BusinessMethods	Around	AroundAction	At entry and exits of all SessionBean methods, which are not standard ejb methods.
EJB_Around_EntityEjb Methods	Around	AroundAction	At exits of methods: EnitityBean.setEntityContext EnitityBean.unsetEntityContext EnitityBean.ejbRemove EnitityBean.ejbActivate EnitityBean.ejbPassivate EnitityBean.ejbLoad EnitityBean.ejbStore
EJB_Around_EntityEjb SemanticMethods	Around	AroundAction	At entry and exits of methods: EnitityBean.set* EnitityBean.get* EnitityBean.ejbFind* EnitityBean.ejbHome* EnitityBean.ejbSelect* EnitityBean.ejbCreate* EnitityBean.ejbPostCreate*
EJB_Around_EntityEjb BusinessMethods	Around	AroundAction	At entry and exits of all EnitityBean methods that are not standard ejb methods.
MDB_Around_Message Received	Around	AroundAction	At entry and exits of methods: MessageDrivenBean.onMessage

Table A-2 Diagnostic Monitors for Use Within Application Scopes (Continued)

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
MDB_Around_Set MessageDrivenContext	Around	AroundAction	At entry and exits of methods: MessageDrivenBean.setMessage DrivenContext
MDB_Around_Remove	Around	AroundAction	At entry and exits of methods: MessageDrivenBean.ejbRemove
JTA_Around_Start	Around	AroundAction	At entry and exits of methods: UserTransaction.begin
JTA_Around_Commit	Around	AroundAction	At entry and exits of methods: UserTransaction.commit
JTA_Around_Rollback	Around	AroundAction	At entry and exits of methods: UserTransaction.rollback

Diagnostic Action Library

Table A-3 lists and describes the actions provided in the Diagnostic Action Library. These diagnostic actions can be used with the delegating monitors described in Table A-1 and Table A-2. They can also be used with custom monitors that you can define and use within applications. Each diagnostic action can only be used with monitors with which they are compatible, as indicated by the Compatible Monitor Type column.

Table A-3 Diagnostic Action Library

Name	Type	Compatible Monitor Types	Description
TraceAction	Stateless Action	Before, After	Generates a trace event at affected location in the program execution.
DisplayArguments Action	Stateless Action	Before, After	Generates an instrumentation event at affected location in the program execution to capture method arguments or return value.

Table A-3 Diagnostic Action Library (Continued)

Name	Type	Compatible Monitor Types	Description
TraceElapsedTime Action	`Around Action`	`Around`	Generates an instrumentation event at affected location in the program execution to capture elapsed times.
StackDumpAction	`Stateless Action`	Before, After	Generates an instrumentation event at affected location in the program execution to capture stack dump.
ThreadDumpAction	StatelessAction	Before, After	Generates an instrumentation event at affected location in the program execution to capture thread dump, if the underlying VM supports it.

Configuring the DyeInjection Monitor and Dye Filtering

This section covers the following topics:

Using the DyeInjection Monitor

The DyeInjection monitor inspects request properties as the request enters the system. If enabled, the DyeInjection Monitor creates a diagnostic context for the request if one does not already exist. According to its dye configuration, which you define, the DyeInjection monitor injects certain dyes into the diagnostic context based on request properties.

This section covers the following topics:

Dyes Supported by the DyeInjection Monitor

Table B-1 describes the dyes supported by the DyeInjection monitor.

Table B-1 Dyes Supported by the DyeInjection Monitor

Dye	Criterion	Example	Description
ADDR0	IP address of client	ADDR0=127.0.0.1	Inject ADDR0 dye if request originated from IP address 127.0.0.1.
ADDR1	IP address of client	ADDR1=192.168.0.1	Inject ADDR1 dye if request originated from IP address 192.168.0.1.
ADDR2	IP address of client	ADDR2=192.168.0.2	Inject ADDR2 dye if request originated from IP address 192.168.0.2.
ADDR3	IP address of client	ADDR3=192.168.0.3	Inject ADDR3 dye if request originated from IP address 192.168.0.3.
USER0	User identity	USER0=testuser0	Inject USER0 dye if request was originated by user testuser0.
USER1	User identity	USER1=testuser1	Inject USER1 dye if request was originated by user testuser1.
USER2	User identity	USER2=testuser2	Inject USER2 dye if request was originated by user testuser2.
USER3	User identity	USER3=testuser3	Inject USER3 dye if request was originated by user testuser3.
COOKIE0	Value of cookie named “weblogic.diagnostics.dye”	COOKIE0=val0	Inject COOKIE0 dye if value of “weblogic.diagnostics.dye” cookie is val0.
COOKIE1	Value of cookie named “weblogic.diagnostics.dye”	COOKIE1=val1	Inject COOKIE1 dye if value of “weblogic.diagnostics.dye” cookie is val1.
COOKIE2	Value of cookie named “weblogic.diagnostics.dye”	COOKIE2=val2	Inject COOKIE2 dye if value of “weblogic.diagnostics.dye” cookie is val2.
COOKIE3	Value of cookie named “weblogic.diagnostics.dye”	COOKIE3=val3	Inject COOKIE3 dye if value of “weblogic.diagnostics.dye” cookie is val3.

Request Protocols Supported by the DyeInjection Monitor

In addition to configured dyes, the DyeInjection monitor implicitly identifies the request protocol and injects appropriate dyes accordingly to the protocol used. Table B-2 describes the request protocols identified by the DyeInjection monitor.

Table B-2 Request Protocols Identified by the DyeInjection Monitor

Dye	Description
HTTP	Request protocol is HTTP/HTTPS.
T3	Request protocol is T3/T3S.
IIOP	Request protocol is IIOP/IIOPS.
JRMP	Request protocol is JRMP.
SSL	Request protocol is SSL, that is, HTTPS/T3S/IIOPS.

Configuring Dyes in the DyeInjection Monitor

To specify which dyes should be injected into the diagnostic context, you configure the `<properties>` element in the DyeInjection monitor configuration. The diagnostic context is part of the WebLogic Diagnostic Service system resource descriptor. The body of the properties element contains dye property specifications. Each dye property specification must be on a separate line. The dye property specification is of the form:

```
dye-name=request-property
```

For example, the DyeInjection monitor configuration shown in Listing B-1 injects dye `ADDR1` into the diagnostic context if the request originated from IP address `127.0.0.1` and injects dye `USER1` if the request was originated by user `weblogic`. In addition, if the request uses the HTTPS protocol, the HTTP and SSL dyes are added as well.

Listing B-1 Specifying the Properties Element in the DyeInjection Monitor Configuration

```
<wldf-instrumentation-monitor>
  <name>DyeInjection</name>
  <description>Dye Injection monitor</description>
  <monitor-type>standard</monitor-type>
```

```
    <enabled>true</enabled>
    <properties>
       ADDR1=127.0.0.1
       USER1=weblogic
    </properties>
  </wldf-instrumentation-monitor>
```

Using Dye Filtering with Other Diagnostic Monitors

Once you have configured and enabled a DyeInjection monitor, you use dye filtering in other diagnostic monitors (that is, diagnostic monitors other than the DyeInjection monitor) in server or application scopes. These diagnostic monitors can use dye filtering to inspect the dyes injected into the diagnostic context by the DyeInjection monitor.

You can enable or disable dye filtering on individual diagnostic monitors. If dye filtering is enabled on a diagnostic monitor, the diagnostic monitor does not execute its diagnostic actions if the dye mask configured with it is not consistent with the dyes injected into the diagnostic context for that request. Thus, using the dye filtering mechanism, the diagnostic actions are performed only for specific requests, without slowing down other requests.

APPENDIX C

Configuring the WebLogic Diagnostic Service System Resources

This section describes how components of the WebLogic Diagnostic Framework (WLDF) are configured in the new WebLogic Server configuration model. It discusses the specifics of Diagnostic configuration artifacts and provides example XML documents that represent these artifacts. For a complete description of Domain Configuration file processing in this release, see "Domain Configuration Files" in *Understanding Domain Configuration.*

Note: The primary approach for configuration should be via the use of the administration tools provided with WebLogic Server (such as the WebLogic Server Administration Console or the WebLogic Scripting Tool (`weblogic.WLST`), however it is important to understand the artifacts themselves.

WebLogic Server configuration has introduced the concept of System Resource configuration in this release, and the WebLogic Diagnostic Framework (WLDF) leverages this facility for configuring the Harvester, Watch and Notification, and server-level instrumentation components of the framework. This is accomplished via the newly introduced WLDFSystemResourceMBean and the, WLDF System Resource descriptor file, which is supported for external use.

There can be any number of WLDFSystemResourceMBeans defined in a WebLogic Server domains `config.xml`; however, at most one can be targeted to a given server or cluster. As shown by the examples that follow, the WLDF System Resource descriptor is referenced from `config.xml`, and the descriptor file itself, `config/diagnostics/$WLDFResourceName.xml`, is located in the domain-relative directory where `$WLDFResourceName` is the name of the WLDF System Resource.

The following sections describe how to perform WLDF System Resource configuration:

For a discussion of how to configure application-level instrumentation, see "Configuring and Using Application-Level Instrumentation" on page D-1.

WLDF System Resource Configuration for Harvester and Watch and Notification Example

This section provides an example a WLDF system resource configuration for Harvester and Watch and Notification. It includes a `interop-jms.xml` configuration file for completeness, as JMS also utilizes the System Resource model and the dependent JMS Queues and Topics that are leveraged by the configured Notifications are defined in this file.

