

Oracle® Fusion Middleware

Configuring and Using the Diagnostics Framework for Oracle
WebLogic Server

12c Release 1 (12.1.1)

E24426-02

January 2012

Documentation for developers and administrators that describes how to configure and use the WebLogic Diagnostics Framework (WLDF), a monitoring and diagnostic framework that defines and implements a set of services that run within WebLogic Server processes and that participate in the standard server life cycle.

Oracle Fusion Middleware Configuring and Using the Diagnostics Framework for Oracle WebLogic Server, 12c Release 1 (12.1.1)

E24426-02

Copyright © 2007, 2012, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	ix
Documentation Accessibility	ix
Conventions	ix
1 Introduction and Roadmap	
1.1 What Is the WebLogic Diagnostics Framework?	1-1
1.2 Document Scope and Audience.....	1-2
1.3 Guide to This Document.....	1-2
1.4 Related Documentation.....	1-4
1.5 Samples and Tutorials.....	1-4
1.5.1 Avitek Medical Records Application (MedRec) and Tutorials.....	1-4
1.5.2 WLDF Samples Available for Download	1-4
1.6 New and Changed Features in this Release.....	1-4
2 Overview of the WLDF Architecture	
2.1 Overview of the WebLogic Diagnostics Framework.....	2-1
2.2 Data Creation, Collection, and Instrumentation	2-2
2.3 Archive	2-3
2.4 Watch and Notification	2-4
2.5 Data Accessor	2-5
2.6 Monitoring Dashboard and Request Performance Pages	2-5
2.6.1 Monitoring Dashboard	2-5
2.6.2 Diagnostics Request Performance Page	2-6
2.7 Diagnostic Image Capture	2-6
2.8 How It All Fits Together	2-7
3 Using WLDF with Oracle JRockit Flight Recorder	
3.1 About Oracle JRockit Flight Recorder.....	3-1
3.2 Key Features of WLDF Integration with JRockit Flight Recorder	3-2
3.3 JRockit Flight Recorder Use Cases	3-4
3.3.1 Diagnosing a Critical Failure — The "Black Box"	3-4
3.3.2 Profiling During Performance Testing or in Production	3-4
3.3.3 Real-time Application Diagnostics and Reporting (RADAR).....	3-4
3.4 Obtaining the JRockit Flight Recording File	3-5
3.5 Analyzing Flight Recorder Data in JRockit Mission Control	3-6

3.5.1	JFR Graphical User Interface.....	3-6
3.5.2	Analyzing Execution Flow — A Sample Walkthrough	3-7
3.5.2.1	Displaying Event Data for a Product Subcomponent	3-8
3.5.2.2	Viewing the Event Log to Display Details.....	3-8
3.5.2.3	Tracking Execution Flow by Analyzing an Operative Set.....	3-10
3.5.2.4	Expanding the Operative Set and Viewing Correlated Diagnostic Data	3-12
3.5.3	Changing the Location of Temporary JFR Files	3-13

4 Understanding WLDF Configuration

4.1	Configuration MBeans and XML.....	4-1
4.2	Tools for Configuring WLDF	4-2
4.3	How WLDF Configuration Is Partitioned	4-2
4.3.1	Server-Level Configuration.....	4-2
4.3.2	Application-Level Configuration	4-3
4.4	Configuring Diagnostic Image Capture and Diagnostic Archives.....	4-3
4.5	Configuring Diagnostic Image Capture for JRockit Flight Recorder	4-4
4.6	Configuring Diagnostic System Modules	4-4
4.6.1	The Diagnostic System Module and Its Resource Descriptor	4-5
4.6.2	Referencing the Diagnostics System Module from Config.xml.....	4-5
4.6.3	The <i>DIAG_MODULE.xml</i> Resource Descriptor Configuration.....	4-6
4.6.4	Managing Diagnostic System Modules.....	4-7
4.6.5	More Information About Configuring Diagnostic System Resources	4-7
4.7	Configuring Diagnostic Modules for Applications	4-7
4.8	WLDF Configuration MBeans and Their Mappings to XML Elements.....	4-8

5 Configuring and Capturing Diagnostic Images

5.1	How to Initiate Image Captures	5-1
5.2	Configuring Diagnostic Image Captures.....	5-1
5.2.1	Configuring WLDF Diagnostic Volume.....	5-2
5.2.2	WLST Commands for Generating an Image Capture	5-2
5.3	How Diagnostic Image Capture Is Persisted in the Server's Configuration	5-3
5.4	Content of the Captured Image File.....	5-3
5.4.1	Data Included in the Diagnostics Image Capture File	5-4
5.4.2	WLST Online Commands for Downloading Diagnostics Image Captures	5-5

6 Configuring Diagnostic Archives

6.1	Configuring the Archive	6-1
6.2	Configuring a File-Based Store	6-1
6.3	Configuring a JDBC-Based Store	6-2
6.3.1	Creating WLDF Tables in the Database	6-2
6.3.2	Configuring JDBC Resources for WLDF	6-3
6.4	Retiring Data from the Archives.....	6-4
6.4.1	Configuring Data Retirement at the Server Level.....	6-4
6.4.2	Configuring Age-Based Data Retirement Policies for Diagnostic Archives	6-4
6.4.3	Sample Configuration	6-5

7 Configuring the Harvester for Metric Collection

7.1	Harvesting, Harvestable Data, and Harvested Data	7-1
7.2	Harvesting Data from the Different Harvestable Entities	7-2
7.3	Configuring the Harvester	7-2
7.3.1	Configuring the Harvester Sampling Period.....	7-3
7.3.2	Configuring the Types of Data to Harvest.....	7-3
7.3.3	Specifying Type Names for WebLogic Server MBeans and Custom MBeans.....	7-4
7.3.4	Harvesting from the DomainRuntime MBeanServer.....	7-4
7.3.5	When Configuration Settings Are Validated.....	7-5
7.3.6	Sample Configurations for Different Harvestable Types	7-5

8 Configuring Watches and Notifications

8.1	Watches and Notifications.....	8-1
8.2	Overview of Watch and Notification Configuration	8-2
8.3	Sample Watch and Notification Configuration.....	8-3

9 Configuring Watches

9.1	Types of Watches	9-1
9.2	Configuration Options Shared by All Types of Watches.....	9-2
9.3	Configuring Harvester Watches	9-2
9.4	Configuring Log Watches.....	9-4
9.5	Configuring Instrumentation Watches	9-5
9.6	Defining Watch Rule Expressions	9-5

