BEA WebLogic Server®

Configuring and Using the WebLogic Diagnostic Framework

Version 9.0
Revised: July 22, 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

3. Configuring and Capturing Diagnostic Images

4. Configuring Diagnostic Archives

5. Configuring the Harvester for Metric Collection

6. Configuring Watches and Notifications

7. Configuring Watches

8. Configuring Notifications

9. Configuring Instrumentation

10.Configuring the Diagnostic Context

11.Accessing Diagnostic Data Using the Data Accessor

12.Introduction to Programming WLDF

A. WLDF Query Language

CHAPTER 1

Introduction and Roadmap

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

What Is the WebLogic Diagnostic Framework?

The WebLogic Diagnostic Framework (WLDF) is a monitoring and diagnostic framework that defines and implements a set of services that run within the BEA WebLogic Server® process and participate in the standard server life cycle. Using WLDF, you can 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 run-time performance of servers and applications and enables you to isolate and diagnose faults when they occur.

WLDF includes several components for collecting and analyzing data, including the following:

- Diagnostic Image Capture—creates a diagnostic snapshot from the server that can be used for post-failure analysis
- Archiver—captures and persists all data events, log records, and metrics from server instances and applications

- Instrumentation—adds code to WebLogic server instances and the applications running on them to execute diagnostic actions at specified locations in the code. The Instrumentation component provides the means for creating and tracking diagnostic context by uniquely identifying requests and tracking them as they flow through the system.
- Harvester—captures metrics from run-time MBeans, including WebLogic Server MBeans and custom MBeans
- Watches and Notifications—provides the means for monitoring server and application states and sending notifications based on criteria set in the watches
- Logging services—manages logs for monitoring server, subsystem, and application events. The WebLogic Server logging services are documented separately from the rest of the WebLogic Diagnostic Framework. See *Configuring Log Files and Filtering Log Messages*.

WLDF 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. Independent Software Vendors (ISVs) can use these APIs to develop custom monitoring and diagnostic tools for integration with WLDF.

WLDF is a new feature in WebLogic 9.0. In previous releases of WebLogic Server, access to diagnostic data by monitoring agents—which were developed by customers or third-party tool developers—was limited to JMX attributes, and changes to monitoring agents required server shut down and restart. However, WLDF 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.

Document Scope and Audience

This document tells how to configure and use the monitoring and diagnostic services provided by WLDF.

WLDF provides features for monitoring and diagnosing problems in running WebLogic server instances and clusters and in applications deployed to them. Therefore, the information in this document is directed both to system administrators and to application developers. It also contains information for third-party tool developers who want to build tools to support and extend WLDF.

It is assumed that readers are familiar with Web technologies and the operating system and platform where 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, “Understanding WLDF Configuration,” provides an overview of how WLDF features are configured for servers and applications.
- Chapter 3, “Configuring and Capturing Diagnostic Images,” tells how to configure and use the WLDF Diagnostic Image Capture component to capture a snapshot of significant server configuration settings and state.
- Chapter 4, “Configuring Diagnostic Archives,” tells how to configure and use the WLDF Diagnostic Archive component to persist diagnostic data to a file store or database.
- Chapter 5, “Configuring the Harvester for Metric Collection,” tells how to configure and use the WLDF Harvester component to harvest metrics from runtime MBeans, including WebLogic Server MBeans and custom MBeans.
- Chapter 6, “Configuring Watches and Notifications,” provides an overview of WLDF watches and notifications.
- Chapter 7, “Configuring Watches,” tells how to configure watches to monitor server instances and applications for specified conditions and then send notifications when those conditions are met.
- Chapter 8, “Configuring Notifications,” tells how to configure notifications that can be triggered by watches.
- Chapter 9, “Configuring Instrumentation,” tells how to add diagnostic instrumentation code to WebLogic Server classes and to the classes of applications running on the server.
- Chapter 10, “Configuring the Diagnostic Context,” tells how to use the `DyeInjection` monitor and how to us dye filtering with diagnostic monitors.
- Chapter 11, “Accessing Diagnostic Data Using the Data Accessor,” tells how to configure and use the WLDF Data Accessor component to retrieve diagnostic data.
- Chapter 12, “Introduction to Programming WLDF,” provides an overview of how you can use the JMX API and the WebLogic Scripting Tool (`weblogic.WLST`) to configure and use WLDF components.
- Appendix A, “WLDF Query Language,” describes the WLDF query language that is used for constructing expressions to query diagnostic data using the Data Accessor, construct watch rules, and construct rules for filtering logs.

- Appendix B, "WLDF Instrumentation Library," describes the predefined diagnostic monitors and diagnostic actions that are included in the WLDF Instrumentation Library.
- Appendix C, "WebLogic Scripting Tool Examples," provides examples of how to perform WLDF monitoring and diagnostic activities using the WebLogic Scripting Tool.
- Appendix D, "Terminology," is a glossary of terms used in WLDF.

Related Documentation

- *Understanding the WebLogic Diagnostic Framework* describes the architecture of WLDF.
- *Configuring Log Files and Filtering Log Messages* describes how to use WLDF logging services to monitor server, subsystem, and application events.
- "Configure the WebLogic Diagnostic Framework" in the *Administration Console Online Help* tells how to use the visual tools in the WebLogic Administration Console to configure WLDF.
- The WLDF system resource descriptor conforms to the `diagnostics.xsd` schema, available at http://www.bea.com/ns/weblogic/90/diagnostics.xsd. See *WebLogic Server Diagnostics Configuration Schema Reference* for documentation.

CHAPTER 2

Understanding WLDF Configuration

The WebLogic Diagnostic Framework (WLDF) provides features for generating, gathering, analyzing, and persisting diagnostic data from BEA WebLogic Server® instances and from applications deployed to server instances. For server-scoped diagnostics, some WLDF features are configured as part of the configuration for the domain. Other features are configured as system resource descriptors that can be targeted to servers (or clusters). For application-scoped diagnostics, diagnostic features are configured as resource descriptors for the application.

The following topics provide an overview of WLDF configuration:

For general information about WebLogic Server domain configuration, see *Understanding Domain Configuration*.

About Configuration MBeans and XML

As in other WebLogic Server subsystems, WLDF is configured using configuration MBeans (Managed Beans), and the configuration is persisted in XML configuration files The configuration MBeans are instantiated at startup, based on the configuration settings in the XML file. When a configuration is changed by changing the values of MBean attributes, those changes are saved (persisted) in the XML files.

One characteristic of the relationship between MBeans and the XML files is that configuration MBean attributes map directly to configuration XML elements. For example, the `Enable` attribute of the `WLDFInstrumentationMBean` maps directly to the `<enabled>` sub-element of `<instrumentation>` in the resource descriptor file (configuration file) for a diagnostic module. If you change the value of the MBean attribute, the content of the XML element is changed when the configuration is saved. Conversely, if you directly edit an XML element in the configuration file, the value of the MBean attribute is updated when the MBean is instantiated.

For more information about WLDF Configuration MBeans, see "About the Configuration MBean Hierarchy and Mappings to XML Elements" on page 2-9. For general information about how MBeans are implemented and used in WebLogic Server, see "Understanding WebLogic Server MBeans" in *Developing Custom Management Utilities with JMX*.

Tools for Configuring WLDF

As with other WebLogic Server subsystems, there are several ways to configure WLDF:

- Use the visual tools in the Administration Console to configure WLDF for server instances and clusters. See "Configure the WebLogic Diagnostic Framework" in the *Administration Console Online Help*.
- Write scripts to be run in the WebLogic Scripting Tool (WLST). For specific information about using WLST with WLDF, see Appendix C, "WebLogic Scripting Tool Examples." Also see *WebLogic Scripting Tool* for general information about using WLST.
- Configure WLDF programmatically using JMX and the WLDF configuration MBeans. See "Introduction to Programming WLDF" for specific information about programming WLDF. See *WebLogic Server MBean Reference* and browse or search for specific MBeans for programming reference.
- Edit the XML configuration files directly. Whenever possible, this documentation explains configuration tasks in terms of the XML used in the configuration files. The XML is easy to understand, and you can edit the configuration files directly, although it is recommended

that you first generate the XML files by configuring WLDF in the Administration Console. Doing so provides a blueprint for valid XML.

How WLDF Configuration Is Partitioned

WLDF can be used to perform diagnostics tasks for server instances (and clusters) and for applications.

Server-Level Configuration

The following WLDF components are configured as part of the configuration for a server instance in a domain. The configuration is persisted in the domain's `config.xml` file.

- Diagnostic Image Capture
- Diagnostic Archives

See "Configuring Diagnostic Image Capture and Diagnostic Archives" on page 2-4

The following WLDF components are configured as diagnostic modules, or resources, that can be deployed to server instances. These configuration settings are persisted in the diagnostic resource descriptor files (configuration files) that can be targeted to one or more server instances.

- Harvester (for collecting metrics)
- Watch and Notification
- Instrumentation

See "Configuring Diagnostic System Modules" on page 2-4.

Note: If you have configured diagnostic context for a server instance (which is configured as part of the instrumentation for a diagnostic module), you globally enable or disable that module's context in the domain's `config.xml` file, not in the configuration for the module itself.

Application-Level Configuration

The WLDF Instrumentation component can be used with applications. The Instrumentation component is configured in a resource descriptor file deployed with the application in the applications archive file. See "Configuring Diagnostic Modules for Applications" on page 2-8.

Configuring Diagnostic Image Capture and Diagnostic Archives

The Diagnostic Image Capture component and the Diagnostic Archive component are configured in a domain's `config.xml` file. They are configured in the `<server-diagnostic-config>` element, which is a child of the a `<server>` element in a domain, as shown in Listing 2-1.

Listing 2-1 Sample WLDF Configuration Information in the Config.xml File for a Domain

```
<domain>
  <server>
    <name>myserver</name>
    <server-diagnostic-config>
      <image-dir>logs\diagnostic_images</image-dir>
      <image-timeout>3</image-timeout>
      <diagnostic-store-dir>data/store/diagnostics</diagnostic-store-dir>
      <diagnostic-data-archive-type>FileStoreArchive
      </diagnostic-data-archive-type>
    </server-diagnostic-config>
  </server>
  <!-- Other server elements to configure other servers in this domain -->
  <!-- Other domain-based configuration elements, including references to
       WLDF system resources, or diagnostic system modules.
       See Listing 2-2. -->
</domain>
```

For more information, see the following:

- Chapter 3, "Configuring and Capturing Diagnostic Images"
- Chapter 4, "Configuring Diagnostic Archives"

Configuring Diagnostic System Modules

To configure and use the Instrumentation, Harvester, and Watch and Notification components at the server level, you must first create a system resource called a *diagnostic system module*.

System modules are globally available for targeting to servers and clusters configured in a domain.

About the Diagnostic System Module and Its Resource Descriptor

A diagnostic system module is created as a `WLDFResourceBean`, and the configuration is persisted in a resource descriptor file (configuration file), called *`DIAG_MODULE`*`.xml`, where *`DIAG_MODULE`* is the name of the diagnostic module. You can specify a name for the descriptor file, but it is not required. If you do not provide a file name, a file name is generated based on the value in the descriptor file's `<name>` element. The file is created by default in the *`DOMAIN_NAME`*`\config\diagnostics` directory, where *`DOMAIN_NAME`* is the name of the domain's home directory. The file has the extension `.xml`.

Note: The diagnostic module conforms to the `diagnostics.xsd` schema, available at http://www.bea.com/ns/weblogic/90/diagnostics.xsd. See *WebLogic Server Diagnostics Configuration Schema Reference* for documentation.

Referencing the Diagnostics System Module from Config.xml

When you create a diagnostic system module using the Administration Console or the WebLogic Scripting Tool (WLST), WebLogic Server creates it in the *`DOMAIN_NAME`*`/config/diagnostics` directory, and a reference to the module is added to the domain's `config.xml` file.

Note: Even if you are writing your own XML configuration files, you may want to create a diagnostic module from the Console. That way, you can start with the valid XML that the Console creates. For information see "Create diagnostic system modules" in the *Administration Console Online Help*.

The `config.xml` file can contain multiple references to diagnostic modules, in one or more `<wldf-system-resource>` elements. The `<wldf-system-resource>` element includes the name of the diagnostic module file and the list of servers and clusters to which the module is targeted. The `<wldf-system-resource>` element can also include a `<diagnostic-context-enabled>` element to enable or disable diagnostic context for the target server instance(s) or cluster(s).

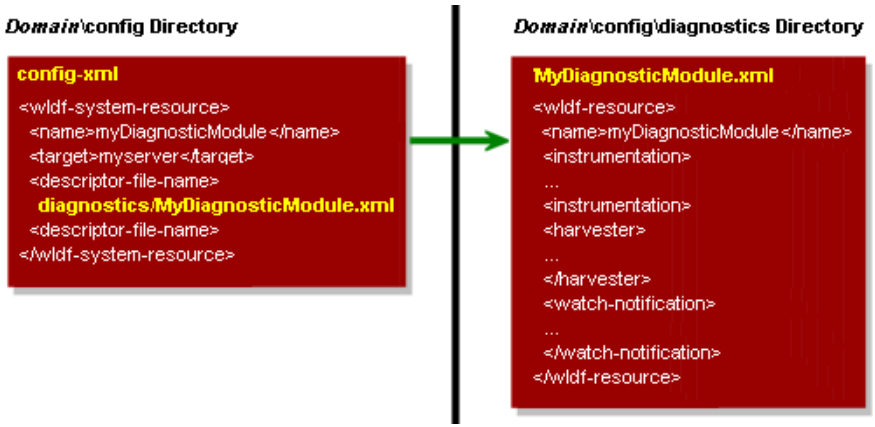
For example, Listing 2-2 shows a module named `myDiagnosticModule` targeted to the server `myserver` and another module named `newDiagnosticMod` targeted to servers `ManagedServer1` and `ManagedServer1`. The diagnostic context is enabled for the `newDiagnosticMod` module.

Listing 2-2 Sample WLDF Configuration Information in the Config.xml File for a Domain

```
<domain>
  <!-- Other domain-level configuration elements -->
  <wldf-system-resource
        xmlns="http://www.bea.com/ns/weblogic/90/diagnostics">
    <name>myDiagnosticModule</name>
    <target>myserver</target>
    <descriptor-file-name>diagnostics/MyDiagosticModule.xml
    </descriptor-file-name>
    <description>My diagnostic module</description>
  </wldf-system-resource>
  <wldf-system-resource>
    <name>newDiagnosticMod</name>
    <target>ManagedServer1,ManagedServer2</target>
    <descriptor-file-name>diagnostics/newDiagnosticMod.xml
    </descriptor-file-name>
    <diagnostic-context-enabled>true</diagnostic-context-enabled>
    <description>A diagnostic module for my managed servers</description>
  </wldf-system-resource>
<!-- Other WLDF system resource configurations -->
</domain>
```

The relationship of the `config.xml` file and the `MyDiagnosticModule.xml` file is shown in Figure 2-1.

Figure 2-1 Relationship of config.xml to System Descriptor File



About the Resource Descriptor Configuration

Except for the name and list of targets, which are listed in the `config.xml` file, as described above, all configuration for a diagnostic module is saved in its resource descriptor file. Listing 2-3 shows portions of the descriptor file for a diagnostic module named `myDiagnosticModule`.

Listing 2-3 Sample Diagnostics System Module Descriptor File, MyDiagnosticModule.xml

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>MyDiagnosticModule</name>

  <instrumentation>

    <!-- Configuration elements for one or more diagnostic monitors -->

  </instrumentation>

  <harvester>
    <!-- Configuration elements for harvesting metrics from one or more
         MBean types, instances, and attributes -->
  </harvester>
```

```
  <watch-notification>
    <!-- Configuration elements for one or more watches and one or more
         notifications-->
  </watch-notification>
</wldf-resource>
```

Managing Diagnostic Modules

A diagnostic system module can be targeted to zero, one, or more servers, although a server can have only one module targeted to it at a time. You can create multiple modules that monitor different aspects of your system. Then, you can choose which module to target to a server or cluster, based on what you want to monitor at that time.

Because you can target the same module to multiple servers or clusters, you can write general purpose modules that you want to use across a domain.

You can change the target of a diagnostic module without restarting the server instance(s) to which it is targeted or untargeted. That capability provides considerable flexibility in writing and using diagnostic monitors that address a specific diagnostic goal, without interfering with the operation of the server instances themselves.

More Information About Configuring Diagnostic System Resources

See the following sections for detailed instructions for configuring WLDF system resources:

- Chapter 5, “Configuring the Harvester for Metric Collection”
- Chapter 6, “Configuring Watches and Notifications”
- Chapter 9, “Configuring Instrumentation”
- Chapter 10, “Configuring the Diagnostic Context”

Configuring Diagnostic Modules for Applications

Application-scoped instrumentation is configured and deployed as a diagnostic module, which is similar to a diagnostic system module. However, an application module is configured in an XML

configuration file named `weblogic-diagnostics.xml` which is packaged with the application archive.

Notes: You can configure only the Instrumentation component in a diagnostic descriptor for an application.

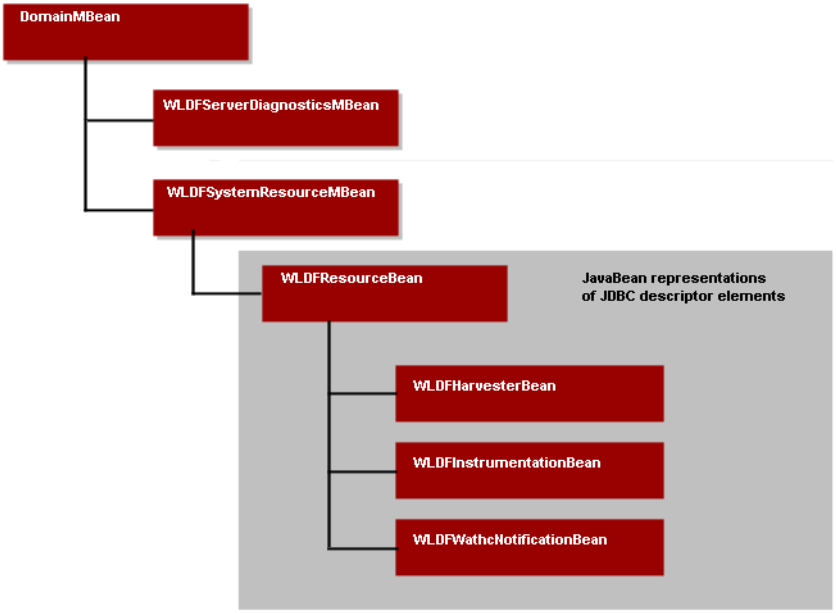
The `DyeInjection` monitor, which is used to configure diagnostic context, can be configured only at the server level. But once a diagnostic context is created, the context attached to incoming requests remain with the requests as they flow through the application.

For detailed instructions for configuring instrumentation for applications. see "Configuring Application-scoped Instrumentation."

About the Configuration MBean Hierarchy and Mappings to XML Elements

Figure 2-2 shows the hierarchy of the WLDF configuration MBeans and the diagnostic system module Beans for WLDF objects in a WebLogic domain.

Figure 2-2 WLDF Configuration Bean Tree



The following WLDF MBeans configure WLDF at the server level. They map to XML elements in the `config.xml` configuration file for a domain:

- `WLDFServerDiagnosticMBean` controls configuration settings for the Data Archive and Diagnostic Images components for a server. It also controls whether diagnostic context for a diagnostic module is globally enabled or disabled.

  This MBean is represented by a `<server-diagnostic-config>` element in the `config.xml` file for the domain.

- `WLDFSystemResourceMBean` contains the name of a descriptor file for a diagnostic module in the `config/diagnostics` directory and the name(s) of the target server(s) to which that module is deployed.

  This MBean is represented by a `<wldf-system-resource>` element in the `config.xml` file for the domain.

  Note: You can create multiple diagnostic system modules in a domain. The configurations for the modules are saved in multiple descriptor files in the `config/diagnostics`

directory for the domain. The domain's `config.xml` file, therefore, can contain the multiple `<wldf-system-resource>` elements that represent those modules. However, you can target only one diagnostic module to a server at a time. You cannot have two files in the `config/diagnostics` directory whose active target is the same server.

- `WLDFResourceBean` contains the configuration settings for a diagnostic module. This Bean is represented by a `<wldf-resource>` element in a *`ModuleName`*`.xml` diagnostics descriptor file in the domain's `config/diagnostics` directory. The `WLDFResourceBean` contains configuration settings for the following components:
  - Harvester: The `WLDFHarvesterBean` is represented by the `<harvester>` element in a *`ModuleName`*`.xml` file.
  - Instrumentation: The `WLDFInstrumentationBean` is represented by the `<instrumentation>` element in a *`ModuleName`*`.xml` file.
  - Watch and Notification: The `WLDFWatchNotificationBean` is represented by the `<watch-notification>` element in a *`ModuleName`*`.xml` file.

  For an application, if a `WLDFResourceBean` is linked from a `WLDFSystemResourceBean`, the settings for WLDF components apply to the targeted server and to any applications running on that server. If a `WLDFResourceBean` is contained within a `weblogic-diagnostics.xml` descriptor file which is deployed as part of an application archive, you can configure only the Instrumentation component, and the settings apply only to that application. In the latter case, the `WLDFResourceBean` is not a child of a `WLDFSystemResourceBean`.

CHAPTER 3

Configuring and Capturing Diagnostic Images

The Diagnostic Image Capture component of the WebLogic Diagnostic Framework (WLDF) can be used to create a diagnostic snapshot, or dump, of a server's internal runtime state at the time of the capture. This information can be particularly useful for support personnel analyzing what might have contributed to a server failure.

The following topics tell how to configure and use the Diagnostic Image Capture component:

How to Initiate Image Captures

A diagnostic image capture can be initiated by:

- A "first failure" detection. See "About First Failure Detection and Notification."
- A configured watch notification. See Chapter 8, "Configuring Notifications."
- A request initiated by a user in the Administration Console (and, potentially, requests initiated from third-party diagnostic tools). See "Configure and capture diagnostic images" in the *Administration Console Online Help*.

- A direct API call, using JMX.
- WLST command

About First Failure Detection and Notification

The Diagnostic Image Capture component includes a first failure notification feature. When a server makes the transition into a failed state, this feature automatically triggers an image creation notification. That notification triggers the creation and capture of a diagnostic image. The first-failure notification feature ensures that the state of a server when it failed is preserved.

You can configure an image lockout period for first failure notifications. The image lockout time limits the number of diagnostic images that are captured in a specified time period. 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.

Configuring Diagnostic Image Captures

Because the diagnostic image capture is meant primarily as a post-failure analysis tool, there is little control over what information is captured. Available configuration options are:

- The default destination for the image
- For a specific capture, a destination that is different from the default destination
- A lockout, or *timeout*, period, to control how often an image is taken during a sequence of server failures and recoveries

Diagnostic Image Capture is configured in the config.xml file for a domain, under the `<server>` element for the server, as shown in Listing 3-1:

Listing 3-1 Sample Diagnostic Image Capture Configuration