The following code examples are included in this section:

config.xml Specification Code Example

Listing C-1 is an example of an `config.xml` file that defines a subset of a valid domain and server configuration supporting the Harvester and Watch and Notification components of the diagnostic framework.

Listing C-1 Example: Config.xml Specification

```
<domain xmlns="http://www.bea.com/ns/weblogic/config">
  <!--

  This example shows a subset of a typical domain level configuration
  supporting the Harvester and Watch & Notification capabilities of the
  Diagnostic Framework. It defines the dependent services that will be used
  by the Notification components, JMS, SNMP and SMTP. It also refers to the
  external WLDF System Resource configuration, which contains the configuration
  of the harvester and Watch & Notification components.
  This is NOT a fully functional configuration, as a number of configuration
```

```
elements have been eliminated for brevity.
-->
<!--
Basic Domain and server configuration
-->
<name>mydomain</name>
<server>
  <name>myserver</name>
</server>
<configuration-version>9.0.0.0</configuration-version>
<admin-server-name>myserver</admin-server-name>
<!--
Reference to the actual external WLDF System Resource configuration.  This
will result in the configuration processing loading the WLDF system
resource configuration from the file myWLDF.xml located in the
config/diagnostics directory.
-->
<wldf-system-resource>
  <name>myWLDF</name>
  <target>myserver</target>
</wldf-system-resource>

<!--
Dependent JMSServer configuration, allowing for JMS based Notifications
to be delivered.
-->
<jms-server>
  <name>MyJMSServer</name>
  <target>myserver</target>
  <store-enabled>false</store-enabled>
  <temporary-template-resource xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance" xsi:nil="true"></temporary-template-resource>
  <temporary-template-name xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance" xsi:nil="true"></temporary-template-name>
</jms-server>
<!--
Reference to external JMS System Resource containing the dependent JMS Queue
and JMS Topic  configuration settings.  This will result in the configuration
processing loading the JMS system resource configuration from the file
interop-jms.xml in the config/jms directory.
-->
<jms-system-resource>
  <name>interop-jms</name>
  <target>myserver</target>
  <sub-deployment>
    <name>MyJMSServer</name>
    <target>MyJMSServer</target>
  </sub-deployment>
  <sub-deployment>
```

```
      <name>MyDistributed Queue</name>
      <target>myserver</target>
    </sub-deployment>
  </jms-system-resource>
  <!--
  Dependent SNMPAgent configuration, allowing for SNMP based Notifications to be
  delivered.
  -->
  <snmp-agent>
    <enabled>true</enabled>
    <name>mydomain</name>
    <targeted-trap-destination>MySNMPTrapDestination
          </targeted-trap-destination>
    <snmp-trap-destination>
      <name>MySNMPTrapDestination</name>
      <port>9119</port>
    </snmp-trap-destination>
  </snmp-agent>
  <!--
  Dependent Mail Session configuration, allowing for SMTP based Notifications to
  be delivered.
  -->
  <mail-session>
    <name>MyMailSession</name>
    <jndi-name>MyMailSession</jndi-name>
    <target>myserver</target>
  </mail-session>
</domain>
```


interop-jms.xml Specification Code Example

Listing C-2 provides an example of an `interop-jms.xml` file that is used along with the `config.xml` shown in Listing C-1. The `interop-jms.xml` configuration file is included here for completeness, as JMS also utilizes the System Resource model and the dependent JMS Queues and Topics that are leveraged by the configured Notifications and not defined in `config.xml` file.

Listing C-2 Example: Interop-jms.xml Specification

```
<weblogic-jms xmlns="http://www.bea.com/ns/weblogic/90">
  <queue name="MyJMSQueue">
    <sub-deployment-name>MyJMSServer</sub-deployment-name>
```

```
    <jndi-name>MyJMSQueue</jndi-name>
  </queue>
  <topic name="MyJMSTopic">
    <sub-deployment-name>MyJMSServer</sub-deployment-name>
    <jndi-name>MyJMSTopic</jndi-name>
  </topic>
  <distributed-queue name="MyDistributed Queue">
    <jndi-name>MyDistributedQueue</jndi-name>
    <distributed-queue-member name="MyDistributedQueueMember">
      <queue>MyJMSQueue</queue>
    </distributed-queue-member>
  </distributed-queue>
</weblogic-jms>
```

myWLDF.xml Specification Code Example

Listing C-3 is an example of a WLDF System Resource descriptor configuration file, `myWLDF.xml`, which defines the specific configuration of the Harvester and Watch and Notification components of the Diagnostic Framework.

Listing C-3 Example: myWLDF.xml Specification

```
<!--
This file contains a sample diagnostics resource configuration.  This
configuration monitors a variety of diagnostic data sources and provides
feedback through all of the available destination types.  Documentation is
provided for each individual element of this configuration.
-->
<wldf-resource xmlns="java:weblogic.diagnostics.descriptor">
  <name>myWLDF</name>
  <!--
  This section contains the specifications for the information which will
  be harvested from a running server. Our configuration harvested from
```

```
the ServerRuntime MBean, the runtime MBean for the Harvester, and from
a custom (non-WLS) MBean.

Note that the watch rules of type "Harvester" are driven by harvested data.
If a Harvester specification is not provided to gather the information
on which a watch depends, that watch will never fire.
-->

<harvester>

  <!--
  The sample period for the Harvester.  The Harvester runs periodically
  in discreet cycles.  This value specifies the time between each cycle.
  For example, if the sample time is provided as <I> seconds and the
  harvester fires at time <T> and takes <A> seconds to complete, then the
  targeted time of the next Harvester cycle is T+I.  If the unlikely event
 that I exceeds A seconds, then the next cycle occurs at T+A.
  -->
  <sample-period>5000</sample-period>

  <!--
  The following are the specifications for the data to be gathered from a
  running server by the Harvester.  The means of specification is type
  based.  A separate instance of <harvested-type> is provided for each
  type to be collected.

  For each type, the user may optionally specify subsets of information
 to be collected. Using tags <harvested-attribute> it is possible to cause
  only metrics to be collected for instances of the specified type.  Using
 tags <harvested-instance>, the set of instances from which data is
 collected can be
 specified. If no <harvested-attribute> is present, then all attributes
 which are defined for the type are collected.  If no <harvested-instance>
 is present, then all instances which are present at the time of each
 Harvester cycle are collected.  If at least one instance of a tag is
 provided, then the set of all instances of that tag type defines the
 complete set of data to be collected.  That is, data which is not
 explicitly specified will be excluded from the collection.

 TYPES: The Harvester supports two types of MBeans, 1) WLS MBeans and 2)
 Custom MBeans. WLS MBeans are those which come packaged as part of WLS.
```

```
Customers may also create their own MBeans which may be Harvested provided
 they are registered in the local runtime MBean server.

 There is a small difference in how WLS and customer types are specified.
  For WLS types, the type name is the name of the Java interface which
  defines that MBean.  For example, the server runtime MBean's type name
  is "weblogic.management.runtime.ServerRuntimeMBean".
 For custom MBeans, the type name is the name of the implementing class.

  Examples of type specifications are provided below.

  INSTANCES: As mentioned above using sub-tag <harvested-instance>, the
  user can define a specific set of instances to be collected for a type.
  In general, an instance is specified by providing its JMX ObjectName in
  JMX canonical form.

  Examples of instance specifications are provided below.

 ATTRIBUTES: As mentioned above using sub-tag <harvested-attribute>, the
 user can define a specific set of attributes to be collected for a type.
In general, an attribute is specified by providing its name. The first
character should be capitalized.  For example, an attribute defined with
getter method getFoo() is named "Foo".

VALIDATION: The diagnostics system attempts to validate configuration as
soon as possible.  Most configuration is validated at system startup and
whenever a dynamic change is committed.  However, due to limitations in
JMX, custom MBeans cannot be validated until instances of those MBeans
have been registered in the MBean server.
-->

<!--
 The following tag specifies that the server runtime MBean is to be
 harvested. Because no <harvested-instance> sub-tag is present, all
 instances of the type will be collected. However, since there is always
only one instance of the server runtime mbean, there is no need to provide
a specific list of instances. And because there are no
<harvested-attribute> sub-tags present, all available attributes of the
MBean are harvested.
 -->
 <harvested-type>
   <name>weblogic.management.runtime.ServerRuntimeMBean</name>
```

```
  </harvested-type>
  <!--
  This tag specifies that the Harvester runtime MBean be harvested. As
  above, because there is only one Harvester runtime MBean, there is no
  need to provide a specific list of instances. Using sub-tag
  <harvested-attribute> the user has specified that only two of the
  available attributes of the Harvester runtime MBean be harvested,
  TotalSamplingTime and CurrentSnapshotElapsedTime.
  -->
  <harvested-type>
    <name>weblogic.management.runtime.WLDFHarvesterRuntimeMBean</name>
    <harvested-attribute>TotalSamplingTime</harvested-attribute>
   <harvested-attribute>CurrentSnapshotElapsedTime</harvested-attribute>
  </harvested-type>
  <!--
  This tag specifies that a single instance of a custom MBean type be
  Harvested. In this case, because this is a custom MBean, the name is the
  implementation class. This specification utilizes the sub-tag
  <harvested-instance> to specify that only two instances of this type be
  harvested. Each instance is specified by providing the canonical
  representation of its JMX ObjectName. Since no instances of
 <harvested-attribute> are specified, all attributes will
  be Harvested.
  -->
  <harvested-type>
    <name>myMBeans.MySimpleStandard</name>
    <harvested-instance>myCustomDomain:Name=myCustomMBean1
    </harvested-instance>
    <harvested-instance>myCustomDomain:Name=myCustomMBean2
    </harvested-instance>
  </harvested-type>
 </harvester>
<!--
This section contains the specifications for the watches and
notifications. Watches are used to identify certain significant
situations, which the user wishes to trap. Watches, in turn, can be
associated with a set of notifications.  The associated notifications are
used to impart knowledge of the significant situation to a set of users
```