10 Configuring Notifications

10.1	Types of Notifications	10-1
10.2	Configuring JMX Notifications	10-2
10.3	Configuring JMS Notifications.....	10-2
10.4	Configuring SNMP Notifications	10-3
10.5	Configuring SMTP Notifications	10-4
10.6	Configuring Image Notifications.....	10-5

11 Configuring Instrumentation

11.1	Concepts and Terminology	11-1
11.1.1	Instrumentation Scope	11-2
11.1.2	Configuration and Deployment	11-2
11.1.3	Joinpoints, Pointcuts, and Diagnostic Locations.....	11-2
11.1.4	Diagnostic Monitor Types	11-2
11.1.5	Diagnostic Actions.....	11-4
11.2	Instrumentation Configuration Files	11-4
11.3	XML Elements Used for Instrumentation	11-5
11.3.1	<Instrumentation> XML Elements	11-6
11.3.2	<wldf-instrumentation-monitor> XML Elements	11-7
11.3.3	Mapping <wldf-instrumentation-monitor> XML Elements to Monitor Types	11-10
11.4	Configuring Server-Scoped Instrumentation	11-10

11.5	Configuring Application-Scoped Instrumentation.....	11-12
11.5.1	Comparing System-Scoped to Application-Scoped Instrumentation.....	11-12
11.5.2	Overview of the Steps Required to Instrument an Application	11-13
11.5.3	Creating a Descriptor File for a Delegating Monitor	11-14
11.5.4	Creating a Descriptor File for a Custom Monitor	11-14
11.5.4.1	Defining Pointcuts for Custom Monitors.....	11-15
11.5.4.2	Annotation-based Pointcuts.....	11-17
11.6	Creating Request Performance Data	11-18

12 Configuring the DyelInjection Monitor to Manage Diagnostic Contexts

12.1	Contents, Life Cycle, and Configuration of a Diagnostic Context.....	12-1
12.1.1	Context Life Cycle and the Context ID	12-2
12.1.2	Dyes, Dye Flags, and Dye Vectors	12-2
12.1.3	Where Diagnostic Context Is Configured	12-3
12.2	Overview of the Process.....	12-3
12.3	Configuring the Dye Vector via the DyelInjection Monitor	12-4
12.3.1	Dyes Supported by the DyelInjection Monitor.....	12-5
12.3.2	PROTOCOL Dye Flags	12-6
12.3.3	THROTTLE Dye Flag	12-6
12.3.4	When Diagnostic Contexts Are Created.....	12-7
12.4	Configuring Delegating Monitors to Use Dye Filtering.....	12-7
12.5	How Dye Masks Filter Requests to Pass to Monitors.....	12-9
12.5.1	Dye Filtering Example	12-10
12.6	Using Throttling to Control the Volume of Instrumentation Events	12-11
12.6.1	Configuring the THROTTLE Dye	12-11
12.6.2	How Throttling is Handled by Delegating and Custom Monitors	12-13
12.7	Using weblogic.diagnostics.context	12-13

13 Accessing Diagnostic Data With the Data Accessor

13.1	Data Stores Accessed by the Data Accessor.....	13-1
13.2	Accessing Diagnostic Data Online	13-2
13.2.1	Accessing Data Using the Administration Console	13-2
13.2.2	Accessing Data Programmatically Using Run-Time MBeans.....	13-2
13.2.3	Using WLST to Access Diagnostic Data Online.....	13-3
13.2.4	Using the WLDF Query Language with the Data Accessor.....	13-3
13.3	Accessing Diagnostic Data Offline	13-3
13.4	Accessing Diagnostic Data Programmatically.....	13-3
13.5	Resetting the System Clock Can Affect How Data Is Archived and Retrieved.....	13-8

14 Deploying WLDF Application Modules

14.1	Deploying a Diagnostic Module as an Application-Scoped Resource.....	14-1
14.2	Using Deployment Plans to Dynamically Control Instrumentation Configuration.....	14-2
14.3	Using a Deployment Plan: Overview	14-3
14.4	Creating a Deployment Plan Using weblogic.PlanGenerator.....	14-4
14.5	Sample Deployment Plan for Diagnostics.....	14-4
14.6	Enabling Java HotSwap	14-5

14.7	Deploying an Application with a Deployment Plan	14-5
14.8	Updating an Application with a Modified Plan	14-6
15	Using the Monitoring Dashboard	
15.1	Running the Monitoring Dashboard.....	15-1
15.2	Scope of the Diagnostic Information Displayed.....	15-1
15.3	About the Monitoring Dashboard Interface	15-2
15.3.1	View List	15-3
15.3.2	Metric Browser	15-4
15.3.3	View Display Panel	15-6
15.4	Understanding How Metrics Are Collected and Presented.....	15-8
15.4.1	About Metrics and Chart Types	15-8
15.4.1.1	Current Time Range Charts	15-8
15.4.1.2	Custom Time Range Charts	15-9
15.4.2	Sequence in which Metrics Data is Displayed.....	15-9
15.4.3	Notes about Metric Data Retention.....	15-10
15.5	The Parts of a Chart	15-10
16	Configuring and Using WLDF Programmatically	
16.1	How WLDF Generates and Retrieves Data.....	16-1
16.2	Mapping WLDF Components to Beans and Packages.....	16-2
16.3	Programming Tools	16-4
16.3.1	Configuration and Run-Time APIs.....	16-5
16.3.1.1	Configuration APIs	16-5
16.3.1.2	Run-Time APIs.....	16-6
16.4	WLDF Packages	16-6
16.5	Programming WLDF: Examples.....	16-7
16.5.1	Example: DiagnosticContextExample.java	16-7
16.5.2	Example: HarvesterMonitor.java	16-8
16.5.2.1	Notification Listeners.....	16-8
16.5.2.2	HarvesterMonitor.java.....	16-9
16.5.3	Example: JMXAccessorExample.java	16-13
A	WLDF Query Language	
A.1	Components of a Query Expression	A-1
A.2	Supported Operators.....	A-1
A.3	Operator Precedence	A-2
A.4	Numeric Relational Operations Supported on String Column Types.....	A-3
A.5	Supported Numeric Constants and String Literals.....	A-3
A.6	About Variables in Expressions	A-4
A.7	Creating Watch Rule Expressions	A-4
A.7.1	Creating Log Event Watch Rule Expressions	A-4
A.7.2	Creating Instrumentation Event Watch Rule Expressions	A-5
A.7.3	Creating Harvester Watch Rule Expressions.....	A-6
A.8	Creating Data Accessor Queries	A-6
A.8.1	Data Store Logical Names	A-7