```
<domain>
  <!-- Other domain configuration elements -->
  <server>
    <name>myserver</name>
    <server-diagnostic-config>
      <image-dir>logs\diagnostic_images</image-dir>
```

```
      <image-timeout>2</image-timeout>
    </server-diagnostic-config>
    <!-- Other configuration details for this server -->
  </server>
  <!-- Other server configurations in this domain-->
</domain>
```

What Information Is Captured

The most common sources of a server state are captured in a diagnostic image capture, including:

- Configuration
- Log cache state
- Java Virtual Machine (JVM)
- Work manager state
- JNDI state
- Harvestable data.

The Diagnostic 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.

Note: A diagnostic image is a heavyweight artifact meant to serve as a server-level state dump for the purpose of diagnosing significant failures. It provides the capability to capture a significant amount of important data in a structured format and then to provide that data to support personnel for analysis.

About the Captured Image File

An image is captured as a single file for the entire server. Each image has a unique name, as follows:

```
diagnostic_image_domain_server_yyyy_MM_dd_HH_mm_ss
```

The contents of the file include at least the following information:

- 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 OS information, if available
- Command line arguments, if available
- WLS version including patch and build number information

CHAPTER 4

Configuring Diagnostic Archives

The Archiver component of the WebLogic Diagnostic Framework (WLDF) captures and persists all data events, log records, and metrics collected by WLDF from server instances and applications running on them. You can access archived diagnostic data in on-line mode (that is, on a running server), you can also access archived data in off-line mode using WLST.

You can configure WLDF to archive diagnostic data to a file store or a Java Database Connectivity (JDBC) data source, as described in the following sections:

Configuring the Archive

The diagnostic archive is configured on a per-server basis, in the `config.xml` file for a domain, under the `<server>` element for the server. Examples for file-based stores and JDBC-based stores are shown in Listing 4-1 and Listing 4-3.

Configuring a File-Based Store

WLDF creates text files containing the archived information. The only configuration option for a WLDF file-based archive is the directory where the file will be created and maintained. The default directory is the `data/store/diagnostics` directory under the home directory for the server.

When you save to a file-base store, WLDF uses the WebLogic Server persistent store. For more information, see "Using the WebLogic Persistent Store" in *Configuring WebLogic Server Environments*.

An example configuration for a file-based store is shown in Listing 4-1.

Listing 4-1 Sample configuration for File-based Diagnostic Archive

```
<domain>
  <!-- Other domain configuration elements -->
  <server>
    <name>myserver</name>
    <server-diagnostic-config>
      <diagnostic-store-dir>data/store/diagnostics</diagnostic-store-dir>
      <diagnostic-data-archive-type>FileStoreArchive
      </diagnostic-data-archive-type>
    </server-diagnostic-config>
  </server>
  <!-- Other server configurations in this domain -->
</domain>
```

Configuring a JDBC-Based Store

To use a JDBC store, the appropriate tables must exist in a database, and JDBC must be configured to connect to that database.

Creating WLDF Tables in the Database

If they don't already exist, you must create the database tables used by WLDF to store data in a JDBC-based store. Two tables are required:

- The `wls_events` table stores data generated from WLDF instrumentation events.
- The `wls_hvst` table stores data generated from the WLDF Harvester component.

The SQL Data Definition Language (DDL) used to create tables may differ for different databases, depending on the SQL variation supported by the database. The following code listing shows the DDL that can be used to create WLDF tables in the PointBase database.

Listing 4-2 DDL Definition of the WLDF Tables for PointBase Database

```
-- DDL for creating wls_events table for instrumentation events

DROP TABLE wls_events;
CREATE TABLE wls_events (
  RECORDID INTEGER IDENTITY,
  TIMESTAMP NUMERIC default NULL,
  CONTEXTID varchar(128) default NULL,
  TXID varchar(32) default NULL,
  USERID varchar(32) default NULL,
  TYPE varchar(64) default NULL,
  DOMAIN varchar(64) default NULL,
  SERVER varchar(64) default NULL,
  SCOPE varchar(64) default NULL,
  MODULE varchar(64) default NULL,
  MONITOR varchar(64) default NULL,
  FILENAME varchar(64) default NULL,
  LINENUM INTEGER default NULL,
  CLASSNAME varchar(250) default NULL,
  METHODNAME varchar(64) default NULL,
  METHODDSC varchar(4000) default NULL,
  ARGUMENTS clob(100000) default NULL,
  RETVAL varchar(4000) default NULL,
  PAYLOAD blob(100000),
  CTXPAYLOAD VARCHAR(4000),
  DYES NUMERIC default NULL
);


-- DDL for creating wls_events table for instrumentation events

DROP TABLE wls_hvst;
CREATE TABLE wls_hvst (
  RECORDID INTEGER IDENTITY,
  TIMESTAMP NUMERIC default NULL,
  DOMAIN varchar(64) default NULL,
  SERVER varchar(64) default NULL,
  TYPE varchar(64) default NULL,
```

```
  NAME varchar(250) default NULL,
  ATTRNAME varchar(64) default NULL,
  ATTRTYPE INTEGER default NULL,
  ATTRVALUE VARCHAR(4000)
);

COMMIT;
```

Consult the documentation for your database or your database administrator for specific instructions for creating these tables for your database.

Configuring JDBC Resources for Use By WLDF

Once the tables are created in your database, you must configure JDBC to access the tables in the database. For information about configuring JDBC resources in WebLogic Server, see *Configuring and Managing WebLogic JDBC*.

Once you have created the tables and configured the JDBC resource, you can specify that resource as the data source for the JDBC store to be used for an archive for a server.

An example configuration for a JDBC-based store is shown in Listing 4-3.

Listing 4-3 Sample configuration for JDBC-based Diagnostic Archive

```
<domain>
  <!-- Other domain configuration elements -->
  <server>
    <name>myserver</name>
    <server-diagnostic-config>
      <diagnostic-data-archive-type>JDBCArchive
      </diagnostic-data-archive-type>
      <diagnostic-jdbc-resource>JDBCArchive</diagnostic-jdbc-resource>
    <server-diagnostic-config>
  </server>
  <!-- Other server configurations in this domain -->
</domain>
```

CHAPTER 5

Configuring the Harvester for Metric Collection

The Harvester component of the WebLogic Diagnostic Framework (WLDF) gathers metrics from attributes on qualified MBeans that are instantiated in a running server. The Harvester can collect metrics from BEA WebLogic Server® MBeans and from custom MBeans.

This section includes:

About Harvesting, Harvestable Data, and Harvested Data

Harvesting metrics describes the process of gathering data that is useful for monitoring system state and measuring system performance. Metrics are exposed to WLDF as attributes on qualified MBeans. The harvester gathers values from selected MBean attributes at a specified sampling rate. Therefore, you can track potentially fluctuating values over time.

Data must meet certain requirements in order to be *harvestable*, and it must meet further requirements in order to be *harvested*:

- *Harvestable data* is data that can potentially be harvested from *harvestable entities*, including MBean types, instances, and attributes. To be harvestable, an MBean must be registered in the local WebLogic Server runtime MBean server.
- *Harvested data* is data that is currently being harvested. To be harvested, the data must meet all the following criteria:

- The data must be *harvestable.*
- The data must be configured to be harvested.
- For custom MBeans, the MBean must have been discovered.
- The data must not throw exceptions while being harvested.

The `WLDFHarvesterRuntimeMBean` provides the set of harvestable data and harvested data. The information returned by this MBean is a snapshot of a potentially changing state. For a description of the information about the data provided by the this MBean, see the description of the `weblogic.management.runtime.WLDFHarvesterRuntimeMBean` in the *WebLogic Server MBean Reference*.

You can use the Administration Console, the WebLogic Scripting Tool (`weblogic.WLST`), or JMX to configure the harvester to collect, analyze, and archive the metrics that the server MBeans and the custom MBeans contain.

Harvesting Data from the Different Harvestable Entities

You can configure the Harvester to gather data from named MBean types, instances, and attributes. In all cases, the Harvester collects the values of attributes of MBean instances, as explained in Table 5-1.

Table 5-1 Sources of Harvested Data from Different Configurations

When this entity is configured to be harvested...	Data is collected from...
A type	All attributes in all instances of the specified type
An attribute of a type	The specified attribute in all instances of the specified type
An instance of a type	All attributes in the specified instance of the specified type
An attribute of an instance of a type	The specified attribute in the specified instance of the specified type

All WebLogic Server runtime MBean types and attributes are known at startup. Therefore, when the harvester configuration is loaded, the set of harvestable WebLogic Server entities is the same as the set of WebLogic Server runtime MBean types and attributes. As types are instantiated, those instances also become harvestable.

The set of harvestable custom MBean types is dynamic. A custom MBean must be instantiated before its type can be known. (The type does not exist until at least one instance is created.) Therefore, as custom MBeans are registered with and removed from the MBean server, the set of custom harvestable types grows and shrinks. This process of detecting a new type based on the registration of a new MBean is called *type discovery*.

When you configure the harvester through the Administration Console, the Console provides a list of harvestable entities that can be configured. The list is always complete for WebLogic Server MBeans, but it must be discovered dynamically for custom MBeans.

Configuring the Harvester

The Harvester is configured and metrics are collected in the scope of a diagnostic module targeted to one or more server instances.

Listing 5-1 shows Harvester configuration elements in a WLDF system resource descriptor file, `myWLDF.xml`. This sample configuration harvests from the `ServerRuntimeMBean`, the `WLDFHarvesterRuntimeMBean`, and from a custom (non-WLS) MBean. The text following the listing explains each of the elements in the listing.

Listing 5-1 Sample Harvester Configuration

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<name>myWLDF</name>

  <harvester>

    <enabled>true</enabled>

    <sample-period>5000</sample-period>

    <harvested-type>
      <name>weblogic.management.runtime.ServerRuntimeMBean</name>
    </harvested-type>

    <harvested-type>
      <name>weblogic.management.runtime.WLDFHarvesterRuntimeMBean</name>
        <harvested-attribute>TotalSamplingTime</harvested-attribute>
        <harvested-attribute>CurrentSnapshotElapsedTime
```

```
        </harvested-attribute>
    </harvested-type>
    <harvested-type>
      <name>myMBeans.MySimpleStandard</name>
      <harvested-instance>myCustomDomain:Name=myCustomMBean1
      </harvested-instance>
      <harvested-instance>myCustomDomain:Name=myCustomMBean2
      </harvested-instance>
    </harvested-type>
  </harvester>
<!-- ----- Other elements ----- -->
</wldf-resource>
```

Configuring the Harvester Sampling Period

The <`sample-period`> element sets the sample period for the Harvester, in milliseconds. For example:

```
<sample-period>5000</sample-period>
```

The sample period specifies the time between each cycle. For example, if the Harvester begins execution at time *T*, and the sample period is *I*, then the next harvest cycle begins at *T*+*I*. If a cycle takes *A* seconds to complete and if *A* exceeds *I*, then the next cycle begins at *T*+*A*.

Configuring the Types of Data to Harvest

You configure the types of data to harvest in one or more `<harvested-type>` elements. Each `<harvested-type>` element specifies an MBean type from which metrics are to be collected. Optional sub-elements specify the instances and/or attributes to be collected for that type. Set these options as follows:

- The optional `<harvested-instance>` element specifies that metrics are to be collected only from the listed instances of the specified type. In general, an instance is specified by providing its JMX `ObjectName` in JMX canonical form.
- If no `<harvested-instance>` is present, all instances that are present at the time of each harvest cycle are collected.

- The optional `<harvested-attribute>` element specifies that metrics are to be collected only for the listed attributes of the specified type. 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`.
- If no `<harvested-attribute>` is present, all attributes defined for the type are collected.

Specifying Type Names for WebLogic Server MBeans vs. Custom MBeans

The Harvester supports WebLogic Server MBeans and custom MBeans. WebLogic Server MBeans are those which come packaged as part of the WebLogic Server. Custom MBeans can be harvested as long as they are registered in the local runtime MBean server.

There is a small difference in how WebLogic Server and customer types are specified. For WebLogic Server 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 (for example, see Listing 5-1, "Sample Harvester Configuration," on page 5-3).

When Configuration Settings are Validated

WLDF 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.

Example Configurations for Different Harvestable Types

`Listing 5-2` shows a `<harvested-type>` element that specifies that the `ServerRuntimeMBean` is to be harvested. Because no `<harvested-instance>` sub-element 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-elements present, all available attributes of the MBean are harvested.

Listing 5-2 Example Configuration for Collecting All Instances and All Attributes of a Type

```
<harvested-type>
  <name>weblogic.management.runtime.ServerRuntimeMBean</name>
</harvested-type>
```

Listing 5-3 shows a <harvested-type> element that specifies that the WLDFHarvesterRuntimeMBean is to be harvested. As above, because there is only one WLDFHarvesterRuntimeMBean, there is no need to provide a specific list of instances. The sub-element <harvested-attribute> specifies that only two of the available attributes of the WLDFHarvesterRuntimeMBean will be harvested: TotalSamplingTime and CurrentSnapshotElapsedTime.

Listing 5-3 Example Configuration for Collecting Specified Attributes of All Instances of a Type

```
<harvested-type>
  <name>weblogic.management.runtime.WLDFHarvesterRuntimeMBean</name>
    <harvested-attribute>TotalSamplingTime</harvested-attribute>
    <harvested-attribute>CurrentSnapshotElapsedTime
    </harvested-attribute>
</harvested-type>
```

Listing 5-4 shows a <harvested-type> element that specifies that a single instance of a custom MBean type is to be harvested. Because this is a custom MBean, the name is the implementation class. In this example, the two <harvested-instance> elements specify that only two instances of this type will be harvested. Each instance is specified using the canonical representation of its JMX ObjectName. Since no instances of<harvested-attribute> are specified, all attributes will be harvested.

Listing 5-4 Example Configuration for Collecting Specified Attributes of a Specified Instance of a Type

```
<harvested-type>
  <name>myMBeans.MySimpleStandard</name>
```

```
    <harvested-instance>myCustomDomain:Name=myCustomMBean1
    </harvested-instance>
    <harvested-instance>myCustomDomain:Name=myCustomMBean2
    </harvested-instance>
  </harvested-type>
```

CHAPTER 6

Configuring Watches and Notifications

The Watch and Notification component of the WebLogic Diagnostic Framework (WLDF) provides the means for monitoring server and application states and then sending notifications based on criteria set in the watches.Watches and notifications are configured as part of a diagnostic module targeted to one or more servers.

Watches and notifications are described in the following sections:

About Watches and Notifications

A *watch* identifies a situation that you want to trap for monitoring or diagnostic purposes. You can configure watches to analyze log records, data events, and harvested metrics. A watch is specified as a watch rule, which includes:

- A watch rule expression
- An alarm setting
- One or more notification handlers

A *notification* is an action that is taken when a watch rule expression evaluates to `true`. WLDF supports the following types of notifications:

- Java Management Extensions (JMX)
- Java Message Service (JMS)
- Simple Mail Transfer Protocol (SMTP), for example, e-mail
- Simple Network Management Protocol (SNMP)
- Diagnostic Images

A watch must be associated with a notification for a useful diagnostic activity to occur, for example, to notify an administrator about specified states or activities in a running server.

Watches and notifications are configured separately from each other. A notification can be associated with multiple watches, and a watch can be associated with multiple notifications. This provides the flexibility to recombine and re-use watches and notifications, according to current needs.

Overview of Watch and Notification Configuration

A complete watch and notification configuration includes settings for one or more watches, one or more notifications, and any underlying configurations required for the notification media, for example, the SNMP configuration required for an SNMP-based notification.

The main elements required for configuring watches and notifications in a WLDF system resource configuration file are shown in Listing 6-1. As the listing shows, the main element for defining watches and notifications is `<watch-notification>`. Watches are defined in `<watch>` elements, and notifications are defined in elements named for each of the types of notification, for example `<jms-notification>`, `<jmx-notification>`, `<smtp-notification>`, and `<image-notification>`.

Listing 6-1 A Skeleton Watch and Notification Configuration

```
<wldf-resource>
<!-- ----- Other system resource configuration elements ----- -->
  <watch-notification>
  <log-watch-severity>
     <!-- Severity for a log watch that triggers notifications -->
  </log-watch-severity>
```

```
<!-- ----- Watch configuration elements: ----- -->

<watch>
  <!-- A watch rule -->
</watch>

<watch>
  <!-- A watch rule -->
</watch>

<!-- Any other watch configurations -->


<!-- ----- Notification configuration elements: ----- -->

<!-- The following notification configuration elements show one of each
     type of supported notifications. However, not all types are
     required in any one system resource configuration, and multiples
     of any type are permitted. -->

<jms-notification>
  <!-- Configuration for a JMS-based notification; requires a
       corresponding JMS configuration via a jms-server element and a
       jms-system-resource element -->
</jms-notification>


<jmx-notification>
  <!-- Configuration for a JMX-based notification -->
</jmx-notification>

<smtp-notification>
  <!-- Configuration for an SMTP-based notification; requires a
       corresponding SMTP configuration via a mail-session element -->
</smtp-notification>

<snmp-notification>
  <!-- Configuration for an SNMP-based notification; requires a
       corresponding SNMP agent configuration via an snmp-agent
       element -->
</snmp-notification>
```

```
    <image-notification>
      <!-- Configuration for an image-based notification -->
    </image-notification>
  <watch-notification>
<!-- ----- Other configuration elements ----- -->
</wldf-resource>
```

Note: While the notification media must be configured so they can be used by the notifications that depend on them, those configurations are not part of the configuration of the diagnostic module itself. That is, they are not configured in the `<wldf-resource>` element in the diagnostic module's configuration file.

Each watch and notification can be individually enabled and disabled by setting `<enabled>true</enabled>` or `<enabled>false</enabled>` for the individual watch and/or notification. In addition, the entire watch and notification facility can be enabled and disabled by setting `<enabled>true</enabled>` or `<enabled>false</enabled>` for all watches and notifications. The default value is `<enabled>true</enabled>`.

The `<watch-notification>` element contains a `<log-watch-severity>` sub-element, which affects how notifications are triggered by log-rule watches. If the maximum severity level of the log messages that 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. Do not confuse this element with the `<severity>` element defined on watches. The `<severity>` element assigns a severity to the watch itself, whereas the `<log-watch-severity>` element controls which notifications are triggered by log-rule watches.

Sample Watch and Notification Configuration

A complete configuration for a set of watches and notifications in a diagnostic module is shown in Listing 6-2. The details of this example are explained in the following two sections:

- Chapter 7, "Configuring Watches"
- Chapter 8, "Configuring Notifications"

Listing 6-2 Sample Watch and Notification Configuration

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

<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">
  <name>mywldf1</name>

  <!-- Instrumentation must be configured and enabled for instrumentation
       watches -->

  <instrumentation>
    <enabled>true</enabled>
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <description>Dye Injection monitor</description>
      <dye-mask xsi:nil="true"></dye-mask>
      <properties>ADDR1=127.0.0.1</properties>
    </wldf-instrumentation-monitor>
  </instrumentation>

  <!-- Harvesting must be configured and enabled for harvester watches -->

  <harvester>
    <name>mywldf1</name>
    <sample-period>20000</sample-period>
    <harvested-type>
      <name>weblogic.management.runtime.ServerRuntimeMBean</name>
    </harvested-type>
    <harvested-type>
      <name>weblogic.management.runtime.WLDFHarvesterRuntimeMBean</name>
    </harvested-type>
  </harvester>

  <!-- All watches and notifications are defined under the
       watch-notification element -->

  <watch-notification>
    <enabled>true</enabled>
    <log-watch-severity>Info</log-watch-severity>

    <!-- A harvester watch configuration -->
```

```
    <watch>
      <name>myWatch</name>
      <enabled>true</enabled>
      <rule-type>Harvester</rule-type>
      <rule-expression>${com.bea:Name=myserver,Type=ServerRuntime//Sockets
OpenedTotalCount} &gt;= 1</rule-expression>
      <alarm-type>AutomaticReset</alarm-type>
      <alarm-reset-period>60000</alarm-reset-period>
      <notification>myMailNotif,myJMXNotif,mySNMPNotif</notification>
    </watch>

    <!-- An instrumentation watch configuration -->

    <watch>
      <name>myWatch2</name>
      <enabled>true</enabled>
      <rule-type>EventData</rule-type>
      <rule-expression>
      (MONITOR LIKE 'JDBC_After_Execute') AND
      (DOMAIN = 'MedRecDomain') AND
      (SERVER = 'medrec-adminServer') AND
      ((TYPE = 'ThreadDumpAction') OR (TYPE = TraceElapsedTimeAction')) AND
      (SCOPE = 'MedRecEAR')
      </rule-expression>
      <notification>JMXNotifInstr</notification>
    </watch>

    <!-- A log watch configuration -->

    <watch>
      <name>myLogWatch</name>
      <rule-type>Log</rule-type>
      <rule-expression>MSGID='BEA-000360'</rule-expression>
      <severity>Info</severity>
      <notification>myMailNotif2</notification>
    </watch>

    <!-- A JMX notification -->
```

```
    <jmx-notification>
      <name>myJMXNotif</name>
    </jmx-notification>

    <!-- Two SMTP notifications -->

    <smtp-notification>
      <name>myMailNotif</name>
      <enabled>true</enabled>
      <mail-session-jndi-name>myMailSession</mail-session-jndi-name>
      <subject>This is a harvester alert</subject>
      <recipient>username@emailservice.com</recipient>
    </smtp-notification>

    <smtp-notification>
      <name>myMailNotif2</name>
      <enabled>true</enabled>
      <mail-session-jndi-name>myMailSession</mail-session-jndi-name>
      <subject>This is a log alert</subject>
      <recipient>username@emailservice.com</recipient>
    </smtp-notification>

    <!-- An SNMP notification -->

    <snmp-notification>
      <name>mySNMPNotif</name>
      <enabled>true</enabled>
    </snmp-notification>

  </watch-notification>

</wldf-resource>
```

CHAPTER 7

Configuring Watches

The following topics describe the types of watches and their configuration options:

About the Types of Watches

WLDF provides three main types of watches, based on what the watch can monitor:

- **Harvester** watches monitor the set of harvested MBeans in the local runtime MBean server.
- **Log** watches monitor the set of messages generated into the server log.
- **Instrumentation** (or Event Data) watches monitor the set of events generated by the WLDF Instrumentation component.

In the WLDF system resource configuration file for a diagnostic module, each type of watch is defined in a `<rule-type>` element, which is a child of `<watch>`. For example:

```
<watch>
```

```
    <rule-type>Harvester</rule-type>
    <!-- Other configuration elements -->
  </watch>
```

Watches with different rule types differ in two ways:

- The rule syntax for specifying the conditions being monitored are unique to the type.
- Log and event watches are triggered in real time, whereas harvester watches are triggered only after the current harvest cycle completes.

Configuration Options Shared by all Types of Watches

All watches share certain configuration options:

- Watch rule expression

  In the diagnostic module configuration file, watch rule expressions are defined in `<rule-expression>` elements.

  A watch rule expressions is a logical expression that specifies what significant events the watch is to trap. For information about the query language you use to define watch rules, including the syntax available for each type of watch rule, see Appendix A, "WLDF Query Language."

- Notifications associated with the watch

  In the diagnostic module configuration file, notifications are defined in `<notification>` elements.

  Each watch can be associated with one or more notifications that are triggered whenever the watch evaluates to `true`. The content of this element is a comma-separated list of notifications. For information about configuring notifications, see Chapter 8, "Configuring Notifications."

- Alarm options

  In the diagnostic module configuration file, alarm options are set using `<alarm-type>` and `<alarm-reset-period>` elements.

  Watches can be specified to trigger repeatedly, or to trigger once, when a condition is met. For watches that trigger repeatedly, you can optionally define a minimum time between occurrences. The `<alarm-type>` element defines 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 in

the <alarm-reset-period>. A value of ManualReset causes the watch to fire a single time. After it fires, you must manually reset it to fire again. For example, you can use the WatchNotification runtime MBean to reset a manual watch. The default for <alarm-type> is None.

- Severity options

  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-element <severity>. The default is Notice.

- Enabled options

  Each watch can be individually enabled and disabled, using the sub-element <enabled>. When disabled, 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 (that is, the value of this flag on a specific watch is ignored).

Configuring Harvester Watches

A harvester watch can monitor any runtime MBean in the local runtime MBean server.

Note: If you define a watch rule to monitor an MBean (or MBean attributes) that the Harvester is not configured to harvest, the watch *will* work. The Harvester will "implicitly" harvest values to satisfy the requirements set in the defined watch rules. However, data harvested in this way (that is, implicitly for a watch) will not be archived. See Chapter 5, "Configuring the Harvester for Metric Collection," for more information about the Harvester.

Harvester watches are triggered in response to a harvest cycle. So, for harvester watches, the Harvester sample period defines a 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.

Listing 7-1, shows an example of a harvester watch that monitors several runtime MBeans.When the rule evaluates to true, six different notifications are sent: a JMX notification, an SMTP notification, an SNMP notification, an image notification, and JMS notifications for both a topic and a queue.

Note: In this example of a harvester watch, the Harvester is explicitly configured for this watched value. Therefore, Listing 7-1 below contains the appropriate harvester configuration elements.

The watch rule is a logical expression composed of four 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.

Note: The comparison operators are qualified in order to be valid in XML.

Listing 7-1 Harvester Watch Example

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>mywldf1</name>

  <harvester>
    <harvested-type>
      <name>myMBeans.MySimpleStandard</name>
      <harvested-instance>myCustomDomain:Name=myCustomMBean1
      </harvested-instance>
      <harvested-instance>myCustomDomain:Name=myCustomMBean2
      </harvested-instance>
    </harvested-type>
    <!-- Other Harvester configuration elements -->
  <harvester>

  <watch-notification>

    <watch>
      <name>simpleWebLogicMBeanWatchRepeatingAfterWait</name>
      <enabled>true</enabled>
      <rule-type>Harvester</rule-type>
      <rule-expression>
       (${mydomain:Name=WLDFHarvesterRuntime,ServerRuntime=myserver,Type=
       WLDFHarvesterRuntime,WLDFRuntime=WLDFRuntime//TotalSamplingTime}
          >= 100
           AND
          ${mydomain:Name=myserver,Type=
```

```
                    ServerRuntime//OpenSocketsCurrentCount} > 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>
      <!-- Other watch-notification configuration elements -->
    </watch-notification>
</wldf-resource>
```

This watch uses an alarm type of `AutomaticReset`, which means that it may be triggered repeatedly provided that the last time it was triggered was longer than the interval set as the 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.

Configuring Log Watches

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

An example configuration for a log watch is shown in Listing 7-2.

Listing 7-2 Example Configuration for a Log Watch

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">
  <name>mywldf1</name>
  <watch-notification>
    <enabled>true</enabled>
    <log-watch-severity>Info</log-watch-severity>
    <watch>
      <name>myLogWatch</name>
      <rule-type>Log</rule-type>
      <rule-expression>MSGID='BEA-000360'</rule-expression>
      <severity>Info</severity>
      <notification>myMailNotif2</notification>
    </watch>
    <smtp-notification>
      <name>myMailNotif2</name>
      <enabled>true</enabled>
      <mail-session-jndi-name>myMailSession</mail-session-jndi-name>
      <subject>This is a log alert</subject>
      <recipient>username@emailservice.com</recipient>
    </smtp-notification>
  </watch-notification>
</wldf-resource>
```

Configuring Instrumentation Watches

Instrumentation watches are used to monitor the events from the WLDF Instrumentation component. Watches of this type trigger as soon as the event is posted.

Listing 7-3 shows an example configuration for an instrumentation watch.

Listing 7-3 Example Configuration for an Instrumentation Watch

```
  <watch-notification>
    <watch>
    <name>myInstWatch</name>
    <enabled>true</enabled>
    <rule-type>EventData</rule-type>
    <rule-expression>
      (PAYLOAD &gt; 100000000) AND (MONITOR = 'Servlet_Around_Service')
    </rule-expression>
    <alarm-type xsi:nil="true"></alarm-type>
    <notification>mySMTPNotification</notification>
  </watch>

  <smtp-notification>
    <name>mySMTPNotification</name>
    <enabled>true</enabled>
    <mail-session-jndi-name>myMailSession</mail-session-jndi-name>
    <subject xsi:nil="true"></subject>
    <body xsi:nil="true"></body>
    <recipient>username@emailservice.com</recipient>
  </smtp-notification>

</watch-notification>
```

Defining Watch Rule Expressions

A watch rule expression encapsulates all information necessary for specifying a rule. For documentation on the query language you use to define watch rules, see Appendix A, "WLDF Query Language."

CHAPTER 8

Configuring Notifications

The following topics describe the types of notifications and their configuration options:

About the Types of Notifications

A *notification* is an action that is triggered when a watch rule evaluates to `true`. WLDF supports four types of diagnostic notifications, based on the delivery mechanism: Java Management Extensions (JMX), Java Message Service (JMS), Simple Mail Transfer Protocol (SMTP), and Simple Network Management Protocol (SNMP). You can also create a notification that generates a diagnostic image.

In the descriptor configuration file for a diagnostic module, the different types of notifications are identified by these elements:

- `<jmx-notification>`
- `<jms-notification>`
- `<snmp-notification>`

- <smtp-notification>
- <image-notification>

These notification types all have <name> and <enabled> configuration options. The value of <name> is used as the value in a <notification> element for a watch, to map the watch to its corresponding notification(s). The <enabled> element, when set to true, enables that notification. In other words, the notification is fired when an associated watch evaluates to true. Other than <name> and <enabled>, each notification type is unique.

Note: Notifications are defined programmatically using weblogic.diagnostics.watch.WatchNotification

Configuring JMX Notifications

For each defined JMX notification, WLDF issues JMX events (notifications) whenever an associated watch is triggered. Applications can register a notification listener with the server's WLDFWatchJMXNotificationRuntimeMBeans to receive all notifications and filter the provided output.

Listing 8-1 shows an example of a JMX notification configuration.

Listing 8-1 Example Configuration for a JMX Notification

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>mywldf1</name>

  <watch-notification>
    <!-- One or more watch configurations -->

    <jmx-notification>
      <name>myJMXNotif</name>
      <enabled>true</enabled>
    </jmx-notification>

    <!-- Other notification configurations -->

  </watch-notification>
</wldf-resource>
```

An example of a JMX notification is as follows:

```
Notification name:    myjmx called. Count= 42.
Watch severity:         Notice
Watch time:             Jul 19, 2005 3:40:38 PM EDT
Watch ServerName:       myserver
Watch RuleType:         Harvester
Watch Rule:
 ${com.bea:Name=myserver,Type=ServerRuntime//OpenSocketsCurrentCount} > 1
Watch Name:             mywatch
Watch DomainName:       mydomain
Watch AlarmType:        None
Watch AlarmResetPeriod: 10000
```

Configuring JMS Notifications

JMS notifications are used to post messages to JMS topics and/or queues in response to the triggering of an associated watch. In the system resource configuration file, the elements `<destination-jndi-name>` and `<connection-factory-jndi-name>` define how the message is to be delivered.

Listing 8-2 shows two JMS notifications that 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 the `config.xml` configuration file for the domain.

Listing 8-2 Example JMS Notifications

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>mywldf1</name>

  <watch-notification>
    <!-- One or more watch configurations -->

    <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>

      <!-- Other notification configurations -->

    </watch-notification>
</wldf-resource>
```

The content of the notification message gives details of the watch and notification.

Configuring SNMP Notifications

SNMP notifications are used to post SNMP traps in response to the triggering of an associated watch. To define an SNMP notification you only have to provide a notification name, as shown in Listing 8-3. Generated traps contain the names of both the watch and notification that caused the trap to be generated. For an SNMP trap to work properly, SNMP must be properly configured in the `config.xml` configuration file for the domain.

Listing 8-3 An Example Configuration for an SNMP Notification

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>mywldf1</name>

  <watch-notification>
    <!-- One or more watch configurations -->

    <snmp-notification>
      <name>mySNMPNotif</name>
    </snmp-notification>

    <!-- Other notification configurations -->

  </watch-notification>
```

```
</wldf-resource>
```

The trap resulting from the SNMP notification configuration shown in Listing 8-3 is of type `85`. It contains the following values (configured values are shown 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)
```

Configuring SMTP Notifications

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

Listing 8-4 shows an SMTP notification that causes an SMTP (e-mail) 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 the `config.xml` configuration file for the domain.

In this notification configuration, a custom subject and body are provided. If a subject and/or a body are not specified, defaults are provided, showing details of the watch and notification.

Listing 8-4 Example Configuration for SMTP Notification

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>mywldf1</name>

  <watch-notification>
    <!-- One or more watch configurations -->

    <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 ASAP.
           Reference number 81767366662AG-USA23.</body>
      <recipients>administrator@myCompany.com</recipients>
    </smtp-notification>

    <!-- Other notification configurations -->
  </watch-notification>
</wldf-resource>
```

The content of the notification message gives details of the watch and notification.

Configuring Image Notifications

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 in the configuration file using the sub-element `<image-directory>`.

Listing 8-5 shows an image notification that causes an image file of a standard name to be generated into the provided directory.

Listing 8-5 Example Configuration for Image Notification

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <name>mywldf1</name>

  <watch-notification>
    <!-- One or more watch configurations -->

    <image-notification>
      <name>myImageNotif</name>
      <enabled>true</enabled>
      <image-lockout>2</image-lockout>
      <image-directory>images</image-directory>
    </image-notification>

    <!-- Other notification configurations -->
  </watch-notification>
</wldf-resource>
```

The directory name specified is relative to the `servers` root directory for the domain. For example, in this case, image files will be generated into *`DOMAIN_NAME`*`\servers\`*`SERVER_NAME`*, where *`DOMAIN_NAME`* is the name of the domain's home directory and *`SERVER_NAME`* is the name of the server. `The default directory is` *`DOMAIN_NAME`*`\servers\`*`SERVER_NAME`*`\logs\diagnostic-images`.

You can specify a lockout, or *timeout*, period, to control how often an image is taken during a sequence of server failures and recoveries

Image file names are generated using the current timestamp (e.g. `diagnostic_image_myserver_2005_08_09_13_40_34.zip`), so this notification can fire many times, resulting in a separate image file each time.

For more information about Diagnostic Images, see Chapter 3, "Configuring and Capturing Diagnostic Images."

CHAPTER 9

Configuring Instrumentation

The Instrumentation component of the WebLogic Diagnostic Framework (WLDF) provides a mechanism for adding diagnostic code to BEA WebLogic Server® instances and the applications running on them. The key features provided by WLDF Instrumentation are:

- **Diagnostic monitors**. A *diagnostic monitor* is a dynamically manageable unit of diagnostic code which is inserted into a program at specific locations. Monitors are defined by scope (system or application) and type (standard, delegating, or custom).
- **Diagnostic actions**. A *diagnostic action* is the action a monitor takes when it is triggered during program execution.
- **Diagnostic context**. A *diagnostic context* is contextual information, such as the originating IP address, that identifies characteristics of requests that enter and flow through the system. The diagnostic context provides a means for tracking program execution and for controlling when monitors trigger their diagnostic actions. See Chapter 10, "Configuring the Diagnostic Context."

WLDF provides a library of predefined diagnostic monitors and actions. You can also create application-scoped custom monitors, where you control the locations where diagnostic code is inserted in the application.

Instrumentation is described in the following sections:

Concepts and Terminology

This section introduces instrumentation concepts and terminology.

Instrumentation Scope

You can provide instrumentation services at the system level (servers and clusters) and at the application level. Many concepts, services, configuration options, and implementation features are the same for both. However, there are differences, and they are discussed throughout this documentation. The term "server-scoped instrumentation" refers to instrumentation configuration and features specific to WebLogic Server instances and clusters. "Application-scoped instrumentation" refers to configuration and features specific to applications deployed on WebLogic servers. The scope is built in to each monitor; you cannot modify a monitor's scope.

Configuration and Deployment

Server-scoped instrumentation for a server or cluster is configured and deployed as part of a diagnostic module, an XML configuration file located in the *DOMAIN_NAME*`/config/diagnostics` directory, and linked from `config.xml`.

Application-scoped instrumentation is also configured and deployed as a diagnostics module, in this case an XML configuration file named `weblogic-diagnostics.xml` which is packaged with the application archive.

Joinpoints, Pointcuts, and Diagnostic Locations

Instrumenting code is inserted into server and application code at precise locations. The following terms are used to describe these locations:

- A *joinpoint* is a specific location in a class, for example a method or a call site within a method.
- A *pointcut* is an expression that specifies a set of joinpoints, for example all methods related to scheduling, starting and executing work items. The XML element used to describe a pointcut is `<pointcut>`. Pointcuts are described in "Defining Pointcuts for Custom Monitors" on page 9-17.
- A *diagnostic location* is a the position relative to a joinpoint where the diagnostic activity will take place. Diagnostic locations are *before*, *after*, and *around*. The XML element used to describe a diagnostic location is `<location-type>`.

Diagnostic Monitor Types

A diagnostic monitor is categorized by its scope and its type. The scope is either server-scoped or application-scoped. The type is determined by the monitor's pointcut, diagnostic location, and actions. For example, `Connector_After_Inbound` is an application-scoped delegating monitor, which can be used to trigger diagnostic actions at the exit of (that is, *after*) any method that handles inbound connections.

There are three types of instrumentation diagnostic monitors:

- A *standard monitor* performs specific, predefined diagnostic actions at specific, predefined pointcuts and locations. These actions, pointcuts, and locations are hardcoded in the monitor. You can enable or disable the monitor but you cannot modify its behavior.

  In the current release, the only standard monitor is the `DyeInjection` monitor, which is a server-scoped monitor used to configure diagnostic context and dye injection at the server level. For more information, see Chapter 10, "Configuring the Diagnostic Context."

- A *delegating monitor* has its scope, pointcuts, and locations hardcoded in the monitor, but you select the actions the monitor will perform. In that sense, the monitor delegates its actions to the ones you select. Delegating monitors are either server-scoped or application-scoped.

  A delegating monitor by itself is incomplete. In order for a delegating monitor to perform any useful work, you must assign at least one action to the monitor.

  Not all actions are compatible with all monitors. When you configure a delegating monitor from the Administration Console, you can choose only those actions that are appropriate for the selected monitor. If you are using WLST or editing a descriptor file by hand, you must make sure that the actions are compatible with the monitors. Validation is performed when the XML file is loaded at deployment time.

See Appendix B, "WLDF Instrumentation Library." for a list of the delegating monitors and actions provided by the WLDF Instrumentation Library.

- A *custom monitor* is a special case of a delegating monitor, which is available only for application-scoped instrumentation, and does not have a predefined pointcut or location.

  You assign a name to a custom monitor, define the pointcut and the diagnostics location the monitor will use, and then assign actions from the set of predefined diagnostic actions.

Table 9-1 summarizes the differences among the types of monitors.

Table 9-1 Diagnostic Monitor Types

Monitor Type	Scope	Pointcut	Location	Action
Standard monitor	Server	Fixed	Fixed	Fixed
Delegating monitor	Server or Application	Fixed	Fixed	Configurable
Custom monitor	Application	Configurable	Configurable	Configurable

Diagnostic Actions

Diagnostic actions execute diagnostic code that is appropriate for the associated delegating or custom monitor (standard monitors have predefined actions). In order for a delegating or custom monitor to perform any useful work, you must configure at least one action for the monitor.

The WLDF diagnostics library provides the following actions, which you can attach to a monitor by including the action's name in an `<action>` element when configuring the monitor:

- `DisplayArgumentsAction`
- `StackDumpAction`
- `ThreadDumpAction`
- `TraceAction`
- `TraceElapsedTimeAction`

Actions must be correctly matched with monitors. For example, the `TraceElapsedTime` action is compatible with a delegating or custom monitor whose diagnostic location type is `around`. See Appendix B, "WLDF Instrumentation Library." for more information.

You can restrict when a diagnostic action is triggered by setting a *dye mask* on a monitor. This mask determines which dye flags in the diagnostic context trigger actions. See "<wldf-instrumentation-monitor> XML Elements" on page 9-8 for information on setting a dye mask for a monitor.

Note: Diagnostic context, dye injection, and dye filtering are described in Chapter 10, "Configuring the Diagnostic Context."

Instrumentation Configuration Basics

This section provides information on the following topics:

About the Instrumentation Configuration Files

Instrumentation is configured as part of a diagnostics descriptor, an XML configuration file, whose name and location depend on whether you are implementing system-level (server-scoped) or application-level (application-scoped) instrumentation:

- System-level instrumentation configuration is stored in diagnostics descriptor(s) in the following directory:

  `DOMAIN_NAME/config/diagnostics`

  This directory can contain multiple diagnostic descriptor files. Filenames are arbitrary but must be terminated with `.xml` (`myDiag.xml` is a valid filename). Each file can contain configuration information for one or more of the deployable diagnostic components: Harvester, Instrumentation, or Watch and Notification. An `<instrumentation>` section in a descriptor file can configure one or more diagnostic monitors. Server-scoped instrumentation can be enabled, disabled, and in most cases reconfigured without restarting the server.

  Only one system-level diagnostics descriptor file can be active at a time for a server (or cluster). The active file is linked and targeted from the following configuration file:

  `DOMAIN_NAME/config/config.xml`

  See *Understanding Domain Configuration* for general information about the creation, content, and parsing of configuration files.

- Application-level instrumentation configuration is packaged within an application's archive in the following location:

`META-INF/weblogic-diagnostics.xml`

Because instrumentation is the only diagnostics component that is deployable to applications, this file can contain only instrumentation configuration information. Application-scoped instrumentation can be enabled and disabled without redeploying the application.You can also use JSR-88 deployment plans to enable or disable a monitor, or to modify the actions associated with a monitor.

Note that for instrumentation to be available for an application, instrumentation must be enabled on the server to which the application is deployed.

The diagnostics XML schema is located at:

`http://www.bea.com/ns/weblogic/90/diagnostics.xsd`

See *WebLogic Server Diagnostics Configuration Schema Reference* for documentation.

Each diagnostics descriptor file must begin with the following lines:

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

For an overview of WLDF resource configuration, see Chapter 2, "Understanding WLDF Configuration."

The XML Elements Used for Instrumentation

This section provides descriptor fragments and tables that summarize information about the XML elements used to configure instrumentation and the instrumentation diagnostic monitors.

- "<Instrumentation> XML Elements" on page 9-6 describes the top-level elements used within an `<instrumentation>` element.
- "<wldf-instrumentation-monitor> XML Elements" on page 9-8 describes the elements used within an `<wldf-instrumentation-monitor>` element.
- "Mapping <wldf-instrumentation-monitor> XML Elements to Monitor Types" on page 9-10 summarizes which instrumentation elements apply to which monitors.

<Instrumentation> XML Elements

Table 9-2 describes the `<instrumentation>` elements. The following configuration fragment illustrates the use of those elements:

```
<instrumentation>
   <enabled>true</enabled>
```

```
    <!-- The following <include> element would apply
         only to an appplication-scoped monitor -->
    <include>foo.bar.com.*</include>
</instrumentation>
```

Table 9-2 <instrumentation> XML Elements

Element	Description
<instrumentation>	The element that begins an instrumentation configuration.
<enabled>	If `true`, instrumentation is enabled. If `false`, no instrumented code will be inserted in classes in this instrumentation scope, and all diagnostic monitors within this scope are disabled. You must enable instrumentation at the server (or application) level to use instrumentation within that scope.
<include>	An optional element specifying the list of classes where instrumented code can be inserted. Wildcards (`*`) are supported. You can specify multiple `<include>` elements. Applies only to application-scoped instrumentation. Any specified <include> or <exclude> patterns are applied to the application scope as a whole. As classes are loaded, they must pass an include/exclude pattern check before any instrumentation code is inserted. Even if a class passes the include/exclude pattern checks, whether or not it is instrumented depends on the diagnostic monitors included in the configuration descriptor. An application-scoped delegating monitor from the library has its own predefined classes and pointcuts. A custom monitor specifies its own pointcut expression. Therefore a class can pass the include/exclude checks and still not be instrumented. A performance note: Instrumentation in inserted in applications at class load time. A large application that is loaded often may benefit from a judicious use of `<include>` and/or `<exclude>` elements. You can probably ignore these elements for small applications or for medium-to-large applications that are loaded infrequently.
<exclude>	An optional element specifying the list of classes where instrumented code cannot be inserted. Wildcards (`*`) are supported. You can specify multiple `<exclude>` elements. Applies only to application-scoped instrumentation. See the `<include>` description.

<wldf-instrumentation-monitor> XML Elements

Table 9-3 describes the `<wldf-instrumentation-monitor>` elements. The following configuration fragment illustrates the use of those elements. The fragment configures an application-scoped delegating monitor and a custom monitor. You could modify this fragment for server-scoped instrumentation by replacing the application-scoped monitors with server-scoped monitors.