and/or applications. There are several types of notifications which support delivery of the knowledge using different mechanisms, specifically SMTP (e.g. email), JMS, JMX, and SNMP. In addition, the user can use a notification to cause a diagnostic image to be generated.
Each watch and notification can be individually enabled and disabled using sub-tag <enabled>. In addition, the entire watch and notification facility can be similarly enabled and disabled.The value default is enabled=true.
The <watch-notification> tag contains a sub-tag, <log-watch-severity>, which affects the firing of notifications triggered by log-rule watches. If the maximum severity level of the log messages which triggered the watch do not at least equal the provided severity level, then the resulting notifications are not fired. Note that this only applies to notifications fired by watches which have log rule types. This tag should not be confused with the <severity> tag defined on watches. The <severity> tag assigned a severity to the watch itself, whereas the <log-watch-severity> tag controls which notifications get fired by log-rule watches.
WATCHES - There is a single sub-tag, <watch>, used to define a watch. Each watch can monitor one of three things, 1) the set of harvested MBeans in the local runtime MBean server, 2) the set of messages generated into the server log, and 3) the set of events which are generated by the diagnostic instrumentation facility.
The type of information being monitored is defined by the watch rule type, specified using the sub-tag <rule-type>. The defined values of this tag are "Harvester", "Log", and "EventData". Watches with different rule types differ in two ways: 1) The rule syntax for specifying the conditions being monitored are unique to the type and 2) Log and event rules are triggered in real time whereas Harvester rules have an inherent latency.

Each watch must contain a logical expression that defines when the significant events occur that the watch is designed to trap. This expression is provided using tag <rule-expression>. The syntax is a modified SQL syntax similar to that used in the diagnostic Accessor facility.
Each watch also contains a list of notifications that are fired whenever the watch triggers. The notifications are specified using the sub-tag <notification>. The value of this tag is a comma-separated list of notifications to fire. Each notification is specified by its name as

```
defined in this file.
Watches can be specified to trigger repeatedly, or to trigger only when
explicitly primed.  For watches that trigger repeatedly, the user can
optionally define a minimum spacing between occurrences. The tag
<alarm-type> is used to define whether a watch automatically repeats, and,
if so, how often. A value of "None" causes the watch to trigger whenever
possible.  A value of "AutomaticReset" also causes the watch to trigger
whenever possible, except that subsequent occurrences cannot occur any
sooner than the millisecond interval specified by sub-tag
<alarm-reset-period>. A value of "ManualReset" causes the watch to fire a
single time. After it fires manual intervention is required to make it
fire again. For example, the user can use the WatchNotification runtime
MBean to reset a manual watch.  The default for <alarm-type> is "None".
Watches contain a severity value which is passed through to the recipients
of notifications.  The permissible severity values are as defined in the
logging subsystem. The severity value is specified using sub-tag
<severity>. The default is "Notice".

Each watch can also be individually enabled and disabled, using sub-tag
<enabled>.  When diaabled, the watch does not trigger and corresponding
notifications do not fire. If the more generic watch/notification
flag is disabled, it causes all individual watches to be effectively
disabled (i.e. the value of this flag on a specific watch is ignored).

HARVESTER WATCHES - A Harvester watch can monitor any runtime MBean in the
local runtime MBean server provided that the pertinent MBean data has been
collected by the Harvester. Note that a potential mistake is to define
watch rules which monitor information that is not specified to be
collected.

Harvester watches are fired in response to a Harvester cycle.  So for
Harvester watches, the Harvester sample period defines a worst case time
interval between when a situation is identified, and when it can be
reported though a notification.  On average, the delay will be
SamplePeriod/2.

Note that the watch rules of type "Harvester" are driven by harvested data.
If a Harvester specification is not provided to gather the information
on which a watch depends, that watch will never fire.
```

LOG WATCHES - Log watches are used to monitor the occurrence of specific messages and/or strings in the server log. Watches of this type trigger as soon as the log messages is issued.

INSTRUMENTAION WATCHES - Instrumentation watches are used to monitor the events from the diagnostic instrumentation facility. Watches of this type trigger as soon as the event is posted.

NOTIFICATIONS - There are unique notification types for each delivery mechanism. The corresponding sub-tags are <image-notification>, <jmx-notification>, <jms-notification>, <snmp-notification>, and <smtp-notification>. These notification types share two sub-tags <name> and <enabled>. The values of the notification name is significant as it is used in the <notification> tag of a watch to map the watch to its corresponding notifications.

The <enabled> sub-tag on a notification behaves analogously to the corresponding tag on a watch.

Beyond the <name> and <enabled> tags, each notification type is unique.

JMX NOTIFICATION - JMX notifications are used to cause a JMX notification to be fired in response to the triggering of an associated watch. For this purpose, for each jmx notification defined, the diagnostic system automatically generates an MBean on which JMX listeners can be registered. The ObjectName for this MBean is derived directly from the sub-tag <name> value in the notification definition. At runtime, JMX events(notifications) are issued on behalf of the generated MBeans whenever an associated watch triggers.

JMS NOTIFICATION - JMS notifications are used to post messages to JMS topics and/or queues in response to the triggering of an associated watch. Using the sub-tags <destination-jndi-name> and <connection-factory-jndi-name>, the user defines how the message is to be delivered.