A.8.2	Data Store Column Names.....	A-7
A.9	Creating Log Filter Expressions.....	A-8
A.10	Building Complex Expressions.....	A-9

B WLDF Instrumentation Library

B.1	Diagnostic Monitor Library.....	B-1
B.2	Diagnostic Action Library.....	B-9
B.2.1	TraceAction.....	B-10
B.2.2	DisplayArgumentsAction.....	B-11
B.2.3	TraceElapsedTimeAction.....	B-11
B.2.4	TraceMemoryAllocationAction.....	B-12
B.2.5	StackDumpAction.....	B-12
B.2.6	ThreadDumpAction.....	B-13
B.2.7	MethodInvocationStatisticsAction.....	B-14
B.2.7.1	Instrumenting an Application with MethodInvocationStatisticsAction and Querying the Results.....	B-15
B.2.7.1.1	Configuring the Custom Monitor to Use MethodInvocationStatisticsAction.....	B-15
B.2.7.1.2	Using WLST to Query Method Performance Statistics.....	B-18
B.2.7.2	Configuring the Harvester to Collect MethodInvocationStatisticsAction Data.....	B-20
B.2.7.3	Configuring Watch Rules Based on MethodInvocationStatistics Metrics.....	B-21
B.2.7.4	Using JMX to Collect Data.....	B-22
B.2.8	MethodMemoryAllocationStatisticsAction.....	B-22

C Using Wildcards in Expressions

C.1	Using Wildcards in Harvester Instance Names.....	C-1
C.1.1	Examples.....	C-1
C.2	Specifying Complex and Nested Harvester Attributes.....	C-2
C.2.1	Examples.....	C-3
C.3	Using the Accessor with Harvested Complex or Nested Attributes.....	C-4
C.4	Using Wildcards in Watch Rule Instance Names.....	C-4
C.5	Specifying Complex Attributes in Harvester Watch Rules.....	C-5

D WebLogic Scripting Tool Examples

D.1	Example: Dynamically Creating DyeInjection Monitors.....	D-1
D.2	Example: Configuring a Watch and a JMX Notification.....	D-3
D.3	Example: Writing a JMXWatchNotificationListener Class.....	D-5
D.4	Example: Registering MBeans and Attributes For Harvesting.....	D-8
D.5	Example: Setting the WLDF Diagnostic Volume.....	D-11
D.6	Example: Capturing a Diagnostic Image.....	D-11
D.7	Example: Retrieving a JFR File from a Diagnostic Image Capture.....	D-13

Preface

This preface describes the document accessibility features and conventions used in this guide—*Configuring and Using the Oracle WebLogic Diagnostics Framework*.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Introduction and Roadmap

This document describes the WebLogic Diagnostics Framework (WLDF), a monitoring and diagnostic framework that defines and implements a set of services that run within WebLogic Server processes 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.

The following sections describe the contents and audience for this guide—*Configuring and Using the WebLogic Diagnostics Framework*:

- [What Is the WebLogic Diagnostics Framework?](#)
- [Document Scope and Audience](#)
- [Guide to This Document](#)
- [Related Documentation](#)
- [Samples and Tutorials](#)
- [New and Changed Features in this Release](#)

1.1 What Is the WebLogic Diagnostics Framework?

WLDF includes several components for collecting and analyzing data:

- **Integration with Oracle JRockit**—If WebLogic Server is configured with JRockit, WLDF can generate diagnostic information about WebLogic Server that is captured in the JRockit Flight Recording file.
- **Diagnostic Image Capture**—Creates a diagnostic snapshot from the server that can be used for post-failure analysis. The diagnostic image capture includes JRockit Flight Recorder data, if it is available, that can be viewed in JRockit Mission Control.
- **Archive**—Captures and persists data events, log records, and metrics from server instances and applications.
- **Instrumentation**—Adds diagnostic 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 associating a diagnostic *context* with requests so they can be tracked as they flow through the system. The WebLogic Server Administration Console includes a Request Performance page, which shows real-time and historical views of method performance information that has been captured through the WLDF

instrumentation capabilities, serving as a tool that can help identify performance problems in applications.

- **Harvester**—Captures metrics from run-time MBeans, including WebLogic Server MBeans and custom MBeans, which can be archived and later accessed for viewing historical data.
- **Watches and Notifications**—Provides the means for monitoring server and application states and sending notifications based on criteria set in the watches.
- **Monitoring Dashboard**—Graphically presents the current and historical operating state of WebLogic Server and hosted applications. The Monitoring Dashboard, which is accessed from the WebLogic Server Administration Console, provides a set of tools for organizing and displaying diagnostic data into views, which surface some of the more critical run-time WebLogic Server performance metrics and the change in those metrics over time
- **Logging services**—Manage logs for monitoring server, subsystem, and application events. The WebLogic Server logging services are documented separately from the rest of the WebLogic Diagnostics Framework. See *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

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 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.

1.2 Document Scope and Audience

This document describes and 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.

1.3 Guide to This Document

This document is organized as follows:

- This chapter, "Introduction and Roadmap," provides an overview of WLDF components and describes the audience for this guide.
- [Chapter 2, "Overview of the WLDF Architecture,"](#) provides a high-level view of the WLDF architecture.
- [Chapter 3, "Using WLDF with Oracle JRockit Flight Recorder,"](#) describes the WLDF integration features with JRockit Flight Recorder, describes basic usage scenarios, and provides a sample walkthrough of using JRockit Mission Control to examine WebLogic Server events captured in a JRockit Flight Recorder file.