```
<instrumentation>
   <enabled>true</enabled>
      <wldf-instrumentation-monitor>
       <name>Servlet_Before_Service</name>
       <enabled>true</enabled>
       <dye-mask>USER1</dye-mask>
       <dye-filtering-enabled>true</dye-filtering-enabled>
       <action>TraceAction</action>
     </wldf-instrumentation-monitor>
     <wldf-instrumentation-monitor>
       <name>MyCustomMonitor</name>
       <enabled>true</enabled>
       <action>TraceAction</action>
       <location-type>before</location-type>
          <pointcut>call( * com.foo.bar.* get*(...));</pointcut>
     </wldf-instrumentation-monitor>
</instrumentation>
```

Note that the `Servlet_Before_Service` monitor sets a dye mask and enables dye filtering. This will be useful only if instrumentation is enabled at the server level and the `DyeInjection` monitor is enabled and properly configured. See Chapter 10, "Configuring the Diagnostic Context," for information about configuring the `DyeInjection` monitor.

Table 9-3 <wldf-instrumentation-monitor> XML Elements

Element	Description
`<wldf-instrumentation-monitor>`	The element that begins a diagnostic monitor configuration.
`<enabled>`	If `true`, the monitor is enabled. If `false`, the monitor is disabled. You enable or disable each monitor separately

Table 9-3 <wldf-instrumentation-monitor> XML Elements

`<name>`	The name of the monitor. For standard and delegating monitors, use the names of the predefined monitors in Appendix B, "WLDF Instrumentation Library." For custom monitors, an arbitrary string that identifies the monitor.
`<description>`	An optional element describing the monitor.
`<action>`	An optional element, which applies to delegating and custom monitors. If you do not specify at least one action, the monitor will not generate any information. You can specify multiple `<action>` elements. An action must be compatible with the monitor type. For the list of predefined actions for use by delegating and custom monitors, see Appendix B, "WLDF Instrumentation Library."
`<dye-filtering-enabled>`	An optional element. If `true`, dye filtering is enabled. If `false`, dye-filtering is disabled. If dye filtering is not enabled at the server level, enabling dye filtering for a monitor for will have no effect.
`<dye-mask>`	An optional element. If dye filtering is enabled, the dye mask determines whether actions are taken. See Chapter 10, "Configuring the Diagnostic Context," for information about dyes and dye filtering.
`<properties>`	An optional element. Sets `name=value` pairs for dye flags. Applies only to the `DyeInjection` monitor, ignored by other monitors.
`<location-type>`	An optional element, whose value is one of `before`, `after`, or `around`. The location type determines when an action is triggered at a pointcut: before the pointcut, after the pointcut, or both before and after the pointcut. Applies only to custom monitors; standard and delegating monitors have predefined location types. A custom monitor must define a location type and a pointcut.
`<pointcut>`	An optional element. A pointcut element contains an expression that defines joinpoints where diagnostic code will be inserted. Applies only to custom monitors; standard and delegating monitors have predefined pointcuts. A custom monitor must define a location type and a pointcut.

Additional information on <dye-filtering-enabled> and <dye-mask> follows:

- When a DyeInjection monitor is enabled and configured for a server or a cluster, you can use dye filtering in downstream delegating and custom monitors to inspect the dyes injected into a request's diagnostic context by that DyeInjection monitor.
- The configuration of the DyeInjection monitor determines which bits are set in the 64-bit dye vector associated with a diagnostic context. When the <dye-filtering-enabled> attribute is enabled for a monitor, its diagnostic activity is suppressed if the dye vector in a request's diagnostic context does not match the monitor's configured dye mask. If the dye vector matches the dye mask (a bitwise AND), the application can execute its diagnostic actions:

```
(dye_vector & dye_mask == dye_mask)
```

Thus, the dye filtering mechanism, allows monitors to take diagnostic actions only for specific requests, without slowing down other requests. See Chapter 10, "Configuring the Diagnostic Context," for detailed information on diagnostic contexts and dye vectors.

Mapping <wldf-instrumentation-monitor> XML Elements to Monitor Types

Table 9-4 summarizes which <wldf-instrumentation-monitor> elements apply to which monitors.

Table 9-4 Mapping Instrumentation XML Elements to Monitor Types

Element	Standard	Delegating	Custom
<wldf-instrumentation-monitor>	X	X	X
<name>	X	X	X
<description>	X	X	X
<enabled>	X	X	X
<action>		X	X
<dye-filtering-enabled>		X	X
<dye-mask>		X	X
<properties>	X[1]		

Table 9-4 Mapping Instrumentation XML Elements to Monitor Types

Element	Standard	Delegating	Custom
<location-type>			X
<pointcut>			X

1. Used only by the `DyeInjection` monitor to set `name=value` pairs for dye flags.

Configuring Server-scoped Instrumentation

To enable instrumentation at the server level, and to configure server-scoped monitors, perform the following steps:

1. Decide how many diagnostic descriptor files you want to configure with instrumentation information.

   You can have multiple diagnostic descriptor files in a domain, but for each server (or cluster) you can deploy only one diagnostic descriptor file at a time. One reason for creating more than one file is to give yourself flexibility. You could have, for example, five diagnostic descriptor files in `the` *`DOMAIN_NAME`*`/config/diagnostics` directory. Each file contains a different instrumentation (and perhaps harvester and watch and notification) configuration. You then deploy a file to a server based on which monitors you want active for specific situations.

2. Decide which of the server-scoped monitors you want to include in a configuration:
   - If you plan to use dye filtering on a server, or on any deployed on that server, configure the `DyeInjection` monitor.
   - If you plan to use one or more of the server-scoped delegating monitors, decide which monitors to use and which actions to associate with each monitor.

3. Create the configuration file(s).
   - If you use the Administration Console to create the file (recommended), for delegating monitors the console displays only actions that are compatible with the monitor. If you create a configuration file with an editor or with the WebLogic Scripting Tool (WLST), you must correctly match actions to monitors.
   - See the "Domain Configuration Files" in *Understanding Domain Configuration* for information about configuring the config.xml file.

4. Validate and deploy the descriptor file. For server-scoped instrumentation, you can add and remove monitors and enable or disable monitors while the server is running.

Listing 9-1 contains a sample server-scoped instrumentation configuration file which enables instrumentation, and configures the `DyeInjection` standard monitor and the `Connector_Before_Work` delegating monitor. A single `<instrumentation>` element contains all instrumentation configuration for the module. Each diagnostic monitor is defined in a separate `<wldf-instrumentation-monitor>` element.

Listing 9-1 A Sample Server-scope Instrumentation Descriptor File

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <instrumentation>
    <enabled>true</enabled>
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <description>Inject USER1 and ADDR1 dyes</description>
      <enabled>true</enabled>
      <properties>USER1=weblogic
       ADDR1=127.0.0.1</properties>
    </wldf-instrumentation-monitor>
    <wldf-instrumentation-monitor>
      <name>Connector_Before_Work</name>
      <enabled>true</enabled>
      <action>TraceAction</action>
      <dye-filtering-enabled>true</dye-filtering-enabled>
      <dye-mask>USER1</dye-mask>
    </wldf-instrumentation-monitor>
  </instrumentation>
</wldf-resource>
```

Configuring Application-scoped Instrumentation

At the application level, WLDF instrumentation is configured as a deployable module, which is then deployed as part of the application.

The following sections provide information you need to configure application-scoped instrumentation:

Comparing System-scoped to Application-scoped Instrumentation

Instrumenting an application is similar to instrumenting at the system level, but with the following differences:

- Of the three types of instrumentation diagnostic monitors (standard, delegating, and custom), applications use delegating and custom monitors.
  - The only standard monitor, `DyeInjection`, is server-scoped.
  - Delegating monitors are either server-scoped or application-scoped. Applications must use the application-scoped delegating monitors.
  - All custom monitors are application-scoped.
- The server's instrumentation settings affect the application. In order to enable instrumentation for an application, instrumentation must be enabled for the server on which

the application is deployed. If instrumentation is not enabled on the server, enabling instrumentation in an application will have no effect.

- Application instrumentation is configured with a `weblogic-diagnostics.xml` descriptor file. You create a `META-INF/weblogic-diagnostics.xml` file, configure instrumentation, and put the file in the application's archive. When the archive is deployed, instrumentation is automatically inserted when the application is loaded.
- You can use a *deployment plan* to dynamically update the following configuration elements without redeploying the application:
  - `<enabled>`
  - `<dye-filtering-enabled>`
  - `<dye-mask>`
  - `<action>`
- You can also use a deployment plan to add diagnostic monitors (that is, to modify `<wldf-instrumentation-monitor>`). However, to remove a monitor from an application, you must redeploy the application. (You can dynamically disable a monitor, but the monitor remains embedded in the application's code until the application is redeployed.)

The XML descriptors for application-scoped instrumentation are defined in the same way as for server-scoped instrumentation. You can configure instrumentation for an application solely by using the delegating monitors and diagnostic actions available in the WLDF Instrumentation Library. You can create your own custom monitors; however, the diagnostic actions that you attach to these monitors must be taken from the Instrumentation Library.

Overview of the Steps Required to Instrument an Application

To implement a diagnostic monitor for an application, perform the following steps:

- Make sure that instrumentation is enabled on the server. See "Configuring Server-scoped Instrumentation" on page 9-11.
- Create a well formed `META-INF/weblogic-diagnostics.xml` descriptor file for the application:
  - Enable the `<instrumentation>` element: `<enabled>true</enabled>`
  - Add and enable at least one diagnostic monitor, with appropriate actions attached to it. (A monitor will generate diagnostic events only if the monitor is enabled and actions that generate events are attached to it.).

See "Creating a Descriptor File for a Delegating Monitor" on page 9-15 and "Creating a Descriptor File for a Custom Monitor" on page 9-16 for samples of well-formed descriptor files.

See "Defining Pointcuts for Custom Monitors" on page 9-17 for information on creating a pointcut expression.

- Put the descriptor file in the application archive.
- Deploy the application. See "Deploying an Application Diagnostic Descriptor" on page 9-19.

Creating a Descriptor File for a Delegating Monitor

The following is an example of a well-formed `META-INF/weblogic-diagnostics.xml` file for an application-scoped delegating monitor:

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

  <instrumentation>
    <enabled>true</enabled>
    <wldf-instrumentation-monitor>
      <name>Servlet_Before_Service</name>
      <enabled>true</enabled>
      <dye-mask>USER1</dye-mask>
      <dye-filtering-enabled>true</dye-filtering-enabled>
      <action>TraceAction</action>
    </wldf-instrumentation-monitor>
  </instrumentation>
</wldf-resource>
```

The `Servlet_Before_Service` monitor is an application-scoped monitor selected from the WLDF monitor library. It is hardcoded with a pointcut that sets joinpoints at method entry for several servlet/jsp methods. Because the application enables dye filtering and sets the `USER1` flag in its dye mask, the `TraceAction` action will be invoked only when the dye vector in the diagnostic context passed to the application also has its `USER1` flag set. (The dye vector is set at the system level via the `DyeInjection` monitor.) Therefore, the `Servlet_Before_Service` monitor in this application is essentially quiescent until it inspects a dye vector and finds the `USER1` flag set. This filtering reduces the amount of diagnostic data generated, and ensures that the generated data is of interest to the administrator.

Creating a Descriptor File for a Custom Monitor

The following is an example of a well-formed `META-INF/weblogic-diagnostics.xml` file for a custom monitor:

```
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.bea.com/ns/weblogic/90/diagnostics.xsd">

 <instrumentation>
     <enabled>true</enabled>
     <wldf-instrumentation-monitor>
       <name>MyCustomMonitor</name>
       <enabled>true</enabled>
       <action>TraceAction</action>
       <location-type>before</location-type>
       <pointcut>call( * com.foo.bar.* get* (...));</pointcut>
     </wldf-instrumentation-monitor>
  </instrumentation>
</wldf-resource>
```

The `<name>` for a custom monitor is an arbitrary string chosen by the developer. Because this monitor is custom, it has no predefined locations when actions should be invoked; the descriptor file must define the location type and pointcut expression. In this example, the `TraceActionc` action will be invoked before (`<location-type>before</location-type>`) any methods defined by the pointcut expression is invoked. The example's pointcut expression is parsed as follows (note the use of wildcards):

Pointcut Expression	Description
`call(` `* com.foo.bar.* get* (...)`**`)`**	**`call( )`**: Trigger any defined actions when the methods whose jointpoints are defined by the remainder of this pointcut expression are invoked.
`call(` **`*`** `com.foo.bar.* get* (...))`	**`*`**: Return value. The wildcard indicates that the methods can have any type of return value.
`call( *` **`com.foo.bar.*`** `get* (...))`	**`com.foo.bar.*`**: Methods from class `com.foo.bar` and its subpackages are eligible.

Pointcut Expression	Description
`call( * com.foo.bar.* get* (...))`	**`get*`**: Any methods whose name starts with the string `get` is eligible.
`call( * com.foo.bar.* get* (...))`	**`(...)`**: The ellipsis indicates that the methods can have any number of arguments.

Therefore, this pointcut expression matches all `get*()` methods in all classes in package `com.foo.bar` and its subpackages. The methods can return values of any type, including `void`, and can have any number of arguments of any type. Instrumentation code will be inserted before of these methods, and, just before of those methods are called, the `TraceAction` action will be invoked.

See "Defining Pointcuts for Custom Monitors" on page 9-17 for a description of the grammar used to define pointcuts.

Defining Pointcuts for Custom Monitors

Custom monitors provide more flexibility than delegating monitors because you create pointcut expressions to control where diagnostics actions are invoked. As with delegating monitors, you must select actions from the action library.

A joinpoint is specific, well-defined location in a program. A pointcut is an expression that specifies a set of joinpoints. This section describes how expressions for pointcuts are defined using a subset of the AspectJ pointcut syntax.

You can specify two types of pointcuts for custom monitors:

- *call*: Take an action when a method is invoked.
- *execution*: Take an action when a method is executed.

The informal grammar is as follows:

```
<pointcut>  ::=  <execution-pointcut> | <callsite-pointcut>
<execution-pointcut> ::= execution ( <access-type> <joinpoint-signature> )
<callsite-pointcut> ::= call ( <joinpoint-signature> )
<joinpoint-signature> ::= <method-signature>
<method-signature> ::= <return-type> <class-type>.<method-name>
  ( <parameter-list> )
<return-type> ::= <class-type> | <primitive-type>
```

```
<parameter-list> ::= <parameter-type> (, <parameter-type>) *
<parameter-type> ::= <class-type> | <primitive-type> | <elepsis>
<class-type> ::= (<use-class-heirarchy>) ?
  <class-or-interface-name-pattern>
<use-class-heirarchy> ::= '+'
<elepsis> ::= '...'
```

The following rules apply:

- Wildcards (**`*`**) can be used in class types and method names.
- An ellipsis (**`...`**) in the argument list signifies a variable number of arguments of any types beyond the argument.
- A **`+`** (plus sign) prefix to a class type identifies all subclasses, subinterfaces or concrete classes implementing the specified class/interface pattern.
- A pointcut expression specifies a pattern to identify matching joinpoints. An attempt to match a joinpoint against it will return a boolean, indicating a valid match (or not).
- Pointcut expressions can be combined with AND, OR and NOT boolean operators to build complex pointcut expression trees.

For example, the following pointcut matches method executions of all public initialized methods in all classes in package `com.foo.bar` and its subpackages. The initialized methods may return values of any type, including `void`, and may have any number of arguments of any types.

```
execution(public * com.foo.bar.* initialize(...))
```

The following pointcut matches the method calls (callsites) on all classes that directly or indirectly implement the `com.foo.bar.MyInterface` interface (or a subclass, if it happens to be a class). The method names must start with `get`, be public, and return an `int` value. The method must accept exactly one argument of type `java.lang.String`:

```
call(int +com.foo.bar.MyInterface get*(java.lang.String))
```

The following example shows how to use boolean operators to build a pointcut expression tree:

```
call(void com.foo.bar.* set*(java.lang.String)) OR
call( * com.foo.bar.* get*())
```

The following example illustrates how the previous expression tree would be rendered as a `<pointcut>` element in a configuration file:

```
<pointcut>call(void com.foo.bar.* set*(java.lang.String)) OR
call( * com.foo.bar.* get*())</pointcut>
```

Deploying an Application Diagnostic Descriptor

If an application is deployed with a well-formed `META-INF/weblogic-diagnostics.xml` diagnostics descriptor file in place, the Instrumentation component automatically inserts diagnostic instrumentation code into matching application classes when the classes are loaded.

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: If an application `ear` archive contains `war`, `rar` or `ejb` modules that have the `weblogic-diagnostics.xml` descriptors in their `META-INF` directory, those descriptors will be ignored.

You can use any of the standard WebLogic Server tools provided for controlling deployment, including the WebLogic Administrative Console or the WebLogic Scripting Tool (WLST).

Creating Deployment Plans using weblogic.PlanGenerator

You can use the `weblogic.PlanGenerator` tool to create an initial deployment plan, and interactively override specific properties of the `weblogic-diagnostics.xml` descriptor. For example, to create the plan:

```
java weblogic.PlanGenerator -root c:\exportapps\myApplication
```

The `PlanGenerator` tool inspects all J2EE deployment descriptors in the selected application, and creates a deployment plan with null variables for all relevant WebLogic Server deployment properties that configure external resources for the application.

For more information about creating and using deployment plans, see "Configuring Applications for Production Deployment" in *Deploying Applications to WebLogic Server*.

For more information about exporting an application's WebLogic Server deployment configuration to a custom deployment plan, including instructions for using `PlanGenerator`, see "Exporting an Application for Deployment to New Environments" in *Deploying Applications to WebLogic Server*.

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 WebLogic Scripting Tool (WLST). For example, the following WLST command 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.

Support for Dynamic Control of the Instrumentation Configuration

Dynamic control of instrumentation monitors within applications is provided with the Deployment Plan (JSR-88) mechanism. With deployment plans, you 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 that 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.

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')
```

CHAPTER 10

Configuring the Diagnostic Context

The WLDF Instrumentation component provides the means for uniquely identifying requests and tracking them as they flow through the system. You can configure WLDF to check for certain characteristics of every request that enters the system (such as the originating user or client address), attach a context to the request (defined by a unique ID and by flags that represent the characteristics of the request), and then trigger instrumentation events based on the context of the request. You can then use these instrumentation events to generate logs and trigger notifications.

Diagnostic context is available at both the system level and the application level, with some differences in how it is configured and used.

The process of configuring and using diagnostic context is described in detail throughout this chapter, which contains the following topics:

About the Contents, Life Cycle, and Configuration of a Diagnostic Context

A diagnostic context contains a unique *context ID* and a *dye vector*, which identifies characteristics of the context.

About Context Life Cycle and the Context ID

The diagnostic context for a request is created and initialized when the request enters the system, for example when a client makes an HTTP request. The context remains attached to the request, even as the request crosses thread boundaries and Java Virtual Machine (JVM) boundaries. The diagnostic context lives for the duration of the life cycle of the request.

Every diagnostic context is identified by an ID that is unique in the domain. Because the ID travels with the request, it is possible to track given requests as they flow through the system.

About Dyes, Dye Flags, and Dye Vectors

Contextual information travels with a request as a 64-bit dye vector, where each bit is a flag to identify the presence of a *dye*. Each dye represents one attribute of a request, for example an originating user, an originating client IP address, access protocol, etc.

When a dye flag for a given attribute is set, it indicates that the attribute is present. When the flag is not set, it indicates the attribute is not present.

For example, consider a configuration where the dye ADDR1 is configured to indicate that a request originated from IP address 127.0.0.0**.** The dye flag `ADDR2` is configured to indicate that a request originated from IP address 127.0.0.1. If a request from IP address 127.0.0.0 enters the system, the `ADDR1` dye flag in the dye vector for the request is set. The `ADDR2` dye flag remains unset.

Diagnostic and monitoring features that take advantage of the diagnostic context can examine the dye vector to determine if an attribute is present. In the example above, the administrator could configure a watch to examine every request that is dyed with `ADDR1`, that is, that originated from IP address 127.0.0.0.

The dye vector also contains a `THROTTLE` dye, which is used to set how often incoming requests are dyed. For more information about this special dye, see "About the THROTTLE Dye Flag.".

For a list of the available dyes and the attributes they represent, see "Dyes Supported by the DyeInjection Monitor." The process of configuring dye vectors and using them is discussed throughout the rest of this chapter.

Where Diagnostic Context is Configured

Diagnostic context is configured as part of a diagnostic module. The primary mechanism for configuring the diagnostic context is the `DyeInjection` monitor, which is a *standard* diagnostic monitor. The joinpoints where the `DyeInjection` monitor is woven into the code are those locations where a request can enter the system. The *diagnostic action* is to check every request against the `DyeInjection` monitor's configuration, then create and attach a context to the request, setting dye flags as appropriate. For information about diagnostic monitor types, pointcuts (which define the joinpoints), and diagnostic actions, see Chapter 9, "Configuring Instrumentation."

Overview of the Process

This overview describes the configuration and use of context in a server-scoped diagnostic module.

1. The administrator configures a diagnostic module to use the `DyeInjection` monitor.
2. The administrator enables instrumentation for the module.
3. The administrator configures the `DyeInjection` monitor by assigning values to dyes, for example, `USER1=`*`username1`*, `USER2=`*`username2`*, `ADDR1=`*`ip_address1`*, `ADDR2=`*`ip_address2`*, and so forth.
4. When a request enters the system, WLDF creates and instantiates a context for the request. The context includes a unique ID and a dye vector, as described in the following step.
5. When a request enters the system, the `DyeInjection` monitor examines the request to see which dye values in the dye vector match attributes of the request, if any. For example, it checks to see if the request originated with *`username1`* or *`username2`*, and it checks to see if the request came from *`ip_address1`* or *`ip_address2`*.
6. For each dye value that matches a request attribute, the `DyeInjection` monitor "injects" that dye into the request. This is done by setting the dye flag for that dye in the dye vector attached to the request. For example, if a request originates with *`username2`* from *`ip_address1`*, the `DyeInjection` monitor sets the dye flags `USER2` and `ADDR1`. (`USER1` and `ADDR2`, therefore, remain unset.)
7. The dye vector travels with the request (as part of the diagnostic context) as it flows through the system. This 64-bit dye vector contains only flags, not values. So, in this example, the dye vector contains only two flags that are explicitly set (`USER2` and `ADDR1`).

Note: All dye vectors also contain one of the implicit `PROTOCOL` dyes, as explained in "Configuring the Dye Vector via the DyeInjection Monitor."

8. The administrator enables dye filtering in one or more delegating diagnostic monitors and configures the dye mask for each monitor. If the dyes set in the dye mask exactly match the dyes in the dye vector attached to a request--that is, if `((dye-mask & dye-vector) == dye-mask)`--the diagnostic action for the monitor will be triggered when that request is processed.

These steps are discussed in more detail in the following sections.