SNMP NOTIFICATION - SNMP notifications are used to post SNMP traps in response to the triggering of an associated watch. To define an SNMP notification the user need only provide a notification name. Generated traps contain the names of both the watch and notification that caused trap to be generated.

```
SMTP NOTIFICATION - SMTP notifications are used to send messages (email)
over the SMTP protocol in response to the triggering of an associated
watch. To define an SMTP notification the user must provide the configured
SMTP session using sub-tag <mail-session-jndi-name>, and provide a list of
at least one recipient using sub-tag <recipients>.  An optional subject
and/or body can be provided using sub-tags <subject> and <body>
respectively. If these are not provided, they will be defaulted.

IMAGE NOTIFICATION - Image notifications are used to cause a diagnostic
image to be generated in response to the triggering of an associated watch.
The user provides a single piece of information, the directory into which
the image is to be placed, relative to the server's root directory. The
relative directory name is specified using sub-tag <image-directory>.
-->
<watch-notification>

  <!--
 The following Harvester watch is used to monitor several runtime MBeans.
 When the rule triggers six different notifications are fired, including
 a JMX notification, an SMTP notification, an SNMP notification, an image
 notification, and JMS notifications for both a topic and a queue.

 The corresponding watch rule is a logical expression comprised of 4
 Harvester "variables". The rule has the form ( (A >= 100) && (B > 0) )
 || C || D.equals("active").  Each variable is of the form
 {entityName}//{attributeName}; where {entityName} is the JMX ObjectName
 as registered in the runtime MBean server and {attributeName} is the
 name of an attribute defined on that MBean type.  The comparison operators
 are qualified so as to be valid in XML.

 For Harveter-based watches, in order to cause the watch to trigger, a
 variable must be defined over an MBean that is currently being harvested.
This means that instances of the MBean must be registered in the MBean
server and the Harvester must be configured to Harvest the appropriate
attributes of the MBean.

Note that the third clause of the expression (C==true) will never evaluate
to true because the Accessor runtime MBean is not configured to be
harvested. Similarly the last clause (D.equals("active")) also will not
evaluate to true because the MBean instance provided is not in the provided
```

```
list of Harvestable instances (see <harvested-type> entry
myMBeans.MySimpleStandard above). However, these issues do not prevent the
rule from firing due to the initial clauses ( (A >= 100) && (B > 0) ). In
fact, this rule is designed to almost always fire.

This watch utilizes an alarm type of AutomaticReset which means that it
may re-fire repeatedly provided that the last time it fired was longer
than the interval provided by the provided alarm reset period (in this
case 10000 milliseconds).
 The severity level provided, "Warning" has no effect on the triggering
 of the watch, but will be passed on through the notifications.
   -->
  <watch>
    <name>simpleWebLogicMBeanWatchRepeatingAfterWait</name>
    <enabled>true</enabled>
    <rule-type>Harvester</rule-type>
    <rule-expression>
     (${mydomain:Name=WLDFHarvesterRuntime,ServerRuntime=myserver,Type=
     WLDFHarvesterRuntime,WLDFRuntime=WLDFRuntime//TotalSamplingTime}
           &gt;= 100
         AND
        ${mydomain:Name=myserver,Type=
                 ServerRuntime//OpenSocketsCurrentCount} &gt; 0)
         OR
       ${mydomain:Name=WLDFWatchNotificationRuntime,ServerRuntime=
            myserver,Type=WLDFWatchNotificationRuntime,
             WLDFRuntime=WLDFRuntime//Enabled} = true
         OR
       ${myCustomDomain:Name=myCustomMBean3//State} =
               'active')</rule-expression>
    <severity>Warning</severity>
    <alarm-type>AutomaticReset</alarm-type>
    <alarm-reset-period>10000</alarm-reset-period>
    <notification>myJMXNotif,myImageNotif,
       myJMSTopicNotif,myJMSQueueNotif,mySNMPNotif,
       mySMTPNotif</notification>
  </watch>
  <!--
```

```
The following image notification causes an image file of a standard name
 to be generated into the provided directory.
The directory name specified is relative to the servers root directory.
 For example, in this case image files will be generated into
 "<domain-dir>/servers/myserver/images".

 Image file names are generated using the current timestamp (e.g.
 diagnostic_image_myserver_2004_12_09_13_40_34.zip), so this particular
 notification can fire many times, resulting in a separate image file
 each time.

 This notification is currently disabled.  It will not fire until it is
 enabled.
 -->
 <image-notification>
   <name>myImageNotif</name>
   <enabled>false</enabled>
   <image-directory>images</image-directory>
 </image-notification>
 <!--
The following two JMS notification cause JMS messages to be sent through
the provided topics and queues using the specified connection factory.
For this to work properly, JMS must be properly configured in config.xml.
 -->
 <jms-notification>
   <name>myJMSTopicNotif</name>
   <destination-jndi-name>MyJMSTopic</destination-jndi-name>
   <connection-factory-jndi-name>weblogic.jms.ConnectionFactory
        </connection-factory-jndi-name>
 </jms-notification>
 <jms-notification>
   <name>myJMSQueueNotif</name>
   <destination-jndi-name>MyJMSQueue</destination-jndi-name>
   <connection-factory-jndi-name>weblogic.jms.ConnectionFactory
       </connection-factory-jndi-name>
 </jms-notification>
 <!--
 The following JMX notification causes an MBean to be automatically
 generated and registered.  This MBean posts JMX events(notifications)
```

```
whenever this diagnostic notification fires.  To receive these
messages, a user registers through the JMX server for JMX events on the
generated and registered MBean.
The pattern used for the ObjectNames of the generated MBeans is:
   <domainName>:ServerRuntime=<serverName>,
     Name=<notificationName>,Type=WLDFWatchJMXNotificationRuntime,
     WLDFWatchNotificationRuntime=WatchNotification,
     WLDFRuntime=WLDFRuntime
In this specific case the ObjectName will be:

  mydomain:ServerRuntime=myserver,Name=myJMXNotif,
   Type=WLDFWatchJMXNotificationRuntime,
   WLDFWatchNotificationRuntime=WatchNotification,
   WLDFRuntime=WLDFRuntime
-->
<jmx-notification>
  <name>myJMXNotif</name>
  <enabled>false</enabled>
</jmx-notification>
<!--
The following SMTP notification causes an SMTP (email) message to be
distributed through the configured SMTP session to the configured
recipients. For this to work properly, the SMTP session must be
properly configured in config.xml.
In this notification specification a custom subject and body are
provided.  If not provided, these would be defaulted.
-->
<smtp-notification>
  <name>mySMTPNotif</name>
  <mail-session-jndi-name>MyMailSession</mail-session-jndi-name>
  <subject>Critical Problem!</subject>
  <body>A system issue occurred. Call Winston at 404-575-4433 ASAP.
       Reference number 81767366662AG-USA23.</body>
  <recipients>administrator@myCompany.com</recipients>
</smtp-notification>
<!--
The following SNMP notification causes an SNMP trap to be generated
whenever the notification fires. For this to work properly, SNMP must
```

```
    be properly configured in config.xml. The resulting trap is of type 85.
    It contains the following values (configured values in angle brackets
    "<>"):
    .1.3.6.1.4.1.140.625.100.5   timestamp (e.g. Dec 9, 2004 6:46:37 PM EST)
     .1.3.6.1.4.1.140.625.100.145  domainName (e.g. mydomain")
     .1.3.6.1.4.1.140.625.100.10   serverName (e.g. myserver)
     .1.3.6.1.4.1.140.625.100.120  <severity> (e.g. Notice)
     .1.3.6.1.4.1.140.625.100.105  <name> [of watch] (e.g.
           simpleWebLogicMBeanWatchRepeatingAfterWait)
     .1.3.6.1.4.1.140.625.100.110  <rule-type> (e.g. HarvesterRule)
     .1.3.6.1.4.1.140.625.100.115  <rule-expression>
     .1.3.6.1.4.1.140.625.100.125  values which caused rule to
         fire (e.g..State =
         null,weblogic.management.runtime.WLDFHarvesterRuntimeMBean.
         TotalSamplingTime = 886,.Enabled =
         null,weblogic.management.runtime.ServerRuntimeMBean.
         OpenSocketsCurrentCount = 1,)
     .1.3.6.1.4.1.140.625.100.130  <alarm-type> (e.g. None)
     .1.3.6.1.4.1.140.625.100.135  <alarm-reset-period> (e.g. 10000)
    .1.3.6.1.4.1.140.625.100.140  <name> [of notification] (e.g.mySNMPNotif)
     -->
     <snmp-notification>
       <name>mySNMPNotif</name>
     </snmp-notification>
   </watch-notification>
</wldf-resource>
```


This `myWLDF.xml` resource file, (and all wldf-resource xml descriptor documents), complies with an XML Schema that can currently be located in a JAR file in the file system at `BEA_HOME/weblogic90/server/lib/schema/diagnostics-binding.jar` and within the JAR file at `META-INF/schemas/schema-2.xsd`. The XML Schema enables the use of XML editing tools to modify and validate any wldf-resource files.

Note: Post Beta, the supporting schema will be accessible `at www.bea.com` with all other officially supported `WebLogic .xsd` files.

WLDF System Resource Configuration for Server-Level Instrumentation Example

As with the preceding example, the Server-level Instrumentation component of the WebLogic Diagnostic Framework is also configured via a WLDF System Resource. What follows is an additional configuration example identifying the artifacts necessary to configure the Server-level Instrumentation component of the Diagnostic Framework, along with some additional configuration details specific to enabling Instrumentation in the Server.

The following topics are covered in this section:

Enabling WebLogic Server Instrumentation

As a function of the current WebLogic Server classloading model and the current byte-code Instrumentation implementation, the Administrator must explicitly enable the Instrumentation capabilities via command line property specification at server start time. By default the WebLogic Server Instrumentation capabilities are disabled.

To enable instrumentation for WebLogic Server classes as well as application classes running within WebLogic Server, the Administrator must start the server with:

```
java -Dweblogic.diagnostics.instrumentation=all
```

To enable instrumentation only for application classes running within WebLogic Server, the Administrator must start the server with:

```
java -Dweblogic.diagnostics.instrumentation=apps
```

Example myWLDF.xml Specification

What follows is an example of a WLDF System Resource descriptor configuration file, myWLDF.xml, which defines the specific configuration Server-level Instrumentation component of the Diagnostic Framework. (Please refer to the section on the diagnostic monitor library for a list of applicable monitors.)

Listing C-4 Example: myWLDF.xml

```
<!--
This example shows a typical server level instrumentation configuration
descriptor.
It defines a WLDF System Resource. A WLDF system resource can also contain
otherdiagnostic components such as the harvester and watch-and-notification
component.
For the sake of simplicity, those components are not shown in this example.
-->
<wldf-resource xmlns="java:weblogic.diagnostics.descriptor">
  <!-- Name of the WLDF system resource -->
  <name>myWLDF</name>
  <!-- Server level instrumentation configuration -->
  <instrumentation>
    <!-- If true, enable weaving diagnostic code into server classes -->
    <enabled>true</enabled>
    <!--
      Add the DyeInjection monitor, which will inject dyes into requests
      as they enter the system, based on request properties
    -->
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <description>Dye Injection monitor</description>
      <!-- For DyeInjection monitor, this must always be "standard" -->
      <monitor-type>standard</monitor-type>
      <!-- Enable DyeInjection monitor -->
      <enabled>true</enabled>
      <!--
       Add dyes into the diagnostic context when request enters the system,
       as per the request properties. In this example, dye ADDR1 will be
       added, if the request originated from IP address 127.0.0.1 and dye
       USER1 will be added if the originator was "weblogic". These dyes may
      be inspected and acted upon by other server level or application level
      diagnostic monitors, if dye-filtering is enabled for them.
      This element specifies the properties used to configure the
      DyeInjection monitor. Each property specification must be on a
      separate line and, is of the form:
```

```
        dye-name=value
       Please refer to the section on Dye Filtering for the list of available
       dyes supported by the DyeInjection monitor.
       -->
       <properties>
         ADDR1=127.0.0.1
         USER1=weblogic
       </properties>
     </wldf-instrumentation-monitor>
     <!--
       Add the Connector_Before_Inbound monitor to the server classes. It is
       enabled. So when the program flows through it, it will execute its
       attached action to generate a trace event.
      -->
     <wldf-instrumentation-monitor>
       <name>Connector_Before_Inbound</name>
       <monitor-type>delegating</monitor-type>
       <enabled>true</enabled>
       <action>TraceAction</action>
     </wldf-instrumentation-monitor>
   </instrumentation>
</wldf-resource>
```

Adding or removing monitors for the targeted WLDF System Resource requires that the WebLogic Server be restarted in order for the changes to take effect. However, significant configuration aspects of Server-level Instrumentation can be changed dynamically without having to restart the server. These changes can be made using the WebLogic Server Administration Console or via the new WebLogic Scripting Tool, (weblogic.WLST). Allowed dynamic changes are as follows:

- Enable or disable monitors
- Enable or disable dye filtering on the monitors
- Change dye-mask, monitor properties
- Add or remove actions attached to a delegating monitor.