- [Chapter 4, "Understanding WLDF Configuration,"](#) provides an overview of how WLDF features are configured for servers and applications.
- [Chapter 5, "Configuring and Capturing Diagnostic Images,"](#) describes how to configure and use the WLDF Diagnostic Image Capture component to capture a snapshot of significant server configuration settings and the server state.
- [Chapter 6, "Configuring Diagnostic Archives,"](#) describes how to configure and use the WLDF Diagnostic Archive component to persist diagnostic data to a file store or database.
- [Chapter 7, "Configuring the Harvester for Metric Collection,"](#) describes how to configure and use the WLDF Harvester component to harvest metrics from run-time MBeans, including WebLogic Server MBeans and custom MBeans.
- [Chapter 8, "Configuring Watches and Notifications,"](#) provides an overview of WLDF watches and notifications.
- [Chapter 9, "Configuring Watches,"](#) describes how to configure watches to monitor server instances and applications for specific conditions and send notifications when those conditions are met.
- [Chapter 10, "Configuring Notifications,"](#) describes how to configure notifications that can be triggered by watches.
- [Chapter 11, "Configuring Instrumentation,"](#) describes how to add diagnostic instrumentation code to WebLogic Server classes and to the classes of applications running on the server.
- [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts,"](#) describes how to use the DyeInjection monitor and how to use dye filtering with diagnostic monitors.
- [Chapter 13, "Accessing Diagnostic Data With the Data Accessor,"](#) tells how to use the WLDF Data Accessor component to retrieve diagnostic data.
- [Chapter 14, "Deploying WLDF Application Modules,"](#) explains how to configure and manage instrumentation for an application as a diagnostics application module.
- [Chapter 15, "Using the Monitoring Dashboard,"](#) explains how to graphically present the current and historical operating state of WebLogic Server and hosted applications using, in part, diagnostic data captured by WLDF.
- [Chapter 16, "Configuring and Using WLDF Programmatically,"](#) 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, constructing watch rules, and constructing 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, "Using Wildcards in Expressions,"](#) discusses how to use wildcards in WLDF expressions.
- [Appendix D, "WebLogic Scripting Tool Examples,"](#) provides examples of how to perform WLDF monitoring and diagnostic activities using the WebLogic Scripting Tool.
- ["Glossary"](#) is a glossary of terms used in WLDF.

1.4 Related Documentation

- *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server* describes how to use WLDF logging services to monitor server, subsystem, and application events.
- "Configure the WebLogic Diagnostics Framework" in the *Oracle WebLogic Server Administration Console Online Help* describes how to use the visual tools in the WebLogic Administration Console to configure WLDF.
- The WLDF system resource descriptor conforms to the `weblogic-diagnostics.xsd` schema, available at <http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-diagnostics.xsd>.

1.5 Samples and Tutorials

In addition to this document, we provide a variety of samples and tutorials that show WLDF configuration and use.

1.5.1 Avitek Medical Records Application (MedRec) and Tutorials

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

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

1.5.2 WLDF Samples Available for Download

Additional WLDF samples for download can be found at <http://www.oracle.com/technetwork/indexes/samplecode/index.html>. These examples are distributed as .zip files that you can unzip into an existing WebLogic Server samples directory structure. These samples include Oracle-certified ones, as well as samples submitted by fellow developers.

1.6 New and Changed Features in this Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

Overview of the WLDF Architecture

This chapter describes the components in the WebLogic Diagnostics Framework (WLDF) architecture that work together to collect, archive, and access diagnostic information about a WebLogic Server instance and the applications it hosts.

Note: Concepts are presented in this section in a way to help you understand how WLDF works. Some of this differs from the way WLDF is surfaced in its configuration and run-time APIs and in the WebLogic Server Console. If you want to start configuring and using WLDF right away, you can safely skip this discussion and start with [Chapter 4, "Understanding WLDF Configuration."](#)

This chapter includes the following sections:

- [Overview of the WebLogic Diagnostics Framework](#)
- [Data Creation, Collection, and Instrumentation](#)
- [Archive](#)
- [Watch and Notification](#)
- [Data Accessor](#)
- [Monitoring Dashboard and Request Performance Pages](#)
- [Diagnostic Image Capture](#)
- [How It All Fits Together](#)

2.1 Overview of the WebLogic Diagnostics Framework

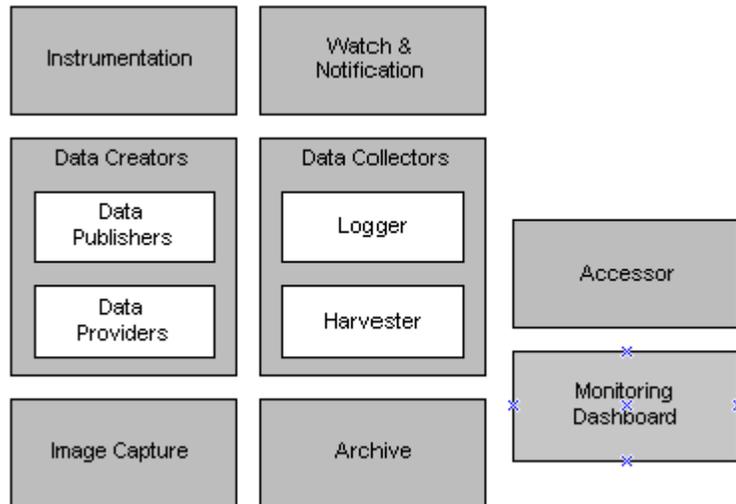
WLDF consists of the following:

- Data creators (data publishers and data providers that are distributed across WLDF components)
- Data collectors (the Logger and the Harvester components)
- Archive component
- Accessor component
- Instrumentation component
- Watch and Notification component
- Image Capture component

- Monitoring Dashboard

Data creators generate diagnostic data that is consumed by the Logger and the Harvester. Those components coordinate with the Archive to persist the data, and they coordinate with the Watch and Notification subsystem to provide automated monitoring. The Accessor interacts with the Logger and the Harvester to expose current diagnostic data and with the Archive to present historical data. The Image Capture facility provides the means for capturing a diagnostic snapshot of a key server state. The relationship among these components is shown in [Figure 2-1](#).

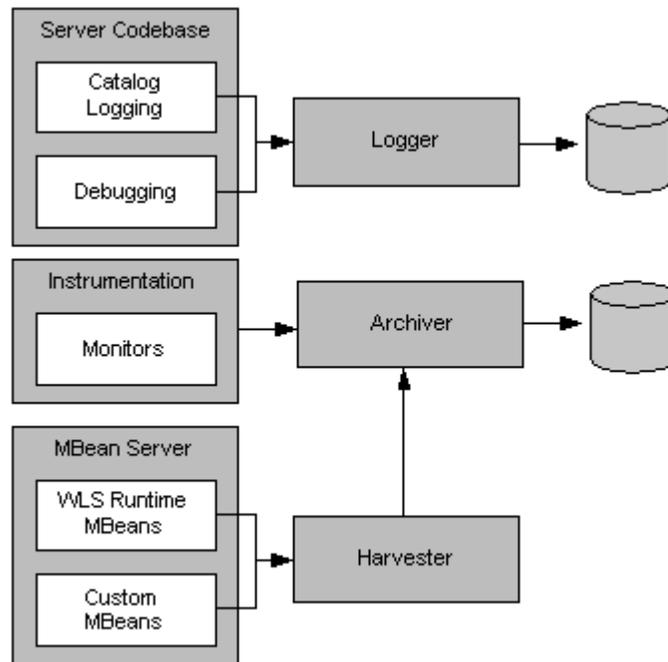
Figure 2-1 Major WLDF Components



All of the framework components operate at the server level and are only aware of server scope. All the components exist entirely within the server process and participate in the standard server lifecycle. All artifacts of the framework are configured and stored on a per server basis.