Configuring the Dye Vector via the DyeInjection Monitor

To create contexts for requests, you must:

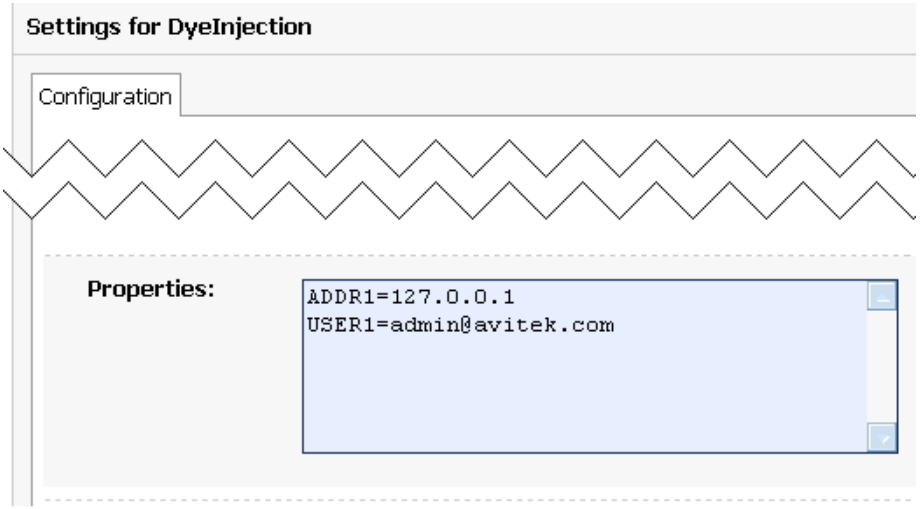
1. Create and enable a diagnostic module for the server or servers you want to monitor.
2. Enable Instrumentation for the module.
3. Configure and enable the `DyeInjection` monitor for the module. (Only one `DyeInjection` monitor can be used with a module at any one time.)

To configure the `DyeInjection` monitor, you assign values to the dyes in the dye vector. The available dye flags are described in Table 10-1. When WLDF evaluates an incoming request to create a context for it, it checks for the presence of the values specified in the dye vector. When a value is present, WLDF sets that flag. This process is called *dyeing* the request or *injecting a dye* into the request.

For example, to monitor all requests initiated by a user named `admin@avitek` from a client at IP address 127.0.0.0, assign the value `admin@avitek` to `USER1` and assign the value 127.0.0.0 to `ADDR1`. Then, when user `admin@avitek` initiates a request from a client with IP address 127.0.0.0, that request is dyed with `USER1` and `ADDR1`; in other words the `USER1` and `ADDR1` flags in the dye vector (in the context for the request) are both set.

In the Administration Console, you assign values to dyes by typing them into the Properties field of the Settings for DyeInjection page.

Figure 10-1 Setting Dye Values in the Administration Console



These settings appear in the descriptor file for the diagnostic module, as shown in the following code listing.

Listing 10-1 Sample DyeInjection Monitor Configuration

```
<wldf-resource>
  <name>MyAdminModule</name>
  <instrumentation>
    <enabled>true</enabled>
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <enabled>true</enabled>
      <dye-mask xsi:nil="true"></dye-mask>
      <properties>ADDR1=127.0.0.1 USER1=admin@avitek</properties>
    </wldf-instrumentation-monitor>
  <!-- Other elements to configure instrumentation -->
  <instrumentation>
<!-- Other elements to configure this diagnostic monitor -->
<wldf-resource>
```

Dyes Supported by the DyeInjection Monitor

The dyes available in the dye vector are listed and explained in the following table.

Table 10-1 Request Protocols for Supported Diagnostic Context Dyes

Dye Flags	Description
ADDR1 ADDR2 ADDR3 ADDR4	The `ADDR1`, `ADDR2`, `ADDR3` and `ADDR4` dyes can be used to specify the IP addresses of clients that originate requests. These dye flags are set in the diagnostic context for a request if the request was originated from an IP address specified by the respective property (`ADDR1`, `ADDR2`, `ADDR3`, `ADDR4`) of the `DyeInjection` monitor.
CONNECTOR1 CONNECTOR2 CONNECTOR3 CONNECTOR4	The `CONNECTOR1`, `CONNECTOR2`, `CONNECTOR3` and `CONNECTOR4` dyes can be used to identify characteristics of connector drivers. These dye flags are set by the connector drivers to identify request properties specific to their situations. You do not configure these directly in the Administration console or in the descriptor files. The connector drivers can assign values to these dyes (using the Connector API), so information about the connections can be carried in the diagnostic context.
COOKIE1 COOKIE2 COOKIE3 COOKIE4	`COOKIE1`, `COOKIE2`, `COOKIE3` and `COOKIE4` are set in the diagnostic context for an HTTP/S request, if the request contains the cookie named `weblogic.diagnostics.dye` and its value is equal to the value of the respective property (`COOKIE1`, `COOKIE2`, `COOKIE3`, `COOKIE4`) of the `DyeInjection` monitor.
DYE_0 DYE_1 DYE_2 DYE_3 DYE_4 DYE_5 DYE_6 DYE_7	`DYE_0` to `DYE_7` are available only for use by application developers. See "Using weblogic.diagnostics.context" on page 10-14.

Table 10-1 Request Protocols for Supported Diagnostic Context Dyes

Dye Flags	Description
PROTOCOL_HTTP PROTOCOL_IIOP PROTOCOL_JRMP PROTOCOL_RMI PROTOCOL_SOAP PROTOCOL_SSL PROTOCOL_T3	The `DyeInjection` monitor implicitly identifies the protocol used for a request and sets the appropriate dye(s) in the dye vector, according to the protocol(s) used. • `PROTOCOL_HTTP` is set in the diagnostic context of a request if the request uses HTTP or HTTPS protocol. • `PROTOCOL_IIOP` is set in the diagnostic context of a request if it uses IIOP protocol. • `PROTOCOL_JRMP` is set in the diagnostic context of a request if it uses JRMP protocol. • `PROTOCOL_RMI` is set in the diagnostic context of a request if it uses RMI protocol. • `PROTOCOL_SSL` is set in the diagnostic context of a request if it uses SSL protocol. • `PROTOCOL_T3` is set in the diagnostic context of a request if the request uses T3 or T3s protocol
THROTTLE	The `THROTTLE` dye is set in the diagnostic context of a request if it satisfies requirements specified by `THROTTLE_INTERVAL` and/or `THROTTLE_RATE` properties of the `DyeInjection` monitor.
USER1 USER2 USER3 USER4	The `USER1`, `USER2`, `USER3` and `USER4` dyes can be used to specify the user names of clients that originate requests. These dye flags are set in the diagnostic context for a request if the request was originated by a user specified by the respective property (`USER1`, `USER2`, `USER3`, `USER4`) of the `DyeInjection` monitor.

About the PROTOCOL Dye Flags

You must explicate set the values for the dye flags `USER`*n*, `ADDR`*n*, `COOKIE`*n*, and `CONNECTOR`*n*. in the `DyeInjection` monitor. However, the flags `PROTOCOL_HTTP`, `PROTOCOL_IIOP`, `ROTOCOL_JRMP`, `PROTOCOL_RMI`, `PROTOCOL_SOAP`, `PROTOCOL_SSL`, and `PROTOCOL_T3` are set implicitly by WLDF. When the `DyeInjection` monitor is enabled, every request is injected with the appropriate protocol dye. For example, every request that arrives via HTTP is injected with the `PROTOCOL_HTTP` dye.

About the THROTTLE Dye Flag

The `THROTTLE` dye flag can be used to control the volume of incoming requests that are dyed. `THROTTLE` is configured differently from the other flags, and WLDF uses it differently. For more information, see "Using Throttling to Control the Volume of Instrumentation Events."

When Contexts Are Created

When the `DyeInjection` monitor is enabled in a diagnostic module, a diagnostic context is created for every incoming request. Even if no properties are explicitly set in the `DyeInjection` monitor, the context for every request will contain a unique context ID and a dye vector with one of the implicit `PROTOCOL` dyes. If the `DyeInjection` monitor is not added to a diagnostic module or if it is disabled, no diagnostic contexts will be created for any incoming requests (however, see the following note).

Note: In the Administration Console, you can create diagnostic contexts only by enabling the `DyeInjection` monitor. However, through WLST, you can set the `DiagnosticContextEnabled` attribute on the `WLDFServerDiagnosticMBean`.

Configuring Delegating Monitors to Use Dye Filtering

You can use the `DyeInjection` monitor as a mechanism to restrict when a delegating or custom diagnostic monitor in the diagnostic module is triggered. This process is called *dye filtering*.

Each monitor can have a *dye mask*, which specifies a selection of the dyes from the `DyeInjection` monitor. When dye filtering is enabled for a diagnostic monitor, the monitor's diagnostic action is triggered only for those requests that meet the criteria set by the mask.

For example, consider a `Connector_After_Inbound` diagnostic monitor that has a `TraceAction` action attached to it. If dye filtering is *not* enabled, any request that is handled by `Connector_After_Inbound` (that is, any request that exits a method) will trigger a `TraceAction`. However, you could use a dye mask to trigger those `TracActions` only for requests that originated from a IP address `127.0.0.0`, as explained below:

1. Configure the `DyeInjection` monitor so that `ADDR1=127.0.0.0`, and enable the `DyeInjection` monitor. For instructions, see "Configuring the Dye Vector via the DyeInjection Monitor," earlier in this section.
2. Configure a *dye mask* and enable dye filtering for the `Connector_After_Inbound` diagnostic monitor. In the Administration Console, you do this in the Settings for Connector_After_Inbound page, as described below and shown in Figure 10-2.

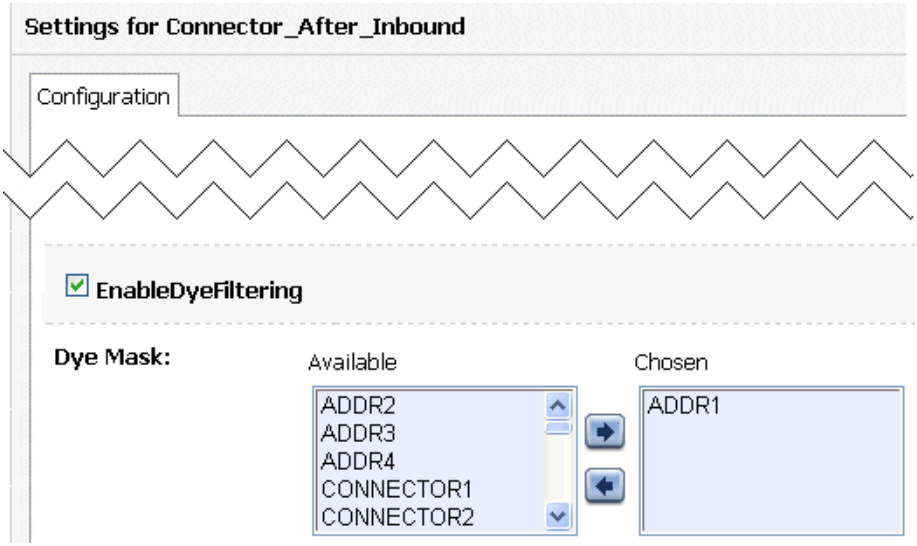
a. Navigate to the Settings for Connector_After_Inbound page. (See the Administration Console online help for complete instructions for adding and configuring the Connector_After_Inbound diagnostic monitor in a diagnostic module.)

b. In the **Dye Mask** field of the Settings for Connector_After_Inbound page, move `ADDR1` from the **Available** list to the **Chosen** list.

c. Select the **EnableDyeFiltering** check box.

With this configuration, the `TraceAction` action will be triggered for the `Connector_After_Inbound` diagnostic monitor only for those requests that originate from IP address `127.0.0.0`.

Figure 10-2 Setting Dye Filtering in the Administration Console



These settings appear in the descriptor file for the diagnostic module as shown in the following code listing.

Listing 10-2 Sample Configuration for Using Dye Filtering in a Delegating Monitor

```
<wldf-resource>
  <name>MyAdminModule</name>
```

```
  <instrumentation>
    <enabled>true</enabled>
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <enabled>true</enabled>
      <properties>ADDR1=127.0.0.1</properties>
    </wldf-instrumentation-monitor>
    <wldf-instrumentation-monitor>
      <name>Connector_After_Inbound</name>
      <dye-mask>ADDR1</dye-mask>
      <dye-filtering-enabled>true</dye-filtering-enabled>
      <action>TraceAction</action>
    </wldf-instrumentation-monitor>
  <!-- Other elements to configure instrumentation -->
  <instrumentation>
<!-- Other elements to configure this diagnostic monitor -->
<wldf-resource>
```

How Dye Masks Filter Requests To Pass to Monitors

A dye vector attached to a request can contain multiple dyes, and a dye mask attached to a delegating monitor can contain multiple dyes. For a delegating monitor's dye mask to allow a monitor to take action on a request, all of the following must be true:

- The `DyeInjection` monitor is enabled for the diagnostic module. (If the `DyeInjection` monitor is not added or is disabled, dye filtering is disabled.)
- Dye filtering for the delegating or custom monitor is enabled.
- The request's dye vector contains all the dyes that are defined in the dye mask. (The dye vector can also contain dyes that are not in the dye mask, but the dye mask cannot contain dyes that are not in the dye vector.)

Using Throttling to Control the Volume of Instrumentation Events

Throttling is used to control the number of requests that are processed by the monitors in a diagnostic module. Throttling is configured using the THROTTLE dye, which is defined in the DyeInjection monitor.

Configuring the THROTTLE Dye

Unlike other dyes in the dye vector, the THROTTLE dye is configured through two properties.

- THROTTLE_INTERVAL sets an interval (in milliseconds) after which a new incoming request is dyed with the THROTTLE dye.

  If the THROTTLE_INTERVAL is greater than 0, the DyeInjection monitor sets the THROTTLE dye flag in the dye vector of an incoming request if the last request dyed with THROTTLE arrived at least *THROTTLE_INTERVAL* before the new request. For example, if THROTTLE_INTERVAL=3000, the DyeInjection monitor waits at least 3000 milliseconds before it will dye an incoming request with THROTTLE.

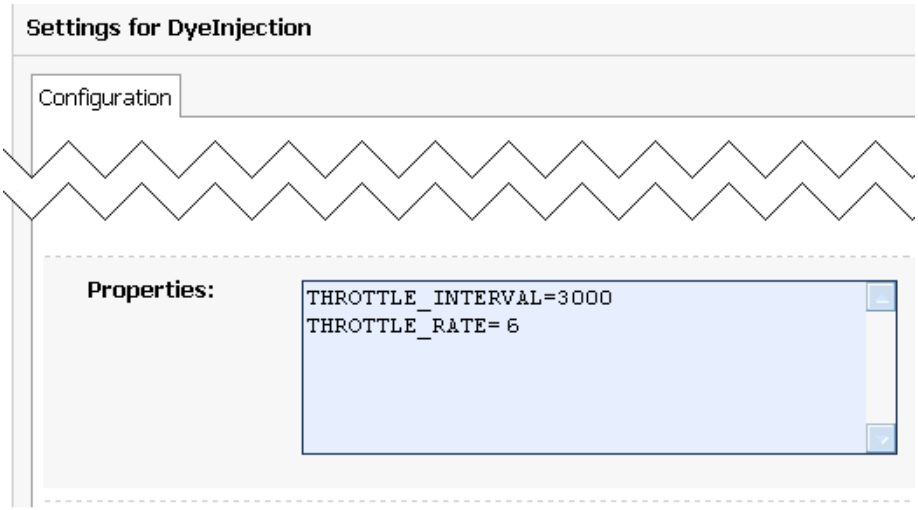
- THROTTLE_RATE sets the rate (in terms of the number of incoming requests) by which new incoming requests are dyed with the THROTTLE dye.

  If THROTTLE_RATE is greater than 0, the DyeInjection monitor sets the THROTTLE dye flag in the dye vector of an incoming request when the number of requests since the last request dyed with THROTTLE equals *THROTTLE_RATE*. For example, if THROTTLE_RATE = 6, every sixth request is dyed with THROTTLE.

THROTTLE_INTERVAL and THROTTLE_RATE can be used together. Both conditions must be satisfied for a request to be dyed with THROTTLE. For example, if THROTTLE_RATE=6, but the sixth request (after the last THROTTLE-dyed request) arrives before THROTTLE_INTERVAL has elapsed, the sixth request will not be dyed. The next request after THROTTLE_INTERVAL elapses will be dyed, and the counting for the THROTTLE_RATE will be reset. Conversely, if THROTTLE_INTERVAL elapses, but the number of requests since the last THROTTLE-dyed request is not greater than or equal to the THROTTLE_RATE, no incoming request will be dyed until the THROTTLE_RATE is reached.

If you assign a value to either THROTTLE_INTERVAL or THROTTLE_RATE (or both, or neither), you are configuring the THROTTLE dye. A THROTTLE configuration setting in the Administration Console is shown in the following figure.

Figure 10-3 Configuring the THROTTLE Dye



Listing 10-3 shows the resulting configuration in the descriptor file for the diagnostics module.

Listing 10-3 Sample THROTTLE Configuration in the DyeInjection Monitor

```
<wldf-instrumentation-monitor>
  <name>DyeInjection</name>
  <properties>
    THROTTLE_INTERVAL=3000
      THROTTLE_RATE=6
    </properties>
  </wldf-instrumentation-monitor>
```

Listing 10-4 shows the configuration for a `Connector_After_Inbound` delegating monitor where the `THROTTLE` dye is set in the dye mask for the monitor.

Listing 10-4 Sample Configuration for Setting THROTTLE in a Dye Mask of a Delegating Monitor

```
<wldf-instrumentation-monitor>
  <name>Connector_After_Inbound</name>
  <enabled>true</enabled>
  <dye-mask>THROTTLE</dye-mask>
</wldf-instrumentation-monitor>
```

How Throttling is Handled by Delegating and Custom Monitors

Dye masks and dye filtering provide a mechanism for restricting which requests are passed to delegating and custom monitors for handling, based on properties of the requests. The presence of a property in a request is indicated by the presence of a dye, as discussed in "Configuring the Dye Vector via the DyeInjection Monitor," earlier in this section. One of those dyes can be the `THROTTLE` dye, so that you can filter on `THROTTLE`, just like any other dye. However, the most common way to use `THROTTLE` is to use it without dye filtering to restrict which requests are passed to delegating monitors.

The items in the following list explain how throttling is handled:

- If dye filtering for a delegating or custom monitor is enabled and that monitor has a dye mask, filtering is performed based on the dye mask. That mask may include the `THROTTLE` dye, but it does not have to. If `THROTTLE` is included in a dye mask, then `THROTTLE` must also be included in the request's dye vector for the request to be passed to the monitor. However, if `THROTTLE` is not included in the dye mask, all qualifying requests are passed to the monitor, whether their dye vectors include `THROTTLE` or not.
- If dye filtering for a delegating or custom monitor is not enabled and neither `THROTTLE` property is set in the `DyeInjection` monitor, dye filtering will not take place and throttling will not take place.
- If dye filtering for a delegating or custom monitor is not enabled and `THROTTLE` is configured in the `DyeInjection` monitor, delegating monitors ignore dye masks but do check for the presence of the `THROTTLE` dye in all requests. Only those requests dyed with `THROTTLE` are passed to the delegating monitors for handling. Therefore, by setting a `THROTTLE_RATE` and/or `THROTTLE_INTERVAL` in the `DyeInjection` monitor, you reduce the number of requests handled by all delegating monitors. You do not have to configure dye masks for all your delegating monitors to take advantage of throttling.

- If dye filtering for a delegating or custom monitor is enabled and the only dye set in a dye mask is `THROTTLE`, only those requests that are dyed with `THROTTLE` are passed to the delegating monitor. This behavior is the same as when dye filtering is not enabled and `THROTTLE` is configured in the `DyeInjection` monitor.

Using weblogic.diagnostics.context

The `weblogic.diagnostics.context` package provides applications limited access to a diagnostic context.

An application can use the `weblogic.diagnostics.context.DiagnosticContextHelper` APIs to perform the following functions:

- Inspect a diagnostics context's immutable context ID.
- Inspect the settings of the dye flags in a context's dye vector.
- Retrieve an array of valid dye flag names.
- Set, or unset, the `DYE_0` through `DYE_7` flags in a context's dye vector. (Note that there is no way to set these flag bits via XML. You can configure `DyeInjection` monitor `<properties>` to set the non-application-specific flag bits via XML, but `setDye()` is the only method for setting `DYE_0` through `DYE_7` in a dye vector.)
- Create a dye mask for its own use.
- Attach a payload (a String) to a diagnostic context, or read an existing payload.

An application cannot:

- Set any flags in a dye vector other the eight flags reserved for applications.
- Prevent another application for setting the same application flags in a dye vector. A well-behaved application can test whether a dye flag is set before setting it.
- Prevent another application from replacing a payload. A well-behaved application can test for the presence of a payload before adding one.

A monitor, or another application, that is downstream from the point where an application has set one or more of the `DYE_0` through `DYE_7` flags can set a dye mask to check for those flags, and take an action when the flag(s) are present in a context's dye vector. If a payload is attached to the diagnostics context, any action taken by that monitor will result in the payload being archived, and thus available through the accessor component.

Listing 10-5 is a short example which (implicitly) creates a diagnostic context, prints the context ID, checks the value of the DYE_0 flag, and then sets the DYE_0 flag.

Listing 10-5 Example: DiagnosticContextExample.java

```
package weblogic.diagnostics.examples;

import weblogic.diagnostics.context.DiagnosticContextHelper;

public class DiagnosticContextExample {

 public static void main(String args[]) throws Exception {
   System.out.println("\nContextId=" +
     DiagnosticContextHelper.getContextId());
     System.out.println("isDyedWith(DYE_0)=" +
       DiagnosticContextHelper.isDyedWith(DiagnosticContextHelper.DYE_0));

     DiagnosticContextHelper.setDye(DiagnosticContextHelper.DYE_0, true);
     System.out.println("isDyedWith(DYE_0)=" +
       DiagnosticContextHelper.isDyedWith(DiagnosticContextHelper.DYE_0));
 }
}
```

CHAPTER 11

Accessing Diagnostic Data Using the Data Accessor

The Data Accessor component of the WebLogic Diagnostic Framework (WLDF) provides the means for accessing diagnostic data from various sources, including log records, data events, and harvested metrics.

Using the Data Accessor, you can perform data lookups by type, component, and attribute. You can perform time-based filtering and, when accessing events, filtering by severity, source, and content. You can also access diagnostic data in tabular form.

The following sections describe the Data Accessor and tell how to use it online (when a server is running) and offline (when a server is not running):

About the Data Stores Accessed by the Data Accessor

The data accessor retrieves diagnostic information from other WLDF 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.

WLDF maintains diagnostic data on a per-server basis. Therefore, the Data Accessor provides access to data stores for individual servers.

Data stores can be modeled as tabular data. Each record in the table represents one item, and the columns describe characteristics of the item. Different data stores may have different columns. However, most data stores have some of the same columns, such as the time when the data was collected.