APPENDIX D

Configuring and Using Application-Level Instrumentation

The Instrumentation component of the WebLogic Diagnostic Framework (WLDF) provides facilities to add diagnostic instrumentation code to WebLogic Server classes as well as applications running on the server. Although the machinery for adding such code is the same for classes and applications, the way the instrumentation is configured for server classes versus application classes is slightly different.

This section describes how to configure the application-scoped components of the WebLogic Diagnostic Framework (WLDF) using the new WebLogic Server configuration model. It discusses the specifics of the diagnostic configuration tools provided and provides example XML documents that show how to use these tools.

The following sections describe how to add diagnostic instrumentation code to applications.

For a discussion of how to configure Server-level Instrumentation, see "Configuring the WebLogic Diagnostic Service System Resources" on page C-1.

Configuring a WebLogic Diagnostic Descriptor

Application-level instrumentation is configured with the `META-INF/weblogic-diagnostics.xml` descriptor in an application archive. The Instrumentation component adds diagnostic instrumentation code to matching application classes when the classes are loaded, if the descriptor exists in the `META-INF` directory. This descriptor

may be specified inside an application ear archive or a stand-alone module such as a war, rar or ejb, and can be done for both exploded and unexploded archives. (Note that if an application ear archive contains war, rar or ejb modules, which have the `weblogic-diagnostics.xml` descriptors in their `META-INF` directory, those descriptors will be ignored. The implementation currently requires that for instrumentation configuration for an application EAR, the `weblogic-diagnostics.xml` must exist in the application's `META-INF` directory and will be applied to all contents of that EAR.)

The form of the instrumentation descriptor is the same as that of the WLDF System Resource descriptor discussed in the prior section, with the exception that only the instrumentation component is available in the application scope. Only the diagnostic monitors from the library which are designated to be used in application scope can be used in the `weblogic-diagnostics.xml` descriptor. For a list of applicable monitors, see "Diagnostic Monitor Library" on page A-1. Listing D-1 shows an example of a typical `weblogic-diagnostics.xml` instrumentation descriptor within an application that adds diagnostics instrumentation to the application.

Listing D-1 Example: weblogic-diagnostics.xml

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

<!--
This example shows a typical weblogic-diagnostics.xml instrumentation
descriptor within an application to add diagnostics instrumentation to the
application.
-->

<wldf-resource xmlns="java:weblogic.diagnostics.descriptor">
  <instrumentation>
    <!--
    Enable weaving of diagnostic code into eligible classes within the
application.
    -->
    <enabled>true</enabled>

    <!--
    Add the Servlet_Before_Service monitor, which will add diagnostic code
   at the entry of a servlet's (or jsp's) service method. When executed, the
   attached actions will perform their respective diagnostic activity. The
   TraceAction will generate a trace event. The DisplayArgumentsAction will
```

```
    generate an event which captures the arguments passed into affected
    methods.
    -->
    <wldf-instrumentation-monitor>
      <name>Servlet_Before_Service</name>
      <monitor-type>delegating</monitor-type>
      <enabled>true</enabled>

      <!--
      Actions attached to the monitor. More than one action may be attached
      -->
      <action>TraceAction</action>
      <action>DisplayArgumentsAction</action>
    </wldf-instrumentation-monitor>

    <!--
    Add the Servlet_Around_Service monitor, which will add diagnostic code
    at the entry and exit of a servlet's (or jsp) service method. When
    executed, the attached TraceElapsedTimeAction will compute the
    time spent while executing the method and generate an event to capture
    the elapsed time.
    -->

    <wldf-instrumentation-monitor>
      <name>Servlet_Around_Service</name>
      <monitor-type>delegating</monitor-type>
      <enabled>true</enabled>
      <action>TraceElapsedTimeAction</action>
    </wldf-instrumentation-monitor>

    <!--
    Add a custom monitor. Custom monitors can only be used within
applications
    -->
    <wldf-instrumentation-monitor>
      <name>MyCustomMonitor</name>
      <monitor-type>custom</monitor-type>
      <enabled>true</enabled>

      <!--
      The dye mask used for dye-filtering. The monitor will execute its
```

```
      actions only if the dyes injected into the diagnostic context by the
      DyeInjection monitor are consistent with the dye-mask specified here.
      In this example, together with the configuration of the DyeInjection
      monitor specified in the Server Level Instrumentation example, the
      action attached to this monitor will be executed only if the request
      originator is "weblogic".
      -->
      <dye-mask>USER1</dye-mask>

      <!--
      Enable dye filtering. The dye mask is ignored if dye-filtering is
      disabled.
      -->
      <dye-filtering-enabled>true</dye-filtering-enabled>
      <action>TraceAction</action>

      <!--
      Relative position of the diagnostic code, with respect to matching
      joinpoints. In this case, the diagnostic code will be emitted at the
      entry of affected methods. This attribute is relevant only for custom
      monitors.
      -->
      <location-type>before</location-type>

      <!--
      Pointcut expression identifying matching joinpoints. In this example,
      public getter methods from any class and returning any type (including
      void) are fair game. The methods may have any number of arguments of
      any type. Diagnostic code will be emitted in the body of the affected
      methods. This attribute is relevant only for custom monitors.
      -->
      <pointcut>
        execution(public * * get*(...));
      </pointcut>
    </wldf-instrumentation-monitor>
  </instrumentation>
</wldf-resource>
```

Using WebLogic Tools to Enable and Control Application-Level Instrumentation

You can use the WebLogic tools to perform the following tasks:

Deploying the Application

Simply by deploying the application with the `weblogic-diagnostics.xml` descriptor described above, the application classes will be instrumented. The Administrator can use any of the standard WebLogic Server tools provided for controlling deployment, including the WebLogic Administrative Console or the new WebLogic Scripting Tool (WLST). As the application classes are loaded, diagnostic code will be added to eligible classes as per the `weblogic-diagnostics.xml` descriptor.

Support for Dynamic Control of the Instrumentation Configuration

Dynamic control of instrumentation monitors within applications is provided with the Deployment Plan (JSR88) functionality. With deployment plans, one can add diagnostic monitors into applications after they are built, without having to modify the application archives. The application must, however, include at least an empty `weblogic-diagnostics.xml` descriptor for application instrumentation to work. With deployment plans, you can add monitors which are not present in the descriptor in the application archive. You can also update certain attributes of the monitors using deployment plans without having to restart the server or redeploy the application. For example, you can enable/disable diagnostic monitors or add/remove actions to monitors without redeploying the application.

Creating Deployment Plans using weblogic.Configure

Detailed information regarding deployment plan capabilities can be found at "Configuring Applications for Production Deployment" in *Deploying Applications to WebLogic Server*. The `weblogic.Configure` tool can be used to create an initial deployment plan, and allows for the ability to interactively override specific properties of the `weblogic-diagnostics.xml` descriptor. For example:

```
java weblogic.Configure -plan plan.xml -type ear myApp.ear
```

This tool prompts for values of attributes that can be overridden. If no value is supplied for an attribute, the attribute value is not overridden and the value from the descriptor is retained.

Listing D-2 shows an example of the deployment plan generated by the `weblogic.Configure` tool. In this example, the `enabled` attribute of the Servlet_Before_Service monitor is over-ridden to `false`, thus, disabling the monitor.

Listing D-2 Example: Deployment Plan

```
<deployment-plan xmlns="http://www.bea.com/ns/weblogic/90"
global-variables="false">
  <application-name>testapp</application-name>
  <variable-definition>
    <variable>
      <name>WLDFInstrumentationMonitor_Servlet_Before_Service_
        Enabled_11026294848990</name>
      <value>false</value>
    </variable>
  </variable-definition>
  <module-override>
    <module-name>testapp</module-name>
    <module-type>ear</module-type>
    <module-descriptor external="true">
      <root-element>weblogic-application</root-element>
      <uri>META-INF/weblogic-application.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>application</root-element>
      <uri>META-INF/application.xml</uri>
```

```
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>wldf-resource</root-element>
    <uri>META-INF/weblogic-diagnostics.xml</uri>
    <variable-assignment>
      <name>WLDFInstrumentationMonitor_Servlet_Before_Service_
            Enabled_11026294848990</name>
      <xpath>/wldf-resource/instrumentation/wldf-instrumentation
           -monitor/[name="Servlet_Before_Service"]/enabled</xpath>
    </variable-assignment>
  </module-descriptor>
</module-override>
<module-override>
  <module-name>web</module-name>
  <module-type>war</module-type>
  <module-descriptor external="true">
    <root-element>weblogic-web-app</root-element>
    <uri>WEB-INF/weblogic.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>web-app</root-element>
    <uri>WEB-INF/web.xml</uri>
  </module-descriptor>
</module-override>
<module-override>
  <module-name>web1</module-name>
  <module-type>war</module-type>
  <module-descriptor external="true">
    <root-element>weblogic-web-app</root-element>
    <uri>WEB-INF/weblogic.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>web-app</root-element>
    <uri>WEB-INF/web.xml</uri>
  </module-descriptor>
</module-override>
<module-override>
  <module-name>web2</module-name>
```

```
    <module-type>war</module-type>
    <module-descriptor external="true">
      <root-element>weblogic-web-app</root-element>
      <uri>WEB-INF/weblogic.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>web-app</root-element>
      <uri>WEB-INF/web.xml</uri>
    </module-descriptor>
  </module-override>
  <config-root>C:\Documents and Settings\bea\Local
       Settings\Temp\config\deployments\testapp\plan
  </config-root>
</deployment-plan>
```

Deploying an Application with Deployment Plans

For dynamic control over diagnostic monitors in the application, the application must be deployed with a deployment plan. Again, the Administrator can use any of the standard WebLogic Server tools provided for controlling deployment, including the WebLogic Administrative Console or the new WebLogic Scripting Tool (WLST). For example, the following is a WLST command that deploys an application with a corresponding deployment plan.