2.2 Data Creation, Collection, and Instrumentation

Diagnostic data is collected from a number of sources. These sources can be logically classified as either *data providers*, data creators that are sampled at regular intervals to harvest current values, or *data publishers*, data creators that synchronously generate events. Data providers and data publishers are distributed across components, and the generated data can be collected by the Logger and/or by the Harvester, as shown in [Figure 2-2](#), and explained below.

Figure 2–2 Relationship of Data Creation Components to Data Collection Components

Invocations of the server logging infrastructure serve as inline data publishers, and the generated data is collected as events. (The logging infrastructure can be invoked through the catalog infrastructure, the debugging model, or directly through the Logger.)

The Instrumentation component creates monitors and inserts them at well-defined points in the flow of execution. These monitors publish data directly to the Archive.

Components registered with the MBean Server may also make themselves known as data providers by registering with the Harvester. Collected data is then exposed to both the Watch and Notification system for automated monitoring and to the Archive for persistence.

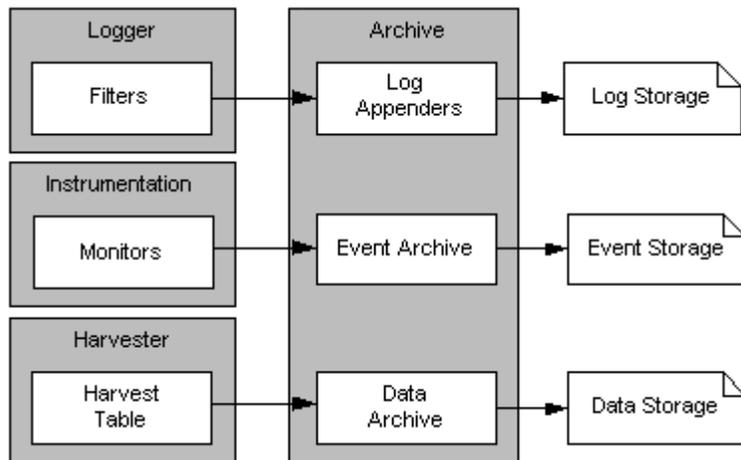
2.3 Archive

The past state is often critical in diagnosing faults in a system. This requires that the state be captured and archived for future access, creating a historical archive. In WLDF, the Archive meets this need with several persistence components. Both events and harvested metrics can be persisted and made available for historical review.

Traditional logging information, which is human readable and intended for inclusion in the server log, is persisted through the standard logging appenders. New event data that is intended for system consumption is persisted into an event store using an event archiver. Metric data is persisted into a data store using a data archiver. The relationship of the Archive to the Logger and the Harvester is shown in [Figure 2–3](#).

The Archive provides access interfaces so that the Accessor may expose any of the persisted historical data.

Figure 2-3 Relationship of the Archive to the Logger and the Harvester

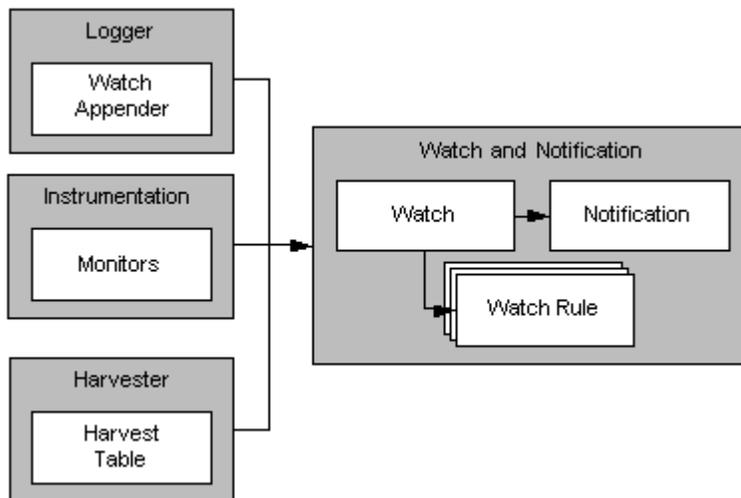


2.4 Watch and Notification

The Watch and Notification system can be used to create automated monitors that observe specific diagnostic states and send notifications based on configured rules.

A watch rule can monitor log data, event data from the Instrumentation component, or metric data from a data provider that is harvested by the Harvester. The Watch Manager is capable of managing watches that are composed of a number of watch rules. These relationships are shown in [Figure 2-4](#).

Figure 2-4 Relationship of the Logger and the Harvester to the Watch and Notification System



One or more notifications can be configured for use by a watch. By default, every watch logs an event in the server log. SMTP, SNMP, JMX, and JMS notifications are also supported.

2.5 Data Accessor

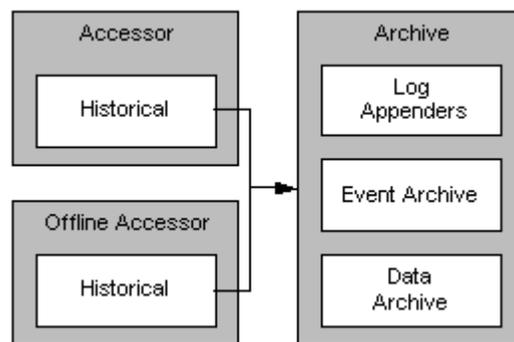
The Accessor provides access to all the data collected by WLDF, including log, event, and metric data. The Accessor interacts with the Archive to get historical data including logged event data and persisted metrics.

When accessing data in a running server, a JMX-based access service is used. The Accessor provides for data lookup by type, by component, and by attribute. It permits time-based filtering and, in the case of events, filtering by severity, source and content.

Tools may need to access data that was persisted by a currently inactive server. In this case, an offline Accessor is also provided. You can use it to export archived data to an XML file for later access. To use the Accessor in this way, you must use the WebLogic Scripting Tool (WLST) and must have physical access to the machine.

The relationship of the Accessor to the Harvester and the Archive is shown in [Figure 2-5](#).