The Data Accessor can retrieve the following information about data stores used by WLDF for a server:

- A list of supported data store types, including:
  - HTTP_LOG
  - HARVESTED_DATA_ARCHIVE
  - EVENTS_DATA_ARCHIVE
  - SERVER_LOG
  - DOMAIN_LOG
  - HTTP_ACCESS_LOG
  - WEBAPP_LOG
  - CONNECTOR_LOG
  - JMS_MESSAGE_LOG
  - CUSTOM_LOG
- A list of available data store instances
- The layout of each data store (information that describes the columns in the data store)

You can use the `WLDFAccessRuntimeMBean` to discover such data stores, determine the nature of the data they contain, and access their data selectively using a query.

For complete documentation about WebLogic logs, see *Configuring Log Files and Filtering Log Messages*.

Accessing Diagnostic Data Online

You can access diagnostic data from a running server by using the Administration Console, JMX APIs, or the WebLogic Scripting Tool (WLST).

Accessing Data Using the Administration Console

You don't use the Data Accessor explicitly in the Administration Console, but information collected by the accessor is displayed, for example, in the Summary of Log Files page. See "View and Configure Logs" in the *Administration Console Online Help*.

Accessing Data Programmatically Using Runtime MBeans

The Data Accessor provides the following runtime MBeans for discovering data stores and retrieving data from them:

- Use the `WLDFAccessRuntimeMBean` to do the following:
  - Get the logical names of the available data stores on the server.
  - Look up a `WLDFDataAccessRuntimeMBean` to access the data from a specific data source, based on its logical name. The different data stores are uniquely identified by their logical names.

  See WLDFAccessRuntimeMBean in the *WebLogic Server MBean Reference*.

- Use the `WLDFDataAccessRuntimeMBean` to retrieve data stores based on a search condition, or query. You can optionally specify a time interval with the query, to retrieve data records within a specified time duration. This MBean provides meta-data about the columns of the data set and the earliest and latest timestamp of the records in the data store.

  Data Accessor runtime mbeans are currently created and registered lazily. So, when a remote client attempts to access them, they may not be present and an InstanceNotFoundException may be thrown.

  The client can retrieve the `WLDFDataAccessRuntimes` attribute of the `WLDFAccessRuntime` to cause all known data access runtimes to be created, for example:

```
ObjectName objName =
    new ObjectName("com.bea:ServerRuntime=" + serverName +
                   ",Name=Accessor," +
                   "Type=WLDFAccessRuntime," +
                   "WLDFRuntime=WLDFRuntime");
rmbs.getAttribute(objName, "WLDFDataAccessRuntimes");
```

  See WLDFDataAccessRuntimeMBean in the *WebLogic Server MBean Reference*.

Note: If the clock on a server hosting a WebLogic Server instance is reset, it can give you unexpected results when you query based on a timestamp.

Using WLST to Access Diagnostic Data Online

Use the WLST `exportDiagnosticDataFromServer` command to access diagnostic data from a running server. For the syntax and examples of this command, see "Diagnostic Commands," in the *WLST Command and Variable Reference*.

Using the WLDF Query Language with the Data Accessor

To query data from data stores, use the WLDF query language. For Data Accessor query language syntax, see Appendix A, "WLDF Query Language."

Accessing Diagnostic Data Offline

Use the WLST `exportDiagnosticData` command to access historical diagnostic data from an offline server. For the syntax and examples of this command, see "Diagnostics Commands" in the *WLST Command and Variable Reference*.

Notes: You can use `exportDiagnosticData` to access archived data only from the machine on which the data is persisted.

You cannot discover data store instances using the offline mode of the Data Accessor. You must already know what they are.

CHAPTER 12

Introduction to Programming WLDF

You can use the WebLogic Server Administration Console to enable, configure, and monitor features of WebLogic Server, including the WebLogic Diagnostic Framework (WLDF). You can do the same tasks programmatically using the JMX API and the WebLogic Scripting Tool (WLST).

In addition to the information provided in the following sections, use the information in the following manuals to develop and deploy applications, and to use WLST:

- *Developing Applications with WebLogic Server*
- *Developing Manageable Applications with JMX*
- *Developing Custom Management Utilities with JMX*
- *Deploying Applications to WebLogic Server*
- *WebLogic Scripting Tool*

The WLDF framework consists of several components. If you conceptualize the flow of data in terms of data generation and data retrieval:

- The WLDF XML descriptor file settings for the Harvester, Instrumentation, Image Capture, and Watch and Notification components determine the type and amount of diagnostic data generated while a server is running.
- The diagnostic context and instrumentation settings filter and monitor this data as it flows through the system. Data is harvested, actions are triggered, events are generated, and configured notifications are sent.

- The Archive component stores the data.
- The Accessor component retrieves the data.

In general, server input configuration is primarily an administrative task, accomplished either through the console or through WLST scripts. Deployable descriptor modules, XML configuration files, are the primary method for configuring diagnostic resources at both the system level (servers and clusters) and at the application level. (For information on configuring WLDF resources, see Chapter 2, "Understanding WLDF Configuration.")

Output retrieval via the Accessor component can be either an administrative or a programmatic task.

When you create WLDF resources using the Administration Console or WLST, WebLogic Server creates MBeans, managed beans, for each resource. You can then access these MBeans using JMX or the WebLogic Scripting Tool (WLST). Because `weblogic.WLST` is a JMX client; any task you can perform using WLST you can also perform programmatically through JMX.

Table 12-1 lists the beans and packages associated with WLDF and its components. Figure 12-1 groups the beans by type.

Table 12-1 Mapping WLDF Components to Beans and Packages

Component	Beans / Packages

WLDF	`WLDFServerDiagnosticMBean`
	`WLDFSystemResourceMBean`
	`WLDFBean (abstract)`
	`WLDFResourceBean`
	`WLDFRuntimeMBean`

Diagnostic Image	`WLDFImageNotificationBean`
	`WLDFImageCreationTaskRuntimeMBean`
	`WLDFImageRuntimeMBean`

Instrumentation	`WLDFInstrumentationBean`
	`WLDFInstrumentationMonitorBean`
	`WLDFInstrumentationRuntimeMBean`

Table 12-1 Mapping WLDF Components to Beans and Packages

Component	Beans / Packages

Diagnostic Context	Package: `weblogic.diagnostics.context`
	`DiagnosticContextHelper`
	`DiagnosticContextConstants`

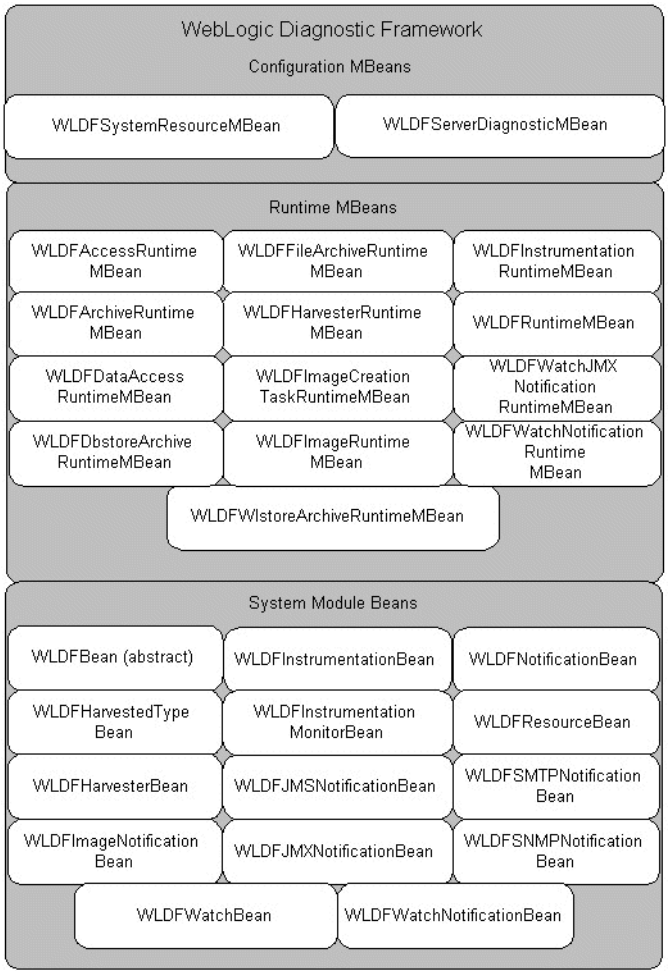
Harvester	`WLDFHarvesterBean`
	`WLDFHarvestedTypeBean`
	`WLDFHarvesterRuntimeMBean`

Watch & Notification	`WLDFNotificationBean`
	`WLDFWatchNotificationBean`
	`WLDFJMSNotificationBean`
	`WLDFJMXNotificationBean`
	`WLDFSMTPNotificationBean`
	`WLDFSNMPNotificationBean`
	`WLDFWatchJMXNotificationRuntimeMBean`
	`WLDFWatchNotificationRuntimeMBean`
	Package: `weblogic.diagnostics.watch`
	`JMXWatchNotification`
	`WatchNotification`

Archive	`WLDFArchiveRuntimeMBean`
	`WLDFDbstoreArchiveRuntimeMBean`
	`WLDFFileArchiveRuntimeMBean`
	`WLDFWlstoreArchiveRuntimeMBean`

Accessor	`WLDFAccessRuntimeMBean` `WLDFDataAccessRuntimeMBean`

Figure 12-1 WLDF Configuration MBeans, Runtime MBeans, and System Module Beans



Programming Tools

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

- Use diagnostic descriptor files to configure WLDF components: Harvester, Instrumentation, and Watch and Notification at the server level; Instrumentation at the application level. (Note that these components are the only WLDF components that can be configured via descriptor files.)
- Use JMX to access exposed WLDF operations and attributes.
- 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
  - Capture notifications using JMS
  - Retrieve archived data through the Accessor. (The Accessor, as are the other components, is surfaced as JMX; you can use WLST or straight JMX programming to retrieve diagnostic data.)

Configuration and Runtime APIs

The configuration and runtime APIs configure and monitor WLDF. Both the configuration and the runtime APIs are exposed as MBeans.

- The configuration MBeans and system module Beans create and configure WLDF resources, and determine their runtime behavior.
- The runtime MBeans monitor the runtime state and the operations defined for the different components.

You can use the APIs to configure, activate, and deactivate data collection; to configure watches, notifications, alarms, and diagnostic image captures; and to access data.

This section covers the following topics:

- "Configuration APIs" on page 12-5
- "Runtime APIs" on page 12-6

Configuration APIs

The Configuration APIs define interfaces that are used to configure the following WLDF components:

- Data Collectors: You can use the configuration APIs to configure and control Instrumentation, Harvesting, and Image Capture.
  - For the Instrumentation component, 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 component, 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 Diagnostic Image Capture component, 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.
- Watch and Notifications: 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 diagnostic image capture
  - Add and remove notifications from watches
- Archive: Set the archive type and the archive directory

Runtime APIs

The runtime APIs define interfaces that are used to monitor the runtime state of the WLDF 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 WLDF components. These APIs are included in the `weblogic.management.runtime` package.

You can use the runtime APIs to monitor the following WLDF components:

- Data Collectors—You can use the runtime APIs to monitor the Instrumentation, Harvester, and the Image Capture components.

  - For the Instrumentation component, you can monitor 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 component, 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 Image Capture component, you can specify the destination and lockout period for diagnostic images and initiate image captures.

- Watches and Notifications: You can use the runtime APIs to monitor the Watches and Notifications and Archive components.
  - For the Watches and Notifications component, 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.
- Archive: You can monitor information about the archive, such as file name and archive statistics.
- Data Accessor—You can use the runtime APIs to retrieve 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).

WLDF Packages

The following two packages are provided:

- `weblogic.diagnostics.context` contains:
  - `DiagnosticContextConstants`, which defines the indices of dye flags supported by the WebLogic diagnostics system.
  - `DiagnosticConDtextHelper`, which provides applications limited access to the diagnostic context.
- `weblogic.diagnostics.watch` contains:

- `JMXWatchNotification`, an extended JMX notification object which includes additional information about the notification. This information is contained in the referenced `WatchNotification` object returned from method `getExtendedInfo`.
- `WatchNotification`, which defines a notification for a watch rule.

Deploying WLDF Application Modules

WLDF configurations are stored in XML descriptor files that conform to the WebLogic Server schema `weblogic-diagnostics.xsd`. The schema is available at the following URL:

http://www.bea.com/ns/weblogic/90/diagnostics.xsd

See *WebLogic Server Diagnostics Configuration Schema Reference* for documentation.

You create and manage WLDF resources either as system modules or as application modules, similar to standard J2EE modules.

A WLDF application module can be deployed as a stand-alone resource, in which case the resource is available to the servers or cluster targeted during the deployment process, or as part of an enterprise application. An application module deployed as part of an enterprise application is available only to the enclosing application (an application-scoped resource). Using application-scoped resources ensures that an application always has access to required resources, and simplifies the process of deploying the application into new environments.

In contrast to system modules, application modules are owned by the developer who created and packaged the module, rather than the administrator who deploys the module. This means that the administrator has more limited control over WLDF application modules. When deploying an application module, an administrator can change resource properties that were specified in the module, but cannot add or delete resources.

Application modules can be deployed, or undeployed, using JSR-88 interfaces, and updated by a deployment plan.

Table 12-2, “Comparing System and Application Modules,” on page 12-9 compares the function and scope of system and application modules.

Table 12-2 Comparing System and Application Modules

Module Type	Add/Remove Objects Dynamically	Add/Remove Objects with Console	Modify with JMX Remotely	Modify with JSR-88 (non-remote)	Modify with Console
System Module	Yes	Yes	Yes	No	Yes - via JMX
Application Module	No - module must be redeployed	No	No	Yes	Yes - via plan

For detailed information on creating modules and deploying applications, see *Deploying Applications to WebLogic Server*.

Programming WLDF: Examples

The following examples use WLDF beans and packages to access and modify information on a running server:

- "Example: DiagnosticContextExample.java" on page 12-9
- "Example: HarvesterMonitor.java" on page 12-10
- "Example: JMXAccessorExample.java" on page 12-19

In addition, see the WLST and JMX examples in Appendix C, "WebLogic Scripting Tool Examples."

Example: DiagnosticContextExample.java

The following example uses the `DiagnosticContextHelper` class from the `weblogic.diagnostics.context` package to get and set the value of the DYE_0 flag. (For information on diagnostic contexts, see Chapter 10, "Configuring the Diagnostic Context.")

To compile and run the program:

1. Copy the DiagnosticContextExample.java example (Listing 12-2) to a directory and compile it with:

   ```
   javac -d . DiagnosticContextExample.java
   ```

This will create the `./weblogic/diagnostics/examples` directory and populate it with `DiagnosticContextExample.class`.

2. Run the program. The command syntax is:

   ```
   java weblogic.diagnostics.examples.DiagnosticContextExample
   ```

   Sample output is similar to:

   ```
   # java weblogic.diagnostics.examples.DiagnosticContextExample

   ContextId=5b7898f93bf010ce:40305614:1048582efd4:-8000-0000000000000001
   isDyedWith(DYE_0)=false
   isDyedWith(DYE_0)=true
   ```

Listing 12-1 Example: DiagnosticContextExample.java

```
package weblogic.diagnostics.examples;

import weblogic.diagnostics.context.DiagnosticContextHelper;

public class DiagnosticContextExample {

  public static void main(String args[]) throws Exception {
    System.out.println("ContextId=" +
       DiagnosticContextHelper.getContextId());
    System.out.println("isDyedWith(DYE_0)=" +
       DiagnosticContextHelper.isDyedWith(DiagnosticContextHelper.DYE_0));

    DiagnosticContextHelper.setDye(DiagnosticContextHelper.DYE_0, true);
    System.out.println("isDyedWith(DYE_0)=" +
       DiagnosticContextHelper.isDyedWith(DiagnosticContextHelper.DYE_0));
  }
}
```

Example: HarvesterMonitor.java

The `HarvesterMonitor` program uses the Harvester JMX notification to identify when a harvest cycle has occurred. It then retrieves the new values using the Accessor. All access is performed through JMX. This section includes a description of notification listeners followed by the `HarvesterMonitor.java` code:

- "Notifcation Listeners" on page 12-11
- "HarvesterMonitor.java" on page 12-12

For information on the Harvester component, see Chapter 5, "Configuring the Harvester for Metric Collection."

Notifcation 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 Framework using transport mediums other than SMTP, JMX, SNMP, or JMS. One approach is to use the JMX `NotificationListener` interface to implement an object, and then propagate the notification according to the requirements of the selected transport medium.

Table 12-3 describes each notification listener type that is provided with WebLogic Server and the relevant configuration settings for each type.

Table 12-3 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. Uses predefined singleton for posting the event.
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.

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

HarvesterMonitor.java

To compile and run the `HarvesterMonitor` program:

1. Copy the HarvesterMonitor.java example (Listing 12-2) to a directory and compile it with:

   ```
   javac -d . HarvesterMonitor.java
   ```

   This will create the `./weblogic/diagnostics/examples` directory and populate it with `HarvesterMonitor.class` and `HarvesterMonitor$HarvestCycleHandler.class`.

2. Start the monitor. The command syntax is:

   ```
   java HarvesterMonitor <server> <port> <uname> <pw> [<types>]
   ```

   You will need access to a WebLogic Server instance, and will need to know the server's name, port number, administrator's login name, and the administrator's password.

   You can provide an optional list of harvested type names. If provided, the program will display only the values for those types. However, for each selected type, the monitor displays the complete set of collected values; there is no way to constrain the values displayed for a selected type.

   Only values that are explicitly configured for harvesting are displayed. Values collected solely to support watch rules (implicit values) are not displayed.

   The following command requires that '**.**' is in the CLASSPATH variable, and that you run the command from the directory where you compiled the program. The command connects to the `myserver` server, at port `7001`, as user `weblogic`, with a password of `weblogic`:

   ```
   java weblogic.diagnostics.examples.HarvesterMonitor myserver 7001
     weblogic weblogic
   ```

   See Listing 12-3, "Sample Output from HarvesterMonitor," on page 12-18 for an example of output from the HarvesterMonitor.

Listing 12-2 Example: HarvesterMonitor.java

```
package weblogic.diagnostics.examples;

import weblogic.management.mbeanservers.runtime.RuntimeServiceMBean;

import javax.management.*;
import javax.management.remote.*;
import javax.naming.Context;
import java.util.*;
```

```
public class HarvesterMonitor {


  private static String accessorRuntimeMBeanName;
   private static ObjectName accessorRuntimeMBeanObjectName;

  private static String harvRuntimeMBeanName;
   private static ObjectName harvRuntimeMBeanObjectName;

  private static MBeanServerConnection rmbs;

  private static ObjectName getObjectName(String objectNameStr) {
    try { return new ObjectName(getCanonicalName(objectNameStr)); }
    catch (RuntimeException x) { throw x; }
    catch (Exception x) { x.printStackTrace(); throw new
                          RuntimeException(x); }
  }

  private static String getCanonicalName(String objectNameStr) {
    try { return new ObjectName(objectNameStr).getCanonicalName(); }
    catch (RuntimeException x) { throw x; }
    catch (Exception x) { x.printStackTrace(); throw new
                           RuntimeException(x); }
  }

  private static String serverName;
   private static int port;
   private static String userName;
   private static String password;

  private static ArrayList typesToMonitor = null;

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

    if (args.length < 4) {
       System.out.println(
            "Usage: java weblogic.diagnostics.harvester.HarvesterMonitor " +
            "<serverName> <port> <userName> <password> [<types>]" +
             weblogic.utils.PlatformConstants.EOL +
             "   where <types> (optional) is a comma-separated list " +
             "of types to monitor.");
       System.exit(1);
    }
```

```
    serverName = args[0];
    port = Integer.parseInt(args[1]);
    userName = args[2];
    password = args[3];

    accessorRuntimeMBeanName = getCanonicalName(
        "com.bea:ServerRuntime=" + serverName +
        ",Name=HarvestedDataArchive,Type=WLDFDataAccessRuntime" +
        ",WLDFAccessRuntime=Accessor,WLDFRuntime=WLDFRuntime");
   accessorRuntimeMBeanObjectName =
        getObjectName(accessorRuntimeMBeanName);

   harvRuntimeMBeanName = getCanonicalName(
        "com.bea:ServerRuntime=" + serverName +
        ",Name=WLDFHarvesterRuntime,Type=WLDFHarvesterRuntime" +
        ",WLDFRuntime=WLDFRuntime");
    harvRuntimeMBeanObjectName = getObjectName(harvRuntimeMBeanName);

    if (args.length > 4) {
      String typesStr = args[4];
      typesToMonitor = new ArrayList();
      int index;
      while ((index = typesStr.indexOf(",")) > 0) {
        String typeName = typesStr.substring(0,index).trim();
        typesToMonitor.add(typeName);
        typesStr = typesStr.substring(index+1);
      }
      typesToMonitor.add(typesStr.trim());
    }

    rmbs = getRuntimeMBeanServerConnection();

    new HarvesterMonitor().new HarvestCycleHandler();
    while(true) {Thread.sleep(100000);}
  }

  static protected String JNDI = "/jndi/";
  static public MBeanServerConnection getRuntimeMBeanServerConnection()
      throws Exception {

    JMXServiceURL serviceURL;
    serviceURL =
```

```
      new JMXServiceURL("t3",
      "localhost",
      port,
      JNDI + RuntimeServiceMBean.MBEANSERVER_JNDI_NAME);
  System.out.println("ServerName=" + serverName);
  System.out.println("URL=" + serviceURL);

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

class HarvestCycleHandler implements NotificationListener {
  // used to track harvest cycles

  private int timestampIndex;
  private int domainIndex;
  private int serverIndex;
  private int typeIndex;
  private int instNameIndex;
  private int attrNameIndex;
  private int attrTypeIndex;
  private int attrValueIndex;

  long lastSampleTime = System.currentTimeMillis();

  HarvestCycleHandler() throws Exception{
    System.out.println("Harvester monitor started...");
    try {
      setUpRecordIndices();
      rmbs.addNotificationListener(harvRuntimeMBeanObjectName,
                                   this, null, null);
    }
    catch (javax.management.InstanceNotFoundException x) {
      System.out.println("Cannot find JMX data. " +
                         "Is the server name correct?");
      System.exit(1);
```

```
    }
  }
  private void setUpRecordIndices() throws Exception {
    Map columnIndexMap = (Map)rmbs.getAttribute(
        accessorRuntimeMBeanObjectName, "ColumnIndexMap");
    timestampIndex =
        ((Integer)columnIndexMap.get("TIMESTAMP")).intValue();
    domainIndex =
        ((Integer)columnIndexMap.get("DOMAIN")).intValue();
    serverIndex =
        ((Integer)columnIndexMap.get("SERVER")).intValue();
    typeIndex =
        ((Integer)columnIndexMap.get("TYPE")).intValue();
    instNameIndex =
        ((Integer)columnIndexMap.get("NAME")).intValue();
    attrNameIndex =
        ((Integer)columnIndexMap.get("ATTRNAME")).intValue();
    attrTypeIndex =
        ((Integer)columnIndexMap.get("ATTRTYPE")).intValue();
    attrValueIndex =
         ((Integer)columnIndexMap.get("ATTRVALUE")).intValue();
  }
  public synchronized void handleNotification(Notification notification,
                                              Object handback) {
     System.out.println("\n-----------------------------------------");
    long thisSampleTime = System.currentTimeMillis()+1;
    try {
      String lastTypeName = null;
      String lastInstName = null;
     String cursor = (String)rmbs.invoke(accessorRuntimeMBeanObjectName,
                      "openCursor",
                      new Object[]{new Long(lastSampleTime),
                      new Long(thisSampleTime), null},
                      new String[]{ "java.lang.Long",
                      "java.lang.Long", "java.lang.String" } );
      while (((Boolean)rmbs.invoke(accessorRuntimeMBeanObjectName,
```

```
                        "hasMoreData",
                        new Object[]{cursor},
                      new String[]{"java.lang.String"})).booleanValue()) {
        Object[] os = (Object[])rmbs.invoke(accessorRuntimeMBeanObjectName,
                        "fetch",
                        new Object[]{cursor},
                        new String[]{"java.lang.String"});
          for (int i = 0; i < os.length; i++) {
            Object[] values = (Object[])os[i];
            String typeName = (String)values[typeIndex];
            String instName = (String)values[instNameIndex];
            String attrName = (String)values[attrNameIndex];
            if (!typeName.equals(lastTypeName)) {
              if (typesToMonitor != null &&
                  !typesToMonitor.contains(typeName)) continue;
              System.out.println("\nType " + typeName);
              lastTypeName = typeName;
            }
            if (!instName.equals(lastInstName)) {
              System.out.println("\n  Instance " + instName);
              lastInstName = instName;
            }
            Object attrValue = values[attrValueIndex];
            System.out.println("    - " + attrName + "=" + attrValue);
          }
        }
        lastSampleTime = thisSampleTime;
      }
      catch (Exception e) {e.printStackTrace();}
    }
  }
}
```