```
wls:/mydomain/serverConfig> deploy('myApp', './myApp.ear', 'myserver',
      'nostage', './plan.xml')
```

After deployment, the effective diagnostic monitor configuration is a combination of the original descriptor, combined with the overridden attribute values from the plan. Note that if the original descriptor did not include a monitor with the given name and the plan overrides an attribute of such a monitor, the monitor is added to the set of monitors to be used with the application. This way, if your application is built with an empty `weblogic-diagnostics.xml` descriptor, you can add diagnostic monitors to the application during the deployment process, without having to modify the application archive.

Updating an Application with a Modified Plan

Users can dynamically control monitors which are in use by the deployed application, by simply modifying the deployment plan and updating the application using the already identified tools. For example, you can enable/disable monitors and add/remove actions attached to them. You can also enable/disable dye-filtering and modify the dye mask for the monitor dynamically. Such changes take effect immediately without having to redeploy the application. For example, the following WLST command updates the application with a modified plan value:

```
wls:/mydomain/serverConfig> updateApplication('testapp',
                                               'c:/tmp/plan.xml')
```

APPENDIX E

Accessing Data Online and Offline

This section covers the following topics:

How To Use the Data Accessor

This document details how to make use of the features available in the Data Accessor. The Data Accessor provides access to the diagnostic data that is generated by the server and stored in the different types of logs.

This section covers the following topics:

Using the Data Accessor in Off-line Mode

The offline mode of the Data Accessor is incorporated within WebLogic Scripting Tool (WLST). To access data in off-line mode, start `weblogic.WLST` and at the off-line prompt type `help('exportDiagnosticData')` to print the help for the export command. This command takes several parameters which can be supplied as key value pairs separated by a comma. You must supply the type of the log, the location of the log files, the store directory for the indexes which by default is under the `data/store/diagnostics` directory under each server directory in the domain. The export function also takes a query expression which can be used to filter the

data retrieved. The output format of the data is XML. Refer to the help printed by WLST for the exact syntax or see "Using the WebLogic Scripting Tools exportDiagnosticData Command" on page E-4.

On-line Data Accessor Code Example

Listing E-1 shows the code example for using the Data Accessor. The example demonstrates how to use JMX and includes query examples to retrieve data from the WebLogic Server and HTTP Access logs.

Listing E-1 Example: Data Accessor With JMX

```
package weblogic.diagnostics.test;
import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Hashtable;
import java.util.Iterator;
import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

public class JMXAccessorTest {
  private static final String JNDI = "/jndi/";
  public static void main(String[] args) {
    try {
      if (args.length != 2) {
        System.err.println("Incorrect invocation. Correct usage is java
           weblogic.diagnostics.test.JMXAccessorTest
           [logicalName] [query]");
        System.exit(1);
      }
      String logicalName = args[0];
      String query = args[1];

      MBeanServerConnection mbeanServerConnection =
            lookupMBeanServerConnection();
      ObjectName service = new ObjectName("weblogic:Name=RuntimeService");
      ObjectName serverRuntime =  (ObjectName)
           mbeanServerConnection.getAttribute(service, "ServerRuntime");
      ObjectName wldfRuntime = (ObjectName)
           mbeanServerConnection.getAttribute(serverRuntime, "WLDFRuntime");
```

```
    ObjectName wldfAccessRuntime = (ObjectName)
        mbeanServerConnection.getAttribute(wldfRuntime, "WLDFAccessRuntime");

    ObjectName wldfDataAccessRuntime = (ObjectName)
          mbeanServerConnection.invoke(wldfAccessRuntime,
          "lookupWLDFDataAccessRuntime", new Object[] {logicalName},
          new String[] {"java.lang.String"});

    String cursor = (String)
        mbeanServerConnection.invoke(wldfDataAccessRuntime,
        "openCursor", new Object[] {query},
        new String[] {"java.lang.String"});

    int fetchedCount = 0;
    do {
      Object[] rows = (Object[])
        mbeanServerConnection.invoke(wldfDataAccessRuntime,
        "fetch", new Object[] {cursor},
        new String[] {"java.lang.String"});

      fetchedCount = rows.length;

      for (int i=0; i<rows.length; i++) {
       StringBuffer sb = new StringBuffer();
        Object[] cols = (Object[]) rows[i];
        for (int j=0; j<cols.length; j++) {
          sb.append("Index " + j + "=" + cols[j].toString() + " ");
        }
        System.out.println("Found row = " + sb.toString());
      }
    } while (fetchedCount > 0);

    mbeanServerConnection.invoke(wldfDataAccessRuntime,
        "closeCursor", new Object[] {cursor},
        new String[] {"java.lang.String"});

  } catch(Throwable th) {
    th.printStackTrace();
    System.exit(1);
  }
}

private static MBeanServerConnection lookupMBeanServerConnection () throws
           Exception {

  // get server host and port
  // construct jmx service url
  JMXServiceURL serviceURL;
  serviceURL = new JMXServiceURL("iiop", "localhost", 9991,
      JNDI + "weblogic.management.mbeanservers.runtime");
```

```
    // specify the user and pwd. Also specify weblogic provide package
    Hashtable h = new Hashtable();
    h.put(Context.SECURITY_PRINCIPAL,"system");
    h.put(Context.SECURITY_CREDENTIALS,"gumby1234");
    h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
          "weblogic.management.remote");
    // Get jmx connector
    JMXConnector connector = JMXConnectorFactory.connect(serviceURL,h);

    // return mbean server connection class
    return connector.getMBeanServerConnection();

  } // M - getMBeanServerConnection
}
```

Using the WebLogic Scripting Tools exportDiagnosticData Command

You use the exportDiagnosticData command to access data offline. You can use this command to access archived data on the machine on which the server was running.

This section covers the following topics:

Off-line Access exportDiagnosticData Command Help

This section describes the `weblogic.WLST exportDiagnosticData` command.

Command Description

The `exportDiagnosticData` command exports the diagnostic data created by the WebLogic Server to XML format in the offline mode.

Command Syntax

The `exportDiagnosticData()` command takes different arguments which can be supplied as key value pairs separated by comma. The different parameter names are describe in Table E-1.

Table E-1 exportDiagnosticData Command Syntax

Parameter	Description
`logicalName`	The logical name of the log file being read. If not specified, defaults to `ServerLog`. The different types of logs supported for off-line access are `HarvestedDataArchive`, `EventsDataArchive`, `ServerLog`, `DomainLog`, `HTTPAccessLog`, `WebAppLog`, `ConnectorLog` and `JMSMessageLog`.
`logName`	This parameter refers to the base log file name contains the log data to be exported. If not specified, defaults to `myserver.log`.
`storeDir`	The location of the diagnostic store for the server. If not specified, defaults to `../data/store/diagnostics`.
`logRotationDir`	The directory containing the old rotated log files. If not specified, defaults to "."
`query`	An expression specifying the filter condition for the data records to be included in the result set. If not specified defaults to an empty string which will return all the data.
`beginTime`	The timestamp (inclusive) of the earliest record to be added to the result set. If not specified defaults to 0.
`endTime`	he timestamp (exclusive) of the latest record to be added to the result set. If not specified defaults to `Long.MAX_VALUE`.
`exportFileName`	The name of the file to which the data is exported, if not specified by the user, defaults to `export.xml`.

Export Command Example

The following command exports all the data from the server log named `server1.log` and its rotated files from the current directory to an export file called `foo.xml`.

```
exportDiagnosticData(logicalName="ServerLog",logName="server.log",
  exportFileName="foo.xml")
```

WebLogic Scripting Tool Examples

This section covers the following topics:

Dynamically Creating DyeInjection Monitors Example

This demonstration script (see Listing F-1) shows how to use the `weblogic.WLST` tool to create a DyeInjection Monitor dynamically. This script does the following:

- Connects to a server and boots the server first if necessary.
- Looks up/create a WLDF System Resource.
- Creates the DyeInjection monitor.
- Sets the dye criteria.
- Enables the monitor.
- Saves and activates the configuration.
- Enables the diagnostic context functionality via the ServerDiagnosticConfigMBean

This demonstration script only configures the dye monitor, which injects dye values into the diagnostic context. To trigger events, you must implement downstream diagnostic monitors that use dye filtering to trigger on the specified dye criteria. An example downstream monitor artifact is below. This must be placed in a file named `weblogic-diagnostics.xml` and placed into the `META-INF` directory of a application archive. It is also possible to create a monitor using a JSR 88 deployment plan. For more information, see the related documentation.

Listing F-1 Example: Using WLST to Dynamically Create DyeInjection Monitors (demoDyeMonitorCreate.py)

```
# Script name: demoDyeMonitorCreate.py

###########################################################################
# Demo script showing how to create a DyeInjectionMonitor dynamically
# via WLST. This script will:
# - connect to a server, booting it first if necessary
# - lookup/create a WLDF System Resource
# - create the DyeInjection Monitor (DIM)
# - set the dye criteria
# - enable the monitor
# - save and activate
# - enable the Diagnostic context functionality via the
#   ServerDiagnosticConfig MBean

# Note: This will only configure the dye monitor, which will inject dye
# values into the Diagnostic Context.  To trigger events requires the
# existence of "downstream" monitors set to trigger on the specified
# dye criteria.