Figure 2-5 Relationship of the Online and Offline Accessors to the Archive



2.6 Monitoring Dashboard and Request Performance Pages

WLDF provides two web pages from which diagnostic data is displayed visually:

- [Monitoring Dashboard](#)
- [Diagnostics Request Performance Page](#)

2.6.1 Monitoring Dashboard

The Monitoring Dashboard displays the current and historical operating state of WebLogic Server and hosted applications by providing visualizations of metric run-time MBean attributes, which surface some of the more critical run-time performance metrics and the change in those metrics over time. Historical operating state is represented by collected metrics that have been persisted into the Archive. To view collected metrics from the Archive, you must configure the Harvester to capture the data you want to monitor.

The Monitoring Dashboard displays metric information in a series of views. A view is a collection of one or more charts that display metrics. The Monitoring Dashboard includes a predefined set of built-in views of available run-time metrics for all running WebLogic Server instances in the domain. Built-in views surface some of the more critical run-time WebLogic Server performance metrics and serve as examples of the Monitoring Dashboard's graphic capabilities.

Custom views are available only to the user who creates them. Custom views are automatically persisted and can be accessed again when you restart the Monitoring Dashboard sessions.

For more information, see [Chapter 15, "Using the Monitoring Dashboard."](#)

2.6.2 Diagnostics Request Performance Page

The Diagnostics Request Performance page of the WebLogic Server Administration Console shows real-time and historical views of method performance information that is captured using the Instrumentation component. To view request performance information, you must first configure the Instrumentation component to make that data available.

For more information, see [Section 11.6, "Creating Request Performance Data."](#)

2.7 Diagnostic Image Capture

Diagnostic Image Capture support gathers the most common sources of the key server state used in diagnosing problems. It packages that state into a single artifact which can be made available to support technicians, as shown in [Figure 2–6](#). The diagnostic image is in essence a diagnostic snapshot or dump from the server, analogous to a UNIX "core" dump.

If WebLogic Server is configured with Oracle JRockit, and JRockit Flight Recorder is not disabled, the diagnostic image capture includes all available JRockit Flight Recorder data from all producers. Furthermore, if WLDF is configured to generate WebLogic Server diagnostic information captured by JRockit Flight Recorder, the JFR file includes that information as well. The JFR file can be extracted from the diagnostic image capture and viewed in JRockit Mission Control. See [Chapter 3, "Using WLDF with Oracle JRockit Flight Recorder."](#)

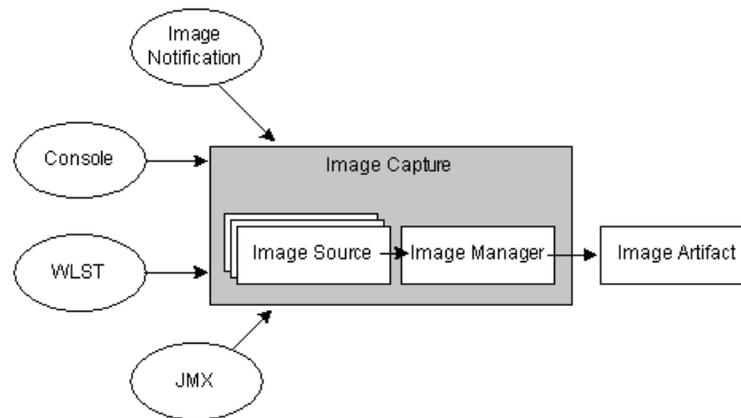
Image Capture support includes:

- On-demand capture, which is the creation of a diagnostic image capture by means of an operation or command issued from the WebLogic Server Administration Console, WLST script, or JMX application
- Image notification, which is automatically creating a diagnostic image capture in response to the triggering of an associated Harvester watch, Log watch, or Instrumentation watch rule. For example, a Harvester watch that monitors run-time MBean attributes in a running server can trigger an image notification if the metrics harvested from specific run-time MBean instances indicate a performance issue. Data in the diagnostic image capture can be analyzed to determine the likely causes of the issue.

For more information, see:

- [Configuring and Capturing Diagnostic Images](#)
- [Configuring Image Notifications](#)

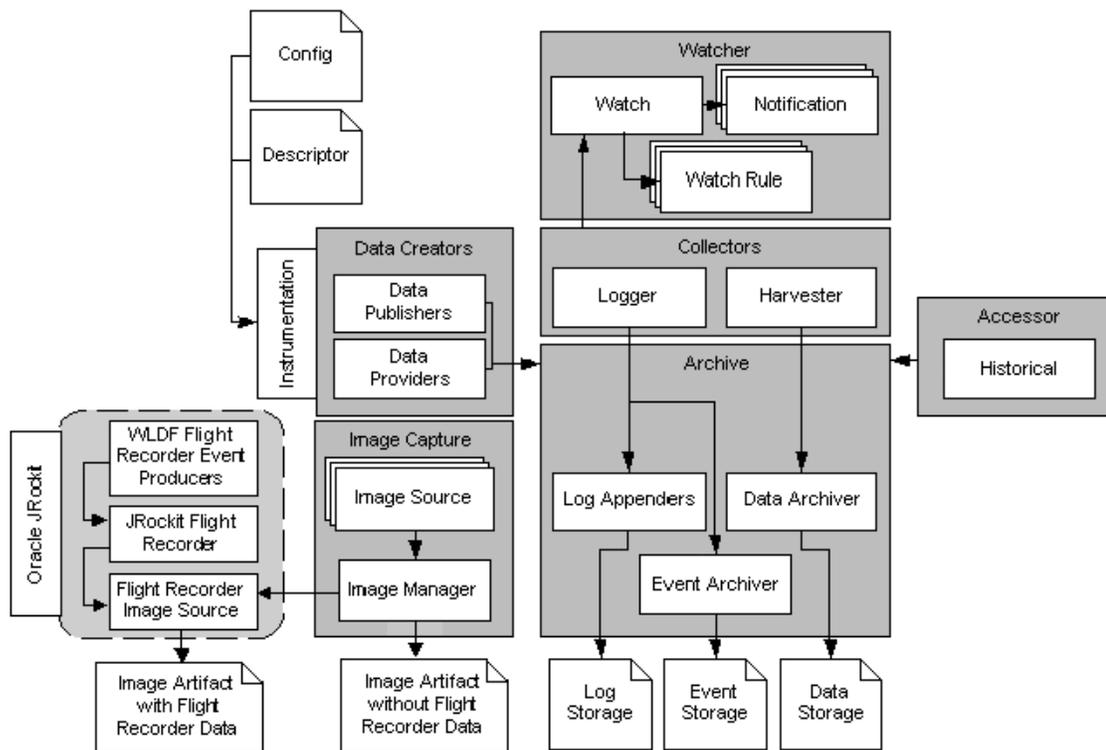
Figure 2-6 Diagnostic Image Capture



2.8 How It All Fits Together

Figure 2-7 shows how all the parts of WLDF fit together.