Listing 12-3 contains sample output from the HarvesterMonitor program:

Listing 12-3 Sample Output from HarvesterMonitor

```
ServerName=myserver
URL=service:jmx:t3://localhost:7001/jndi/weblogic.management.mbeanservers.
runtime
Harvester monitor started...

-------------------------------------------------------

Type weblogic.management.runtime.WLDFHarvesterRuntimeMBean

Instance com.bea:Name=WLDFHarvesterRuntime,ServerRuntime=myserver,Type=WLD
FHarvesterRuntime,WLDFRuntime=WLDFRuntime
    - TotalSamplingTime=202048863
    - CurrentSnapshotElapsedTime=1839619

Type weblogic.management.runtime.ServerRuntimeMBean

  Instance com.bea:Name=myserver,Type=ServerRuntime
    - RestartRequired=false
    - ListenPortEnabled=true
    - ActivationTime=1118319317071
    - ServerStartupTime=40671
    - ServerClasspath= [deleted long classpath listing]
    - CurrentMachine=
    - SocketsOpenedTotalCount=1
    - State=RUNNING
    - RestartsTotalCount=0
    - AdminServer=true
    - AdminServerListenPort=7001
    - ClusterMaster=false
    - StateVal=2
    - CurrentDirectory=C:\testdomain\.
    - AdminServerHost=10.40.8.123
    - OpenSocketsCurrentCount=1
    - ShuttingDown=false
    - SSLListenPortEnabled=false
    - AdministrationPortEnabled=false
    - AdminServerListenPortSecure=false
    - Registered=true
```

Example: JMXAccessorExample.java

The following example program uses JMX to print log entries to standard out. All access is performed through JMX. (For information on the Accessor component, see Chapter 11, "Accessing Diagnostic Data Using the Data Accessor.")

To compile and run the program:

1. Copy the JMXAccessorExample.java example (Listing 12-4) to a directory and compile it with:

   ```
   javac -d . JMXAccessorExample.java
   ```

   This will create the `./weblogic/diagnostics/examples` directory and populate it with `JMXAccessorExample.class`.

2. Start the program. The command syntax is:

   ```
   java weblogic.diagnostics.example.JMXAccessor <logicalName> <query>
   ```

   You will need access to a WebLogic Server instance, and will need to know the server's name, port number, administrator's login name, and the administrator's password.

   The `logicalName` is the name of the log. Valid names are: `HarvestedDataArchive`, `EventsDataArchive`, `ServerLog`, `DomainLog`, `HTTPAccessLog`, `ServletAccessorHelper.WEBAPP_LOG`, `RAUtil.CONNECTOR_LOG`, `JMSMessageLog`, and `CUSTOM`.

   The `query` is constructed using the syntax described in Appendix A, "WLDF Query Language." For the `JMXAccessorExample` program, an empty `query` (an empty pair of double quotation marks, `""`) returns all entries in the log.

   The following command requires that '**.**' is in the CLASSPATH variable, and that you run the command from the directory where you compiled the program. The program uses the IIOP (Internet Inter-ORB Protocol) protocol to connect to port `7001`, as user `weblogic`, with a password of `weblogic`, and prints all entries in the ServerLog to standard out:

   ```
   java weblogic.diagnostics.examples.JMXAccessorExample ServerLog ""
   ```

   You can modify the example to use a username/password combination for your site.

Listing 12-4 JMXAccessorExample.java

```
package weblogic.diagnostics.examples;
```

```
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 JMXAccessorExample {

  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:\n" +
          "java weblogic.diagnostics.examples.JMXAccessorExample " +
          "<logicalName> <query>");
        System.exit(1);
      }
      String logicalName = args[0];
      String query = args[1];

      MBeanServerConnection mbeanServerConnection =
        lookupMBeanServerConnection();
      ObjectName service = new
        ObjectName(weblogic.management.mbeanservers.runtime.RuntimeServic
eMBean.OBJECT_NAME);
      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 {
```

```
        // construct JMX service URL
        JMXServiceURL serviceURL;
        serviceURL = new JMXServiceURL("iiop", "localhost", 7001,
            JNDI + "weblogic.management.mbeanservers.runtime");

        // Specify the user, password, and WebLogic provider package
        Hashtable h = new Hashtable();
        h.put(Context.SECURITY_PRINCIPAL,"weblogic");
        h.put(Context.SECURITY_CREDENTIALS,"weblogic");
        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();

      } // End - lookupMBeanServerConnection
    }
```

APPENDIX A

WLDF Query Language

The WebLogic Diagnostic Framework (WLDF) includes a query language for constructing watch rule expressions, Data Accessor query expressions, and log filter expressions. The syntax is a small and simplified subset of SQL syntax.

The language is described in the following sections:

Components of a Query Expression

A query expression may include any of the following:

- Operators. (See "Supported Operators" on page A-2.)
- Literals. (See "Supported Literals" on page A-3.)

- Variables. The supported variables differ for each type of expression. (See "About Variables in Expressions" on page A-4.)

The query language is case-sensitive.

Supported Operators

The query language supports the operators listed in Table A-1.

Table A-1 WLDF Query Language Operators

Operator	Operator Type	Supported Operand Types	Definition
`AND`	Logical binary	Boolean	Evaluates to true when both expressions are true
`OR`	Logical binary	Boolean	Evaluates to true when either expression is true
`NOT`	Logical unary	Boolean	Evaluates to true when the expression is not true
`=`	Relational	Numeric, String	Equals
`!=`	Relational	Numeric	Not equals
`<`	Relational	Numeric	Less than
`>`	Relational	Numeric	Greater than
`<=`	Relational	Numeric	Less than or equals
`>=`	Relational	Numeric	Greater than or equals
`LIKE`	Match	String	Evaluates to true when a character string matches a specified pattern that can include wildcards. `LIKE` supports two wildcard characters: • A percent sign `(%)` matches any string of zero or more characters • A period (.) matches any single character

Table A-1 WLDF Query Language Operators

Operator	Operator Type	Supported Operand Types	Definition
`MATCHES`	Match	String	Evaluates to true when the value of a regular expression is equivalent to the value of a `String`
`IN`	Search	String	Evaluates to true when the value of a variable exists in a predefined set, for example: `SUBSYSTEM IN ('A','B')`

Operator Precedence

The following list shows the levels of precedence among operators, from the highest precedence to the lowest. Operators listed on the same line have equivalent precedence:

```
1. ( )
2. NOT
3. =, !=, <, >, <=, >=, LIKE. MATCHES,IN
4. AND
5. OR
```

Supported Literals

Literals can be numeric or string types.

Numeric Literals

Rules for numeric literals are as follows:

- Numeric literals can be integers or floating point numbers.
- Numeric literals are specified the same as in Java. Some examples of numeric literals are 2, 2.0, 12.856f, 2.1934E-4, 123456L and 2.0D.

String Literals

Rules for string literals are as follows:

- String literals must be enclosed in single quotes.
- A percent character (`%`) can be used as a wildcard inside string literals.
- A backslash character (`\`) can be used to escape special characters, such as a quote (`'`) or a percent character (`%`).
- For watch rule expressions, you can use comparison operators to specify threshold values for String, Integer, Long, Double, Boolean literals.
- The relational operators do a lexical comparison for Strings. For more information, see the documentation for the `java.lang.String.compareTo(String str)` method.

About Variables in Expressions

Variables represent the dynamic portion of a query expression that is evaluated at runtime. You must use variables that are appropriate for the type of expression you are constructing, as documented in the following sections:

- "Creating Watch Rule Expressions" on page A-4
- "Creating Data Accessor Queries" on page A-7
- "Creating Log Filter Expressions" on page A-10

Creating Watch Rule Expressions

You can create watches based on log events, instrumentation events, and harvested attributes. The variables supported for creating the expressions are different for each type of watch, as described in the following sections:

- "Creating Log Event Watch Rule Expressions" on page A-5
- "Creating Instrumentation Event Watch Rule Expressions" on page A-6
- "Creating Harvester Watch Rule Expressions" on page A-7

For complete documentation about configuring and using WLDF watches, see:

- Chapter 6, "Configuring Watches and Notifications"
- Chapter 7, "Configuring Watches"

Creating Log Event Watch Rule Expressions

A *log event* watch rule expression is based upon the attributes of a log message from the server log.

Variable names for log message attributes are listed and explained in Table A-2:

Table A-2 Variable Names for Log Event Watch Rule Expressions

Variable	Description	Data Type
CONTEXTID	The request ID propagated with the request.	String
DATE	Date when the message was created.	String
MACHINE	Name of machine that generated the log message.	String
MESSAGE	Message content of the log message.	String
MSGID	ID of the log message (usually starts with "BEA=").	String
RECORDID	The number of the record in the log.	Long
SERVER	Name of server that generated the log message.	String
SEVERITY	Severity of log message. Values are ALERT, CRITICAL, DEBUG, EMERGENCY, ERROR, INFO, NOTICE, OFF, TRACE, and WARNING.	String
SUBSYTEM	Name of subsystem emitting the log message.	String
THREAD	Name of thread that generated the log message.	String
TIMESTAMP	Timestamp when the log message was created.	Long
TXID	JTA transaction ID of thread that generated the log message.	String
USERID	ID of the user that generated the log message.	String

An example log event watch rule expression is:

```
(SEVERITY = 'WARNING') AND (MSGID = 'BEA-320012')
```

Creating Instrumentation Event Watch Rule Expressions

An *instrumentation event* watch rule expression is based upon attributes of a data record created by a diagnostic monitor action.

Variable names for instrumentation data record attributes are listed and explained in Table A-3:

Table A-3 Variable Names for Instrumentation Event Rule Expressions

Variable	Description	Data Type
ARGUMENTS	Arguments passed to the method that was invoked.	String
CLASSNAME	Class name of joinpoint.	String
CONTEXTID	Diagnostic context ID of instrumentation event.	String
CTXPAYLOAD	The context payload associated with this request.	String
DOMAIN	Name of domain.	String
DYES	Dyes associated with this request.	Long
FILENAME	Source file name.	String
LINENUM	Line number in source file.	Integer
METHODNAME	Method name of joinpoint.	String
METHODDSC	Method arguments of joinpoint.	String
MODULE	Name of the diagnostic module.	String
MONITOR	Name of the diagnostic monitor.	String
PAYLOAD	Payload of instrumentation event.	String
RECORDID	The number of the record in the log.	Long
RETVAL	Return value of joinpoint.	String
SERVER	Name of server that created the instrumentation event.	String
TIMESTAMP	Timestamp when the instrumentation event was created.	Long
TXID	JTA transaction ID of thread that created the instrumentation event.	String

Table A-3 Variable Names for Instrumentation Event Rule Expressions

Variable	Description	Data Type
TYPE	Type of monitor.	String
USERID	ID of the user that created the instrumentation event.	String

An example instrumentation event data rule expression is:

```
(USERID = 'weblogic')
```

Creating Harvester Watch Rule Expressions

A *harvester* watch rule expression is based upon one or more harvested MBean attributes. The expression can specify an MBean type, an instance, and/or an attribute.

The syntax for constructing a Harvester watch rule expression is as follows:

- To specify an attribute of all instances of a type, use the following syntax:

  ```
  ${[type_name]//attribute_name}
  ```

- To specify an attribute of an instance of a type, use the following syntax:

  ```
  ${com.bea:Name=instance_name//attribute_name}
  ```

- To specify an attribute of an instance of a custom MBean type, use the following syntax:

  ```
  ${domain_name:Name=instance_name//attribute_name}
  ```

The expression must include the complete MBean object name, as shown in the following example:

```
${com.bea:Name=HarvesterRuntime,Location=myserver,Type=HarvesterRuntime,
      ServerRuntime=myserver//TotalSamplingCycles} > 10
```

Creating Data Accessor Queries

Use the WLDF query language with the Data Accessor component to retrieve data from data stores, including server logs, HTTP logs, and harvested metrics. The variables used to build a Data Accessor query are based on the column names in the data store from which you want to extract data.

A Data Accessor query contains the following:

- The logical name of a data store, as described in "Data Store Logical Names" on page A-8.
- Optionally, the name(s) of one or more columns from which to retrieve data, as described in "Data Store Column Names" on page A-9.

When there is a match, all columns of matching rows are returned.

Data Store Logical Names

The logical name for a data store must be unique. It denotes a specific data store available on the server. The logical name consists of a log type keyword followed by zero or more identifiers separated by the forward-slash (`/`) delimiter. For example, the logical name of the server log data store is simply `ServerLog`. However, other log types may require additional identifiers, as shown in Table A-4.

Table A-4 Naming Conventions for Log Types

Log Type	Optional Identifiers	Example

`ConnectorLog`	The JNDI name of the connection factory.	`ConnectorLog/eis/900eisaBlackBoxXATxConnectorJNDINAME`
		where
		`eis/900eisaBlackBoxXATxConnectorJNDINAME`
		is the JNDI name of the connection factory specified in the `weblogic-ra.xml` deployment descriptor.

`DomainLog`	None	`DomainLog`
`EventsDataArchive`	None	`EventsDataArchive`
`HarvestedDataArchive`	None	`HarvestedDataArchive`
`HTTPAccessLog`	Virtual host name	`HTTPAccessLog` - For the default web server's access log. `HTTPAccessLog/MyVirtualHost` - For the Virtual host named MyVirtualHost deployed to the current server. **Note:** In the case of `HTTPAccessLog`s with extended format, the number of columns are user-defined.
`JMSMessageLog`	The name of the JMS Server.	`JMSMessageLog/MyJMSServer`

Table A-4 Naming Conventions for Log Types

Log Type	Optional Identifiers	Example
`ServerLog`	None	`ServerLog`
`WebAppLog`	Web server name + Root servlet context name	`WebAppLog/MyWebServer/MyRootServletContext`

Data Store Column Names

The column names included in a query are resolved for each row of data. A row is added to the result set only if it satisfies the query conditions for all specified columns. A query that omits column names returns all the entries in the log.

All column names from all WebLogic Server log types are listed in Table A-5.

Table A-5 Column Names for Log Types

Log Type	Column Names
`ConnectorLog`	LINE, RECORDID
`DomainLog`	CONTEXTID, DATE, MACHINE, MESSAGE, MSGID, RECORDID, SERVER, SEVERITY, SUBSYSTEM, THREAD, TIMESTAMP, TXID, USERID
`EventsDataArchive`	ARGUMENTS, CLASSNAME, CONTEXTID, CTXPAYLOAD, DOMAIN, DYES, FILENAME, LINENUM, METHODNAME, METHODDSC, MODULE, MONITOR, PAYLOAD, RECORDID, RETVAL, SCOPE, SERVER, TIMESTAMP, TXID, TYPE, USERID
`HarvestedDataArchive`	ATTRNAME, ATTRTYPE, ATTRVALUE, DOMAIN, NAME, RECORDID, SERVER, TIMESTAMP, TYPE
`HTTPAccessLog`	AUTHUSER, BYTECOUNT, HOST, RECORDID, REMOTEUSER, REQUEST, STATUS, TIMESTAMP
`JMSMessageLog`	CONTEXTID, DATE, DESTINATION, EVENT, JMSCORRELATIONID, JMSMESSAGEID, MESSAGE, MESSAGECONSUMER, NANOTIMESTAMP, RECORDID, SELECTOR, TIMESTAMP, TXID, USERID

Table A-5 Column Names for Log Types

Log Type	Column Names
`ServerLog`	Same as `DomainLog`
`WebAppLog`	Same as `DomainLog`

An example of a Data Accessor query is:

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

The following example shows an API method invocation. It includes a query for harvested attributes of the JDBC connection pool named `MyPool`, within an interval between a `timeStampFrom` (inclusive) and a `timeStampTo` (exclusive):

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

For complete documentation about the WLDF Data Accessor, see Chapter 11, “Accessing Diagnostic Data Using the Data Accessor.”

Creating Log Filter Expressions

The query language can be used to filter what is written to the server log. The variables used to construct a log filter expression represent the columns in the log:

- CONTEXTID
- DATE
- MACHINE
- MESSAGE
- MSGID
- RECORDID
- SEVERITY
- SUBSYSTEM
- SERVER
- THREAD

- TIMESTAMP
- TXID
- USERID

Note: These are the same variables that are used to build a Data Accessor query for retrieving historical diagnostic data from existing server logs.

For complete documentation about the WebLogic Server logging services, see "Filtering WebLogic Server Log Messages" in *Configuring Log Files and Filtering Log Messages*.

Building Complex Expressions

You can build complex query expressions using sub-expressions containing variables, binary comparisons, and other complex sub-expressions. There is no limit on levels of nesting. The following rules apply:

- Nest queries by surrounding sub-expressions within parentheses, for example:

```
(Severity = 'Warning') AND (Id = 'BEA-320012')
```

- Enclose a variable name within `${}` if it includes special characters, as in an MBean object name. For example:

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

  Notice that the object name and the attribute name are separated by '//' in the watch variable name.

APPENDIX B

WLDF Instrumentation Library

The WebLogic Diagnostic Framework Instrumentation Library contains diagnostic monitors and diagnostic actions, as discussed in the following sections:

For information about using items from the Instrumentation Library, see Chapter 9, "Configuring Instrumentation."

Diagnostic Monitor Library

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.

The following table 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 B-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	Stateless	At entry of methods handling inbound connections.
`Connector_Before_Outbound`	Before	Stateless	At entry of methods handling outbound connections.
`Connector_Before_Work`	Before	Stateless	At entry of methods related to scheduling, starting and executing work items.
`Connector_Before_Tx`	Before	Stateless	Entry of transaction register, unregister, start, rollback and commit methods.
`Connector_After_Inbound`	Server	Stateless	At exit of methods handling inbound connections.
`Connector_After_Outbound`	After	Stateless	At exit of methods handling outbound connections.
`Connector_After_Work`	After	Stateless	At exit of methods related to scheduling, starting and executing work items.
`Connector_After_Tx`	After	Stateless	At exit of transaction register, unregister, start, rollback and commit methods.
`Connector_Around_Inbound`	Around	Around	At entry and exit of methods handling inbound connections.
`Connector_Around_Outbound`	Around	Around	At entry and exit of methods handling outbound connections.

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
`Connector_Around_Work`	Around	Around	At entry and exit of methods related to scheduling, starting and executing work items.
`Connector_Around_Tx`	Around	Around	At entry and exit of transaction register, unregister, start, rollback and commit methods.

JDBC_Before_Connection_Internal	Before	Stateless	Before calls to methods:
			`Driver.connect`
			`DataSource.getConnection`

JDBC_Before_Start_Internal	Before	Stateless	
JDBC_Before_Commit_Internal	Before	Stateless	
JDBC_Before_Rollback_Internal	Before	Stateless	
JDBC_Before_Statement_Internal	Before	Stateless	
JDBC_After_Connection_Internal	Before	Stateless	
JDBC_After_Start_Internal	After	Stateless	
JDBC_After_Commit_Internal	After	Stateless	
JDBC_After_Rollback_Internal	After	Stateless	
JDBC_After_Statement_Internal	After	Stateless	

The following table lists the diagnostic monitors 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 B-2 Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts

`Servlet_Before_Service`	Before	Stateless	At method entries of servlet/jsp methods:
			`HttpJspPage._jspService`
			`Servlet.service`
			`HttpServlet.doGet`
			`HttpServlet.doPost`
			`Filter.doFilter`

`Servlet_Before_Session`	Before	Stateless	Before calls to servlet methods:
			`HttpServletRequest.getSession`
			`HttpSession.setAttribute/putValue`
			`HttpSession.getAttribute/getValue`
			`HttpSession.removeAttribute/ removeValue`
			`HttpSession.invalidate`

`Servlet_Before_Tags`	Before	Stateless	Before calls to jsp methods:
			`Tag.doStartTag`
			`Tag.doEndTag`

`JNDI_Before_Lookup`	Before	Stateless	Before calls to `javax.naming.Context` lookup methods `Context.lookup*`
`JMS_Before_ TopicPublished`	Before	Stateless	Before call to methods: `TopicPublisher.publish`
`JMS_Before_MessageSent`	Before	Stateless	Before call to methods: `QueSender send`
`JMS_Before_ AsyncMessageReceived`	Before	Stateless	At entry of methods: `MessageListener.onMessage`
`JMS_Before_ SyncMessageReceived`	Before	Stateless	Before calls to methods: `MessageConsumer.receive*`

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JDBC_Before_ GetConnection	Before	Stateless	Before calls to methods: Driver.connect DataSource.getConnection
JDBC_Before_ CloseConnection	Before	Stateless	Before calls to methods: Connection.close
JDBC_Before_ CommitRollback	Before	Stateless	Before calls to methods: Connection.commit Connection.rollback
JDBC_Before_Statement	Before	Stateless	Before calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_Before_Execute	Before	Stateless	Before calls to methods: Statement.execute* PreparedStatement.execute*
EJB_Before_ SessionEjbMethods	Before	Stateless	At entry of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_Before_SessionEjb SemanticMethods	Before	Stateless	At entry of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_Before_ SessionEjbBusinessMethod s	Before	Stateless	At entry of all SessionBean methods, which are not standard ejb methods.