#
# An example downstream monitor artifact is below.  This must be
# placed in a file named "weblogic-diagnostics.xml" and placed
# into the "META-INF" directory of a application archive.  It is
# also possible to create a monitor using a JSR 88 deployment
# plan, see the related documentation for details.

# <wldf-resource xmlns="java:weblogic.diagnostics.descriptor">
#   <instrumentation>
#     <enabled>true</enabled>
#     <!-- Servlet Session Monitors -->
#     <wldf-instrumentation-monitor>
```

```
#        <name>Servlet_Before_Session</name>
#        <monitor-type>delegating</monitor-type>
#        <enabled>true</enabled>
#        <dye-mask>USER1</dye-mask>
#        <dye-filtering-enabled>true</dye-filtering-enabled>
#        <action>TraceAction</action>
#        <action>StackDumpAction</action>
#        <action>DisplayArgumentsAction</action>
#        <action>ThreadDumpAction</action>
#      </wldf-instrumentation-monitor>

#      <wldf-instrumentation-monitor>
#        <name>Servlet_After_Session</name>
#        <monitor-type>delegating</monitor-type>
#        <enabled>true</enabled>
#        <dye-mask>USER2</dye-mask>
#        <dye-filtering-enabled>true</dye-filtering-enabled>
#        <action>TraceAction</action>
#        <action>StackDumpAction</action>
#        <action>DisplayArgumentsAction</action>
#        <action>ThreadDumpAction</action>
#      </wldf-instrumentation-monitor>

#   </instrumentation>
# </wldf-resource>

###############################################################################
domainDirectory="wldfDomain"
url="t3://localhost:7001"
user="weblogic"
serverName="wldfDemo"
domain="wldfDomain"
props="weblogic.GenerateDefaultConfig=true,weblogic.RootDirectory="+domai
   Directory

try:
  connect(user,user,url)
except:

  startServer(adminServerName=serverName,domainName=domain,
     username=user,password=user,systemProperties=props,
```

```
      domainDir=domainDirectory,block="true")
  connect(user,user,url)

# Start an edit session
edit()
startEdit()
cd ("/")

# Lookup or create the WLDF System resource.
wldfResourceName = "mywldf"
# Create an instance of the harvester service helper.

mywldf = cmo.lookupSystemResource(wldfResourceName)
if mywldf==None:
  print "Unable to find named resource,
             creating WLDF System Resource: " + wldfResourceName
  mywldf=cmo.createWLDFSystemResource(wldfResourceName)

# Target the System Resource to the demo server.
wldfServer=cmo.lookupServer(serverName)
mywldf.addTarget(wldfServer)

# create and set properties of the DyeInjection Monitor (DIM).
mywldfResource=mywldf.getWLDFResource()
mywldfInst=mywldfResource.getInstrumentation()
mywldfInst.setEnabled(1)
monitor=mywldfInst.createWLDFInstrumentationMonitor("DyeInjection")
monitor.setMonitorType("standard")
monitor.setEnabled(1)

# Need to include newlines when setting properties on the DIM.
monitor.setProperties("\nUSER1=larry@celtics.com\
                       nUSER2=brady@patriots.com\n")
monitor.setDyeFilteringEnabled(1)

# Enable the diagnostic context functionality via the
# ServerDiagnosticConfig.
cd("/Servers/"+serverName+"/ServerDiagnosticConfig/"+serverName)
cmo.setDiagnosticContextEnabled(1)

# save and disconnect
save()
activate()
```

```
disconnect()
exit()
```

Watch and JMXNotification Example

This demonstration script (see Listing F-2) shows how to use the `weblogic.WLST` tool to configure a Watch and a JMXNotification using the WebLogic Diagnostic Service Watches and Notification component. This script does the following:

- Connects to a server and boots the server first if necessary.
- Looks up/creates a WebLogic Diagnostic Framework (WLDF) System Resource.
- Creates a watch and watch rule on the ServerRuntimeMBean for the `OpenSocketConnections` attribute.
- Configures the watch to use a JMXNotification medium.

This script can be used in conjunction with the following files and scripts:

- The `JMXWatchNotificationListener.java` class (see "JMXWatchNotificationListener CLass Example" on page F-8).
- The `demoHarvester.py` script, which registers the `OpenSocketConnections` attribute with the harvester for collection (see "MBrean Registration and Data Collection Example" on page F-11).

To see these files work together, perform the following steps:

1. To run the watch configuration script (`demoWatch.py`), type:

   ```
   java weblogic.WLST demoWatch.py
   ```

2. To compile the `JMXWatchNotificationListener.java` source and launch it, type:

   ```
   javac JMXWatchNotificationListener.java
   ```

3. To run the `demoHarvester.py script`, type:

   ```
   java weblogic.WLST demoHarvester.py
   ```

When the `demoHarvester.py` script runs, it triggers the JMXNotification for the watch configured in step 1.

Listing F-2 Example: Watch and JMXNotification (demoWatch.py)

```
# Script name: demoWatch.py
################################################################################
# Demo script showing how to configure a Watch and a JMXNotification
# using the WLDF Watches and Notification framework.
# The script will:
# - connect to a server, booting it first if necessary
# - lookup/create a WLDF System Resource
# - Create a watch and watch rule on the ServerRuntimeMBean for the
#     "OpenSocketConnections" attribute
# - Configure the watch to use a JMXNotification medium

# This script can be used in conjunction with
# - the JMXWatchNotificationListener.java class
# - the demoHarvester.py script, which registers the "OpenSocketConnections"
#   attribute with the harvester for collection.

# To see these work together:
# 1. Run the watch configuration script
#       java weblogic.WLST demoWatch.py
# 2. compile the JMXWatchNotificationListener.java source and launch it
#       javac JMXWatchNotificationListener.java

# 3. run the demoHarvester.py script
#       java weblogic.WLST demoHarvester.py
# When the demoHarvester.py script runs, it ends up triggering the
# JMXNotification for the watch configured in step 1.
################################################################################
domainDirectory="wldfDomain"
url="t3://localhost:7001"
user="weblogic"
serverName="wldfDemo"
domain="wldfDomain"
props="weblogic.GenerateDefaultConfig=true,
      weblogic.RootDirectory="+domainDirectory

try:
  connect(user,user,url)
except:
```

```
  startServer(adminServerName=serverName,domainName=domain,
     username=user,password=user,systemProperties=props,
     domainDir=domainDirectory,block="true")
  connect(user,user,url)

edit()
startEdit()

# Lookup or create the WLDF System resource
wldfResourceName = "mywldf"
# create an instance of the harvester service helper
mywldf = cmo.lookupSystemResource(wldfResourceName)
if mywldf==None:
  print "Unable to find named resource,
|       creating WLDF System Resource: " + wldfResourceName
  mywldf=cmo.createWLDFSystemResource(wldfResourceName)

# Target the System Resource to the demo server
wldfServer=cmo.lookupServer(serverName)
mywldf.addTarget(wldfServer)

cd("/WLDFSystemResources/mywldf/WLDFResource/mywldf/
     WatchNotification/mywldf")
watch=cmo.createWatch("mywatch")
watch.setEnabled(1)
jmxnot=cmo.createJMXNotification("myjmx")
watch.addNotification(jmxnot)

serverRuntime()
cd("/")
on=cmo.getObjectName().getCanonicalName()=
watch.setRuleExpression("${"+on+"} > 1")
watch.getRuleExpression()
watch.setRuleExpression("${"+on+"//OpenSocketsCurrentCount} > 1")
watch.setAlarmResetPeriod(10000)

edit()
save()
activate()
disconnect()
exit()
```

JMXWatchNotificationListener CLass Example

Listing F-3 shows how to write a JMXWatchNotificationListener.

Listing F-3 Example: JMXWatchNotificationListener Class (JMXWatchNotificationListener.java)

```
# Class name: JMXWatchNotificationListener.java
#####################################################################

import javax.management.*;
import weblogic.diagnostics.watch.*;

import javax.management.*;
import javax.management.remote.JMXServiceURL;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXConnector;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.naming.Context;
import java.util.Hashtable;
import java.io.IOException;
import weblogic.management.mbeanservers.runtime.RuntimeServiceMBean;

public class JMXWatchNotificationListener implements NotificationListener,
Runnable {
  private MBeanServerConnection rmbs = null;
  private String notifName = "myjmx";
  private int notifCount = 0;

  public JMXWatchNotificationListener() {
  }

  public void register() throws Exception {
    rmbs = getRuntimeMBeanServerConnection();
    addNotificationHandler(notifName, this);
  }

  public void handleNotification(Notification notif, Object handback) {
    synchronized (this) {
```

```
    try {
      WatchNotification wNotif = (WatchNotification)notif;
      notifCount++;
      System.out.println("Listener for notification:
                  " + notifName + " called. Count= " + notifCount + ".");
      System.out.println("Watch severity:
                  " + wNotif.getWatchSeverityLevel());
      System.out.println("Watch time:  " + wNotif.getWatchTime());
      System.out.println("Watch ServerName:
                  " + wNotif.getWatchServerName());
      System.out.println("Watch RuleType:
                 " + wNotif.getWatchRuleType());
     System.out.println("Watch Rule:          " + wNotif.getWatchRule());
     System.out.println("Watch Name:          " + wNotif.getWatchName());
     System.out.println("Watch DomainName:
                 " + wNotif.getWatchDomainName());
     System.out.println("Watch AlarmType:
       " + wNotif.getWatchAlarmType());
     System.out.println("Watch AlarmResetPeriod:
                  " + wNotif.getWatchAlarmResetPeriod());
     } catch (Throwable x) {
       System.out.println("Exception occurred processing JMX watch
              notification: " + notifName +"\n" + x);
       x.printStackTrace();
     }
   }
}
private void addNotificationHandler(String name,
                     NotificationListener list) throws Exception {
   /*
   * The JMX Watch notification listener registers with a Runtime MBean that
   * matches the name of the corresponding watch bean.  Each watch has it's
    * own Runtime MBean instance.
    */
   ObjectName oname =
     new ObjectName("wldfDomain:ServerRuntime=myserver,Name=" + name +
                   ",Type=WLDFWatchJMXNotificationRuntime," +
```