Figure 2-7 Overall View of the WebLogic Diagnostics Framework



Using WLDF with Oracle JRockit Flight Recorder

This chapter describes the integration points that the WebLogic Diagnostics Framework (WLDF) provides with JRockit Flight Recorder. WebLogic Server events can optionally be propagated to the Flight Recorder for inclusion in a common data set for run-time or post-incident analysis. The Flight Recording data is also included in WLDF diagnostic image captures, enabling you to capture flight recording snapshots based on WLDF watch rules. This full set of functionality enables you to capture and analyze run-time system information for both the JVM and the Fusion Middleware components running on it, in a single view.

This chapter also explains common usage scenarios that show how this integration can provide for a comprehensive performance analysis and diagnostic foundation for production systems based on WebLogic Server.

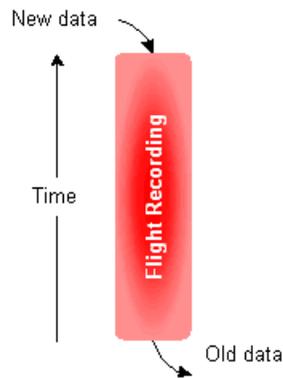
This chapter includes the following sections:

- [About Oracle JRockit Flight Recorder](#)
- [Key Features of WLDF Integration with JRockit Flight Recorder](#)
- [JRockit Flight Recorder Use Cases](#)
- [Obtaining the JRockit Flight Recording File](#)
- [Analyzing Flight Recorder Data in JRockit Mission Control](#)

3.1 About Oracle JRockit Flight Recorder

The version of Oracle JRockit that is available in the WebLogic Server installation program includes a component called JRockit Flight Recorder (JFR), a performance monitoring and profiling tool. JFR records diagnostic information on a continuous basis, making it always available, even in the wake of catastrophic failure such as a system crash.

JFR maintains a buffer of diagnostics and profiling data, called a flight recording or a JFR file, that you can access whenever you need it. The flight recording functions in a manner similar to an aircraft "black box" in which new data is continuously added and older data is stripped out, as shown in [Figure 3-1](#).

Figure 3–1 Circular Flight Recording Buffer

For details about how the JFR flight recording works, see "Flight Recorder Data Flow" in *Oracle JRockit Flight Recorder Run Time Guide*.

The JFR file can be analyzed at any time to examine the details of system execution flow that occurred leading up to an event. The data contained in the JFR file includes events from the JVM and from any other event producer, such as WebLogic Server and Oracle Dynamic Monitoring System (DMS).

The amount of additional processing overhead that results when you enable JFR recording, and also configure WLDF to generate WebLogic Server diagnostics to be captured by JFR, is minimal. This makes it ideal to be used on a full time basis, especially in production environments where it adds the greatest value.

JFR provides the following key benefits:

- Designed to run continuously — When JFR is configured to run full-time, with both JVM and WLDF events captured in the flight recording, diagnostic data is always available at the time an event occurs, including a system crash. This ensures that a record of diagnostic data leading up to the event is available, allowing you to diagnose the event without having to recreate it.
- Comprehensive data — JFR combines data generated by the JRockit Runtime Analyzer and the JRockit Latency Analysis Tool and presents it in one place.
- Integration with event providers — JRockit includes a set of APIs that allow the JFR to monitor additional system components, including WebLogic Server, Oracle Dynamic Monitoring System (DMS), and other Oracle products.

For more information about JFR, see "Introduction to the Flight Recorder" in *Oracle JRockit Flight Recorder Run Time Guide*.

3.2 Key Features of WLDF Integration with JRockit Flight Recorder

The key features provided by WLDF to leverage integration with JFR include the following:

- WLDF diagnostic data captured in JFR flight recording
 - WLDF can be configured to generate diagnostic data about WebLogic Server events that is captured in the JFR flight recording. Captured events include those from components such as: web applications; EJBs; JDBC, JTA, and JMS resources; resource adapters; and WebLogic Web Services.
- WLDF diagnostic volume control

The ability to generate WebLogic Server event data for the Flight Recording is controlled by the WLDF diagnostic volume configuration. This control also determines the amount of WebLogic Server event data that is captured by JFR, and can be adjusted to include more, or less, data for each WebLogic Server event that is generated. For more information, see [Section 5.2.1, "Configuring WLDF Diagnostic Volume."](#)

Notes:

- By default, the WLDF diagnostic volume is set to `Low`.
 - The WLDF diagnostic volume setting does not affect explicitly configured diagnostic modules.
-
-

- Automatic throttling of generated events under load

As processing load rises on a given WebLogic Server instance, WLDF automatically begins throttling the number of incoming WebLogic Server requests that are selected for event generation and recording into the JFR file. The degree of throttling is adjusted continuously as system load rises and falls.

Throttling provides three key benefits:

- The overhead of capturing events generated by WLDF for JFR remains minimized, which is especially important when systems are under load.
- The time interval encompassed in the JFR flight recording buffer is maximized, giving you a better historical record of data.
- Throttling has the effect of sampling incoming WebLogic Server requests, maintaining high performance while still providing an accurate overall view of system activity under load.

Note: Throttling affects only the Flight Recording data that is captured by WLDF. It does not affect data captured by other event producers, such as the JVM.

- WLDF diagnostic image capture support for JFR files

WLDF diagnostic image capture automatically includes the JFR file, if one has been generated by Flight Recorder. The JFR file includes data generated by all active event producers, including WebLogic Server. An image captured using the Watch and Notification component may contain the JFR file, if available.

- WLST commands for downloading the contents of diagnostic image captures

WLST includes a set of commands for downloading the contents of diagnostic image captures, described in [Section 5.4.2, "WLST Online Commands for Downloading Diagnostics Image Captures."](#) Although these commands are generally useful for listing, copying, and downloading all entries contained in the diagnostic image capture, they can also be used for obtaining the JFR file, if available. Once obtained from the diagnostic image capture, the JFR file can be viewed in JRockit Mission Control.

3.3 JRockit Flight Recorder Use Cases

This section summarizes three common business cases where using the JRockit Flight Recorder can help you resolve important diagnostic issues:

- [Diagnosing a Critical Failure — The "Black Box"](#)
- [Profiling During Performance Testing or in Production](#)
- [Real-time Application Diagnostics and Reporting \(RADAR\)](#)

For more information about these scenarios, see "Flight Recorder Use Cases" in *Oracle JRockit Flight Recorder Run Time Guide*.