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_Before_ EntityEjbMethods	Before	Stateless	At entry of methods: EnitityBean.setEntityContext EnitityBean.unsetEntityContext EnitityBean.ejbRemove EnitityBean.ejbActivate EnitityBean.ejbPassivate EnitityBean.ejbLoad EnitityBean.ejbStore
EJB_Before_Entity EjbSemanticMethods	Before	Stateless	At entry of methods: EnitityBean.set* EnitityBean.get* EnitityBean.ejbFind* EnitityBean.ejbHome* EnitityBean.ejbSelect* EnitityBean.ejbCreate* EnitityBean.ejbPostCreate*
EJB_Before_EntityEjb BusinessMethods	Before	Stateless	At entry of all EntityBean methods, which are not standard ejb methods.
MDB_Before_Message Received	Before	Stateless	At entry of methods: MessageDrivenBean.onMessage
MDB_Before_Set MessageDrivenContext	Before	Stateless	At entry of methods: MessageDrivenBean.setMessage DrivenContext
MDB_Before_Remove	Before	Stateless	At entry of methods: MessageDrivenBean.ejbRemove
JTA_Before_Start	Before	Stateless	At entry of methods: UserTransaction.begin
JTA_Before_Commit	Before	Stateless	At entry of methods: UserTransaction.commit
JTA_Before_Rollback	Before	Stateless	At entry of methods: UserTransaction.rollback

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
Servlet_After_Service	After	Stateless	At method exits of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter
Servlet_After_Session	After	Stateless	After calls to servlet methods: HttpServletRequest.getSession HttpSession.setAttribute/putValue HttpSession.getAttribute/getValue HttpSession.removeAttribute/ removeValue HttpSession.invalidate
Servlet_After_Tags	After	Stateless	After calls to jsp methods: Tag.doStartTag Tag.doEndTag
JNDI_After_Lookup	After	Stateless	After calls to javax.naming.Context lookup methods: Context.lookup*
JMS_After_Topic Published	After	Stateless	After call to methods: TopicPublisher.publish
JMS_After_MessageSent	After	Stateless	After call to methods: QueSender send
JMS_After_ AsyncMessageReceived	After	Stateless	At exits of methods: MessageListener.onMessage
JMS_After_Sync MessageReceived	After	Stateless	After calls to methods: MessageConsumer.receive*
JDBC_After_Get Connection	After	Stateless	After calls to methods: Driver.connect DataSource.getConnection

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JDBC_After_CloseConnection	After	Stateless	After calls to methods: Connection.close
JDBC_After_Commit Rollback	After	Stateless	After calls to methods: Connection.commit Connection.rollback
JDBC_After_Statement	After	Stateless	After calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_After_Execute	After	Stateless	After calls to methods: Statement.execute* PreparedStatement.execute*
EJB_After_SessionEjbMethods	After	Stateless	At exits of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_After_SessionEjb SemanticMethods	After	Stateless	At exits of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_After_SessionEjb BusinessMethods	After	Stateless	At exits of all SessionBean methods, which are not standard ejb methods.
EJB_After_EntityEjb Methods	After	Stateless	At exits of methods: EnitityBean.setEntityContext EnitityBean.unsetEntityContext EnitityBean.ejbRemove EnitityBean.ejbActivate EnitityBean.ejbPassivate EnitityBean.ejbLoad EnitityBean.ejbStore

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_After_EntityEjb SemanticMethods	After	Stateless	At exits of methods: EnitityBean.set* EnitityBean.get* EnitityBean.ejbFind* EnitityBean.ejbHome* EnitityBean.ejbSelect* EnitityBean.ejbCreate* EnitityBean.ejbPostCreate*
EJB_After_EntityEjb BusinessMethods	After	Stateless	At exits of all EntityBean methods, which are not standard ejb methods.
MDB_After_Message Received	After	Stateless	At exits of methods: MessageDrivenBean.onMessage
MDB_After_SetMessageDriv enContext	After	Stateless	At exits of methods: MessageDrivenBean.setMessage DrivenContext
MDB_After_Remove	After	Stateless	At exits of methods: MessageDrivenBean.ejbRemove
JTA_After_Start	After	Stateless advice	At exits of methods: UserTransaction.begin
JTA_After_Commit	After	Stateless advice	At exits of methods: UserTransaction.commit
JTA_After_Rollback	After	Stateless advice	At exits of methods: UserTransaction.rollback
Servlet_Around_Service	Around	Around	At method entry and exits of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts

`Servlet_Around_Session`	Around	Around	Before and after calls to servlet methods:
			`HttpServletRequest.getSession`
			`HttpSession.setAttribute/putValue`
			`HttpSession.getAttribute/getValue`
			`HttpSession.removeAttribute/ removeValue`
			`HttpSession.invalidate`

`Servlet_Around_Tags`	Around	Around	Before and after calls to jsp methods:
			`Tag.doStartTag`
			`Tag.doEndTag`

`JNDI_Around_Lookup`	Around	Around	Before and after calls to `javax.naming.Context` lookup methods `Context.lookup*`
`JMS_Around_Topic Published`	Around	Around	Before and after call to methods: `TopicPublisher.publish`
`JMS_Around_Message Sent`	Around	Around	Before and after call to methods: `QueSender send`
`JMS_Around_Async MessageReceived`	Around	Around	At entry and exits of methods: `MessageListener.onMessage`
`JMS_Around_Sync MessageReceived`	Around	Around	Before and after calls to methods: `MessageConsumer.receive*`

`JDBC_Around_Get Connection`	Around	Around	Before and after calls to methods:
			`Driver.connect`
			`DataSource.getConnection`

`JDBC_Around_Close Connection`	Around	Around	Before and after calls to methods: `Connection.close`

`JDBC_Around_CommitRollback`	Around	Around	Before and after calls to methods:
			`Connection.commit`
			`Connection.rollback`

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JDBC_Around_ Statement	Around	Around	Before and after calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_Around_Execute	Around	Around	Before and after calls to methods: Statement.execute* PreparedStatement.execute*
EJB_Around_Session EjbMethods	Around	Around	At entry and exits of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_Around_SessionEjbSem anticMethods	Around	Around	At entry and exits of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_Around_SessionEjbBus inessMethods	Around	Around	At entry and exits of all SessionBean methods, which are not standard ejb methods.
EJB_Around_EntityEjbMeth ods	Around	Around	At exits of methods: EnitityBean.setEntityContext EnitityBean.unsetEntityContext EnitityBean.ejbRemove EnitityBean.ejbActivate EnitityBean.ejbPassivate EnitityBean.ejbLoad EnitityBean.ejbStore
EJB_Around_EntityEjb SemanticMethods	Around	Around	At entry and exits of methods: EnitityBean.set* EnitityBean.get* EnitityBean.ejbFind* EnitityBean.ejbHome* EnitityBean.ejbSelect* EnitityBean.ejbCreate* EnitityBean.ejbPostCreate*

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_Around_EntityEjb BusinessMethods	Around	Around	At entry and exits of all EntityBean methods that are not standard ejb methods.
MDB_Around_MessageReceiv ed	Around	Around	At entry and exits of methods: MessageDrivenBean.onMessage
MDB_Around_Set MessageDrivenContext	Around	Around	At entry and exits of methods: MessageDrivenBean.setMessage DrivenContext
MDB_Around_Remove	Around	Around	At entry and exits of methods: MessageDrivenBean.ejbRemove
JTA_Around_Start	Around	Around	At entry and exits of methods: UserTransaction.begin
JTA_Around_Commit	Around	Around	At entry and exits of methods: UserTransaction.commit
JTA_Around_Rollback	Around	Around	At entry and exits of methods: UserTransaction.rollback

Diagnostic Action Library

The Diagnostic Action Library includes the following actions:

- TraceAction
- DisplayArgumentsAction
- TraceElapsedTimeAction
- StackDumpAction
- ThreadDumpAction

These diagnostic actions can be used with the delegating monitors described in the previous tables. 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.

TraceAction

This action is a stateless action and is compatible with Before and After monitor types.

A `TraceAction` generates a trace event at affected location in the program execution.The following information is generated:

- Timestamp
- Context identifier from the diagnostic context which uniquely identifies the request
- Transaction identifier, if available
- User identity
- Action type, that is, `TraceAction`
- Domain
- Server name
- Instrumentation scope name (for example, application name)
- Diagnostic monitor name
- Location in code from where the action was called (class name, method name, etc.)
- Payload carried by the diagnostic context, if any

DisplayArgumentsAction

This action is a stateless action and is compatible with Before and After monitor types.

A `DisplayArgumentsAction` generates an instrumentation event at affected location in the program execution to capture method arguments or return value. The following information is generated:

When executed, this action causes an instrumentation event which will be dispatched to the events archive. When attached to *before* monitors, the instrumentation event will capture input arguments to the joinpoint (for example, method arguments). When attached to *after* monitors, the instrumentation event will capture the return value from the joinpoint. The even will carry the following information:

- Timestamp
- Context identifier from the diagnostic context which uniquely identifies the request

- Transaction identifier, if available
- User identity
- Action type, that is, `DisplayArgumentsAction`
- Domain
- Server name
- Instrumentation scope name (for example, application name)
- Diagnostic monitor name
- Location in code from where the action was called (class name, method name, etc.)
- Payload carried by the diagnostic context, if any
- Input arguments, if any, when attached to *before* monitors
- Return value, if any, when attached to *after* monitors

TraceElapsedTimeAction

This action is an Around action and is compatible with Around monitor types.

An `TraceElapsedTimeAction` generates an instrumentation event at affected location in the program execution to capture elapsed times.

When executed, this action captures the timestamps before and after the execution of associated joinpoint. It then computes the elapsed time by computing the difference. It generates an instrumentation event which is dispatched to the events archive. The elapsed time is stored as event payload. The event will carry the following information:

- Timestamp
- Context identifier from the diagnostic context which uniquely identifies the request
- Transaction identifier, if available
- User identity
- Action type, that is, `TraceElapsedTimeAction`
- Domain
- Server name

- Instrumentation scope name (for example, application name)
- Diagnostic monitor name
- Location in code from where the action was called (class name, method name, etc.)
- Payload carried by the diagnostic context, if any
- Elapsed time processing the joinpoint, as event payload

StackDumpAction

This action is a stateless action and is compatible with Before and After monitor types.

A `StackDumpAction` generates an instrumentation event at affected location in the program execution to capture stack dump.

When executed, this action generates an instrumentation event which is dispatched to the events archive. It captures the stack trace as event payload. The even will carry following information:

- Timestamp
- Context identifier from the diagnostic context which uniquely identifies the request
- Transaction identifier, if available
- User identity
- Action type, that is, `StackDumpAction`
- Domain
- Server name
- Instrumentation scope name (for example, application name)
- Diagnostic monitor name
- Location in code from where the action was called (class name, method name, etc.)
- Payload carried by the diagnostic context, if any
- Stack trace as event payload

ThreadDumpAction

This action is a stateless action and is compatible with Before and After monitor types.

A `ThreadDumpAction` generates an instrumentation event at affected location in the program execution to capture thread dump, if the underlying VM supports it.

When executed, this action generates an instrumentation event which is dispatched to the events archive. This action may be used only with the JRockit JVM. This action will be ignored when used with other JVMs. It captures the thread dump as event payload. The event will carry following information:

- Timestamp
- Context identifier from the diagnostic context which uniquely identifies the request
- Transaction identifier, if available
- User identity
- Action type, that is, `ThreadDumpAction`
- Domain
- Server name
- Instrumentation scope name (for example, application name)
- Diagnostic monitor name
- Location in code from where the action was called (class name, method name, etc.)
- Payload carried by the diagnostic context, if any
- Thread dump as event payload

APPENDIX C

WebLogic Scripting Tool Examples

The following examples use WLST and JMX to interact with WLDF components:

For information on running WebLogic Scripting Tool (`weblogic.WLST`) scripts, see "Running Scripts" in *Using the WebLogic Scripting Tool*. For information on developing JMX applications, see *Developing Manageable Applications with JMX*.

Dynamically Creating DyeInjection Monitors Example

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

- Connects to a server (boots the server first if necessary).
- Looks up or creates a WLDF System Resource.
- Creates the `DyeInjection` monitor.
- Sets the dye criteria.
- Enables the monitor.

- Saves and activates the configuration.
- Enables the Diagnostic Context feature 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 on deploying applications, see *Deploying Applications to WebLogic Server*.

Listing C-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
# - Look up or 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.
# <?xml version="1.0" encoding="UTF-8"?>
```

```
# <wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics"
# xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
#   <instrumentation>
#     <enabled>true</enabled>
#     <!-- Servlet Session Monitors -->
#     <wldf-instrumentation-monitor>
#       <name>Servlet_Before_Session</name>
#       <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>
#       <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>
##############################################################################
myDomainDirectory="domain"
url="t3://localhost:7001"
user="weblogic"
password="weblogic"
myServerName="myserver"
myDomain="mydomain"
props="weblogic.GenerateDefaultConfig=true,weblogic.RootDirectory="\
       +myDomainDirectory
```

```
try:
  connect(user,password,url)
except:
  startServer(adminServerName=myServerName,domainName=myDomain,
     username=user,password=password,systemProperties=props,
     domainDir=myDomainDirectory,block="true")
  connect(user,password,url)

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

# Look up or create the WLDF System resource.
wldfResourceName = "mywldf"
myWldfVar = cmo.lookupSystemResource(wldfResourceName)
if myWldfVar==None:
  print "Unable to find named resource,\
         creating WLDF System Resource: " + wldfResourceName
  myWldfVar=cmo.createWLDFSystemResource(wldfResourceName)

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

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

# Need to include newlines when setting properties
# on the DyeInjection monitor.
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 C-2) shows how to use the `weblogic.WLST` tool to configure a watch and a JMX notification using the WLDF Watch and Notification component. This script does the following:

- Connects to a server and boots the server first if necessary.
- Looks up/creates a WLDF system resource.
- Creates a watch and watch rule on the `ServerRuntimeMBean` for the `OpenSocketsCurrentCount` 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 C-8).
- The `demoHarvester.py` script, which registers the `OpenSocketsCurrentCount` attribute with the harvester for collection (see "MBean Registration and Data Collection Example" on page C-12).

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, type:

   ```
   javac JMXWatchNotificationListener.java
   ```

3. To run the `JMXWatchNotificationListener.class` file, type:

```
java JMXWatchNotificationListener
```

Note: Be sure the current directory is in your class path, so it will find the class file you just created.

4. 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 C-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
# - Look up or create a WLDF System Resource
# - Create a watch and watch rule on the ServerRuntimeMBean for the
#     "OpenSocketsCurrentCount" 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
#   "OpenSocketsCurrentCount" attribute with the harvester for collection.
# To see these work together:
# 1. Run the watch configuration script
#       java weblogic.WLST demoWatch.py
# 2. Compile and run the JMXWatchNotificationListener.java source code
#       javac JMXWatchNotificationListener.java
#        java JMXWatchNotificationListener
# 3. Run the demoHarvester.py script
#       java weblogic.WLST demoHarvester.py
# When the demoHarvester.py script runs, it triggers the
# JMXNotification for the watch configured in step 1.
```

```
###########################################################################
myDomainDirectory="domain"
url="t3://localhost:7001"
user="weblogic"
myServerName="myserver"
myDomain="mydomain"
props="weblogic.GenerateDefaultConfig=true\
       weblogic.RootDirectory="+myDomainDirectory

try:
  connect(user,user,url)
except:
  startServer(adminServerName=myServerName,domainName=myDomain,
     username=user,password=user,systemProperties=props,
     domainDir=myDomainDirectory,block="true")
  connect(user,user,url)

edit()
startEdit()

# Look up or create the WLDF System resource
wldfResourceName = "mywldf"
myWldfVar = cmo.lookupSystemResource(wldfResourceName)
if myWldfVar==None:
  print "Unable to find named resource"
  print "creating WLDF System Resource: " + wldfResourceName
  myWldfVar=cmo.createWLDFSystemResource(wldfResourceName)

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

cd("/WLDFSystemResources/mywldf/WLDFResource/mywldf/WatchNotification/mywl
df")
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 C-3 shows how to write a JMXWatchNotificationListener.

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

```
import javax.management.*;

import weblogic.diagnostics.watch.*;

import weblogic.diagnostics.watch.JMXWatchNotification;
import javax.management.Notification;
import javax.management.remote.JMXServiceURL;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXConnector;
import javax.naming.Context;
import java.util.Hashtable;
import weblogic.management.mbeanservers.runtime.RuntimeServiceMBean;

public class JMXWatchNotificationListener implements NotificationListener,
Runnable {

  private MBeanServerConnection rmbs = null;
  private String notifName = "myjmx";
  private int notifCount = 0;
  private String serverName = "myserver";
```

```
public JMXWatchNotificationListener(String serverName) {
}

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

public void handleNotification(Notification notif, Object handback) {
  synchronized (this) {
    try {
      if (notif instanceof JMXWatchNotification) {
        WatchNotification wNotif =
            ((JMXWatchNotification)notif).getExtendedInfo();
        notifCount++;
System.out.println("================================================");
        System.out.println("Notification name:    " +
             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());
System.out.println("================================================");
      }
    } catch (Throwable x) {
```

```
        System.out.println("Exception occurred processing JMX watch
              notification: " + notifName +"\n" + x);
        x.printStackTrace();
      }
    }
  }

  private void addNotificationHandler() throws Exception {

    /*
     * The JMX Watch notification listener registers with a Runtime MBean
     * that matches the name of the corresponding watch bean.
     * Each watch has its own Runtime MBean instance.
     */

    ObjectName oname =
      new ObjectName(
          "com.bea:ServerRuntime=" + serverName + ",Name=" +
          JMXWatchNotification.GLOBAL_JMX_NOTIFICATION_PRODUCER_NAME +
          ",Type=WLDFWatchJMXNotificationRuntime," +
          "WLDFWatchNotificationRuntime=WatchNotification," +
          "WLDFRuntime=WLDFRuntime"
          );
    System.out.println("Adding notification handler for: " +
      oname.getCanonicalName());
    rmbs.addNotificationListener(oname, this, null, null);
  }

  private void removeNotificationHandler(String name,
              NotificationListener list) throws Exception {
    ObjectName oname =
      new ObjectName(
          "com.bea:ServerRuntime=" + serverName + ",Name=" +
          JMXWatchNotification.GLOBAL_JMX_NOTIFICATION_PRODUCER_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 {
      String serverName = "myserver";
      if (args.length > 0)
        serverName = args[0];
```

```
        JMXWatchNotificationListener listener =
              new JMXWatchNotificationListener(serverName);
        System.out.println("Adding shutdown hook");
        Runtime.getRuntime().addShutdownHook(new Thread(listener));
        listener.register();
        // Sleep waiting for notifications
        Thread.sleep(Long.MAX_VALUE);
      } catch (Throwable e) {
        e.printStackTrace();
      } // end of try-catch
    } // end of main()
}
```

MBean 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 C-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
# - Look up or 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
#########################################################################
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 addTypeForInstance(harvester, mbeanInstance):
  typeName = "weblogic.management.runtime."\
              + mbeanInstance.getType() + "MBean"
  return addTypeByName(harvester, typeName, 1)

def addInstanceToHarvestedType(harvester, mbeanInstance):
  harvestedType = addTypeForInstance(harvester, mbeanInstance)
  currentInstances = PyList()
  currentInstances.extend(harvestedType.getHarvestedAttributes());
  on = mbeanInstance.getObjectName().getCanonicalName()
  print "Adding " + str(on) + " to set of harvested instances for type "\
        + harvestedType.getName()
  print "Current instances : " + str(currentInstances)
  for inst in currentInstances:
    if inst == on:
      print "Found " + on + " in existing set"
      return harvestedType
  # only get here if the target attribute is not in the set
  currentInstances.append(on)
  # convert the new list back to a Java String array
  newSet = jarray.array(currentInstances, java.lang.String)
  print "New instance set for type " + harvestedType.getName()\
        + ": " + str(newSet)
  harvestedType.setHarvestedInstances(newSet)
  return harvestedType

def addTypeByName(harvester, _typeName, knownType=0):
  ht=findHarvestedType(harvester, _typeName)
  if ht == None:
    print "Adding " + _typeName + " to harvestables collection for "\
```

```
            + harvester.getName()
    ht=harvester.createHarvestedType(_typeName)
    if knownType == 1:
        print "Setting known type attribute to true for " + _typeName
        ht.setKnownType(knownType)
  return ht;

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

########################################################################
# 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
########################################################################
myDomainDirectory="domain"
url="t3://localhost:7001"
user="weblogic"
myServerName="myserver"
myDomain="mydomain"
props="weblogic.GenerateDefaultConfig=true,weblogic.RootDirectory="\
        +myDomainDirectory
```

```
try:
  connect(user,user,url)
except:
  startServer(adminServerName=myServerName,domainName=myDomain,
     username=user,password=user,systemProperties=props,
     domainDir=myDomainDirectory,block="true")
  connect(user,user,url)

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

# Look up or create the WLDF System resource
wldfResourceName = "mywldf"
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(myServerName)
systemResource.addTarget(wldfServer)

# The harvester Jython wrapper maintains refs to
# the SystemResource objects
harvester.setSamplePeriod(5000)
harvester.setEnabled(1)

# add an instance-based RT MBean attribute for collection
serverRuntime()
cd("/")
addAttributeForInstance(harvester, cmo, "OpenSocketsCurrentCount")
```

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

# add a RT MBean type, all instances and attributes,
# with KnownType = "true"
addTypeByName(harvester,
        "weblogic.management.runtime.WLDFInstrumentationRuntimeMBean", 1)
addTypeByName(harvester,
       "weblogic.management.runtime.WLDFWatchNotificationRuntimeMBean", 1)
addTypeByName(harvester,
        "weblogic.management.runtime.WLDFHarvesterRuntimeMBean", 1)

try:
    save()
    activate(block="true")
except:
    print "Error while trying to save and/or activate."
    dumpStack()

# display the data
displayHarvestedTypes(harvester)

disconnect()
exit()
```

APPENDIX D

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 Framework 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 Framework 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 Framework 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 Framework 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 Framework 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 Framework. 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 Diagnostic 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 Framework 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 Framework 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 Framework, 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 `MBeanInfo`s. 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 Framework 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 Framework 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.

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

The `DyeInjection` monitor 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 Framework 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 diagnostic image captures when significant failures occur and you can manually initiate diagnostic 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 Framework to take diagnostic actions. Weaving time affects both the load time for server-level instrumented classes and application deployment time for application-level classes.

Index