```
                    "WLDFWatchNotificationRuntime=WatchNotification,"+
                    "WLDFRuntime=WLDFRuntime");
  System.out.println("Adding notification handler for:
          " + oname.getCanonicalName());
  rmbs.addNotificationListener(oname, list, null, null);
}
private void removeNotificationHandler(String name, NotificationListener
     list) throws Exception {
  ObjectName oname =
    new ObjectName("wldfDomain:ServerRuntime=myserver,Name=" + name +
                    ",Type=WLDFWatchJMXNotificationRuntime," +
                    "WLDFWatchNotificationRuntime=WatchNotification,"+
                    "WLDFRuntime=WLDFRuntime");
  System.out.println("Removing notification handler for:
                       " +oname.getCanonicalName());
  rmbs.removeNotificationListener(oname, list);
}

public void run() {
  try {
    System.out.println("VM shutdown,
                 unregistering notification listener");
    removeNotificationHandler(notifName, this);
  } catch (Throwable t) {
    System.out.println("Caught exception in shutdown hook");
    t.printStackTrace();
  }
}

private String user = "weblogic";
private String password = "weblogic";

public MBeanServerConnection getRuntimeMBeanServerConnection()
  throws Exception {

  String JNDI = "/jndi/";
  JMXServiceURL serviceURL;
  serviceURL =
    new JMXServiceURL("t3", "localhost", 7001,
```

```
                        JNDI + RuntimeServiceMBean.MBEANSERVER_JNDI_NAME);
    System.out.println("URL=" + serviceURL);

    Hashtable h = new Hashtable();
    h.put(Context.SECURITY_PRINCIPAL,user);
    h.put(Context.SECURITY_CREDENTIALS,password);
    h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
          "weblogic.management.remote");
    JMXConnector connector = JMXConnectorFactory.connect(serviceURL,h);
    return connector.getMBeanServerConnection();
  }

  public static void  main(String[] args) {
    try {
      JMXWatchNotificationListener listener =
          new JMXWatchNotificationListener();
      System.out.println("Adding shutdown hook");
      Runtime.getRuntime().addShutdownHook(new Thread(listener));
      listener.register();
      // Sleep waiting for notifications
      Thread.sleep(600000);
    } catch (Throwable e) {
      e.printStackTrace();
    } // end of try-catch

  } // end of main()
}
```


MBrean Registration and Data Collection Example

This demonstration script shows how to use the `weblogic.WLST` tool to register MBeans and attributes for collection by the WebLogic Diagnostic Service Harvester. This script does the following:

- Connects to a server and boots the server first if necessary.
- Looks up/creates a WebLogic Diagnostic Framework (WLDF) System Resource.
- Sets the sampling frequency.

- Adds a type for collection.
- Adds an attribute of a specific instance for collection.
- Saves and activates the configuration.
- Displays a few cycles of the harvested data.

Listing F-4 Example: MBean Registration and Data Collection (demoHarvester.py)

```
# Script name: demoHarvester.py

###########################################################################
# Demo script showing how register MBeans and attributes for collection
# by the WLDF Harvester Service.  This script will:
# - connect to a server, booting it first if necessary
# - lookup/create a WLDF System Resource
# - set the sampling frequency
# - add a type for collection
# - add an attribute of a specific instance for collection
# - save and activate
# - display a few cycles of the harvested data#

###########################################################################

from java.util import Date
from java.text import SimpleDateFormat
from java.lang import Long
import jarray

###########################################################################
# Helper functions for adding types/attributes to the harvester configuration
###########################################################################

def findHarvestedType(harvester, typeName):
  htypes=harvester.getHarvestedTypes()
  for ht in (htypes):
    if ht.getName() == typeName:
      return ht
  return None

def addType(harvester, mbeanInstance):
  typeName = "weblogic.management.runtime." + mbeanInstance.getType() + "MBean"
  ht=findHarvestedType(harvester, typeName)
  if ht == None:
    print "Adding " + typeName + " to harvestables collection for " +
                         harvester.getName()
```

```
    ht=harvester.createHarvestedType(typeName)
  return ht;

def addAttributeToHarvestedType(harvestedType, targetAttribute):
  currentAttributes = PyList()
  currentAttributes.extend(harvestedType.getHarvestedAttributes());
  print "Current attributes: " + str(currentAttributes)

  try:
    currentAttributes.index(targetAttribute)
    print "Attribute is already in set"
    return
  except ValueError:
    print targetAttribute + " not in list, adding"

  currentAttributes.append(targetAttribute)
  newSet = jarray.array(currentAttributes, java.lang.String)
 print "New attributes for type " + harvestedType.getName() + ": " + str(newSet)

  harvestedType.setHarvestedAttributes(newSet)
  return

def addAttribute(harvester, mbeanInstance, attributeName):
  typeName = mbeanInstance.getType() + "MBean"
  ht = addType(harvester, mbeanInstance)
  addAttributeToHarvestedType(ht,attributeName)
  return

################################################################################
# Helper functions for querying for harvested data and displaying
# the formatted data records.
################################################################################

def printDataRecordWithColumns(record, colinfo):
  datePattern = "MM/dd/yyyy HH:mm:ss.SSS"
  formatter = SimpleDateFormat(datePattern)
  values = record.getValues()
  for col in range(0,len(values),1):
    val = values[col]
    colname = colinfo[col].getColumnName()
    print "  [" + str(col) + "] " + colname + ": " + str(val)

def printItWithColumns(it, columns):
  recno = 0
  while(it.hasNext()):
    record = it.next()
    recno += 1;
    print "======================================================"
    print "Record " + str(recno) + ":"
    printDataRecordWithColumns(record, columns)
```

```
    print "====================================================="
    print ""
  print "TotalRecords: " + str(recno)

def queryAccessor(acsr, begin=0, end=Long.MAX_VALUE, queryString=""):
  it=acsr.retrieveDataRecords(begin, end, queryString)
  printItWithColumns(it, acsr.getColumns())

################################################################################
# Display the currently registered types for the specified harvester
################################################################################

def displayHarvestedTypes(harvester):
  harvestedTypes = harvester.getHarvestedTypes()
  print ""
  print "Harvested types:"
  print ""
  for ht in (harvestedTypes):
    print "Type: " + ht.getName()
    attributes = ht.getHarvestedAttributes()
    if attributes != None:
      print "  Attributes: " + str(attributes)
    instances = ht.getHarvestedInstances()
    print "  Instances:  " + str(instances)
    print ""
  return

################################################################################
# Main script flow -- create a WLDF System resource and add harvestables
################################################################################

domainDirectory="wldfDomain"
url="t3://localhost:7001"
user="weblogic"
serverName="wldfDemo"
domain="wldfDomain"
props="weblogic.GenerateDefaultConfig=true,weblogic.RootDirectory=
           "+domainDirectory

try:
  connect(user,user,url)
except:
  startServer(adminServerName=serverName,domainName=domain,
     username=user,password=user,systemProperties=props,
     domainDir=domainDirectory,block="true")
  connect(user,user,url)

# start an edit session
edit()
```

```
startEdit()
cd("/")

# Lookup or create the WLDF System resource
wldfResourceName = "mywldf"
# create an instance of the harvester service helper
systemResource = cmo.lookupSystemResource(wldfResourceName)
if systemResource==None:
  print "Unable to find named resource,
        creating WLDF System Resource: " + wldfResourceName
  systemResource=cmo.createWLDFSystemResource(wldfResourceName)

# Obtain the harvester bean instance for configuration
print "Getting WLDF Resource Bean from " + str(wldfResourceName)
wldfResource = systemResource.getWLDFResource()
print "Getting Harvester Configuration Bean from " + wldfResourceName
harvester = wldfResource.getHarvester()
print "Harvester: " + harvester.getName()

# Target the WLDF System Resource to the demo server
wldfServer=cmo.lookupServer(serverName)
systemResource.addTarget(wldfServer)

# the harvester Jython wrapper maintains refs to
# the SystemResource objects
harvester.setSamplePeriod(5000)

# add a RT MBean attribute
serverRuntime()
cd("/")
addAttribute(harvester, cmo, "OpenSocketsCurrentCount")

# add a RT MBean type,all instances and attributes
cd("ExecuteQueueRuntimes/weblogic.kernel.Default")
addType(harvester,cmo)

# have to return to the edit tree to activate
edit()
# activate changes
activate()

# display the data
displayHarvestedTypes(harvester)

serverRuntime()
cd('/WLDFRuntime/WLDFRuntime/WLDFAccessRuntime/Accessor')
accessor=cmo.lookupWLDFDataAccessRuntime('HarvestedDataArchive')
cd('/')
```

```
time=java.util.Date().getTime()
Thread.sleep(10000)
queryAccessor(accessor,begin=time)

time=java.util.Date().getTime()
Thread.sleep(10000)
queryAccessor(accessor,begin=time)

disconnect()
exit()
```

Index

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

V

W