3.3.1 Diagnosing a Critical Failure — The "Black Box"

When a "catastrophic" failure occurs, the content of the Flight Recorder buffer can be made available for post-failure analysis in a manner analogous to the use of an aircraft's black box. Examples of such failures include a JVM crash or an out-of-memory error (OOM) resulting in an application terminating.

When these situations arise, the flight recording contains the following information, which can be helpful in determining the cause of the failure:

- JVM core dump, including metadata about the Flight Recorder configuration at the time of the crash. Furthermore, if the Flight Recorder was running in persistent storage mode the data buffer file might contain a certain amount of data.
- WebLogic Server events, captured by WLDF, that preceded the failure.

When running in persistent mode, the JFR uses a combination of memory and disk to store its buffer. The most recent data is stored in memory and is flushed out to disk as it "ages". In this mode, the on-disk data will be available even after a power failure or similar catastrophic event; only the most recent data will be unavailable (for example, the data that had not yet been flushed to disk). The text dump file will contain metadata about the Flight Recorder configuration at the time of the crash, including the path to the data buffer file when applicable. For more information about persistent mode, see "Operating Modes" in *Oracle JRockit Flight Recorder Run Time Guide*.

3.3.2 Profiling During Performance Testing or in Production

Profiling involves capturing data beginning at a specific point in time so that, later, you can analyze the events that were generated after that point. In contrast to RADAR, described in the following section, profiling involves analyzing the diagnostic data generated *after* a particular event occurs, as opposed to the data that precedes it.

Profiling with JRockit Flight Recorder optimizes the ability to perform deep analysis of lock contention and causes of latency.

3.3.3 Real-time Application Diagnostics and Reporting (RADAR)

RADAR is the examination of diagnostic data generated during run time when a particular event occurs for the purposes of understanding the system activity that preceded the event; for example, system activity occurring moments before a serious error message is generated. By using the diagnostic capabilities available in WLDF in conjunction with JRockit Flight Recorder, you can capture a large amount of system-wide diagnostic data the moment a problem occurs. You can then leverage the capabilities of JRockit Mission Control to quickly correlate that event with other system activity and process execution data within the "snapshot in time" that the JFR file provides, enabling you to quickly isolate likely causes of the problem.

One WLDF feature whose usage with JRockit Flight Recorder makes for a powerful RADAR capability is image notification, which allows you to create a diagnostic image capture automatically in response to a particular event or error condition. A diagnostic image capture, which is created as the result of an image notification, automatically includes the JFR file. The JFR file can then be extracted from the diagnostic image capture and examined immediately in JRockit Mission Control or stored for later analysis. Image notification, used when WLDF data is captured by JRockit Flight Recorder, is particularly well suited for this sort of real-time diagnosis of intermittent problems.

Image notification is part of the Watch and Notifications system in WLDF. To set up image notification, you create one or more individual watch rules. A watch rule includes a logical expression that uses the WLDF query language to specify the event for the watch to detect. For example, the following log event watch rule expression detects the server log message with severity level `Critical` and ID `BEA-149618`:

```
(SEVERITY = 'Critical') AND (MSGID = 'BEA-149618')
```

Watch rules can monitor any of the following:

- Harvestable run-time MBean instances in the local run-time MBean server
A harvester watch can trigger an image notification if run-time MBean attributes detect a performance issue, such as high memory utilization rates or problems with open socket connections to the server.
- Messages published to the server log
A log watch can trigger an image notification if a specific message, severity level, or string is issued.
- Event generated by the WLDF Instrumentation component
An event watch can trigger an image notification if an instrumentation service generates a particular event.

For more information, see the following topics:

- [Chapter 8, "Configuring Watches and Notifications"](#)
- [Section 10.6, "Configuring Image Notifications"](#)
- [Appendix A, "WLDF Query Language"](#)

The following sections explain how to obtain the JFR file from the diagnostic image capture and provide an example of using JRockit Mission Control to examine the WebLogic Server events contained in the JFR file:

- [Obtaining the JRockit Flight Recording File](#)
- [Analyzing Flight Recorder Data in JRockit Mission Control](#)

3.4 Obtaining the JRockit Flight Recording File

The diagnostic image capture itself is a single JFR file that contains individual images produced by the different server subsystems. If the JFR file is available, it is included in the diagnostic image as the file `FlightRecording.jfr`.

A diagnostic image capture can be generated on-demand — for example, from the WebLogic Server Administration Console, WLST, or a JMX application — or it can be generated as the result of an image notification. For information about how to generate a diagnostic image captures and configure the location in which they are created, see

"Configure and capture diagnostic images" in *Oracle WebLogic Server Administration Console Online Help*.

To view the contents of the JFR file, you first need to extract it from the diagnostic image capture as described in [Chapter 5, "Configuring and Capturing Diagnostic Images."](#) Once you have extracted the JFR file, you can view its contents in JRockit Mission Control.

For an example WLST script that retrieves the JFR file from a diagnostic image file and saves it to a local directory, see [Section D.7, "Example: Retrieving a JFR File from a Diagnostic Image Capture."](#)

3.5 Analyzing Flight Recorder Data in JRockit Mission Control

You use JRockit Mission Control to examine the contents of the Flight Recorder file after it has been extracted from the diagnostic image capture. The following sections highlight some of the capabilities of JRockit Mission Control's graphical user interface, which provides a lot of tooling support for drilling down into the diagnostic data generated not only by WLDF for WebLogic Server events, but also from all other available event producers, including the JRockit JVM:

- [JFR Graphical User Interface](#)
- [Analyzing Execution Flow — A Sample Walkthrough](#)
- [Changing the Location of Temporary JFR Files](#)

For complete details about the JRockit Mission Control interface, see the Oracle JRockit Mission Control Online Help. See also *Introduction to JRockit Mission Control Client*.

3.5.1 JFR Graphical User Interface

JRockit Mission Control includes the JRockit Flight Recorder graphical user interface, which allows users who are running a Flight Recorder-compliant version of Oracle JRockit to view the JVM's recordings, current recording settings, and run-time parameters. The JFR interface includes the Events Type View, which gives you direct access to event information that has been recorded in the JFR file, such as event producers and types, event logging and graphing, event by thread, event stack traces, and event histograms.

The Overview tab in the JFR interface is useful for analyzing a system's general health because it can reveal behavior that might indicate bottlenecks or other sources of poor system performance. [Figure 3-2](#) shows an example of the Overview tab in the Events Type View.

Note the following regarding the information shown in [Figure 3-2](#):

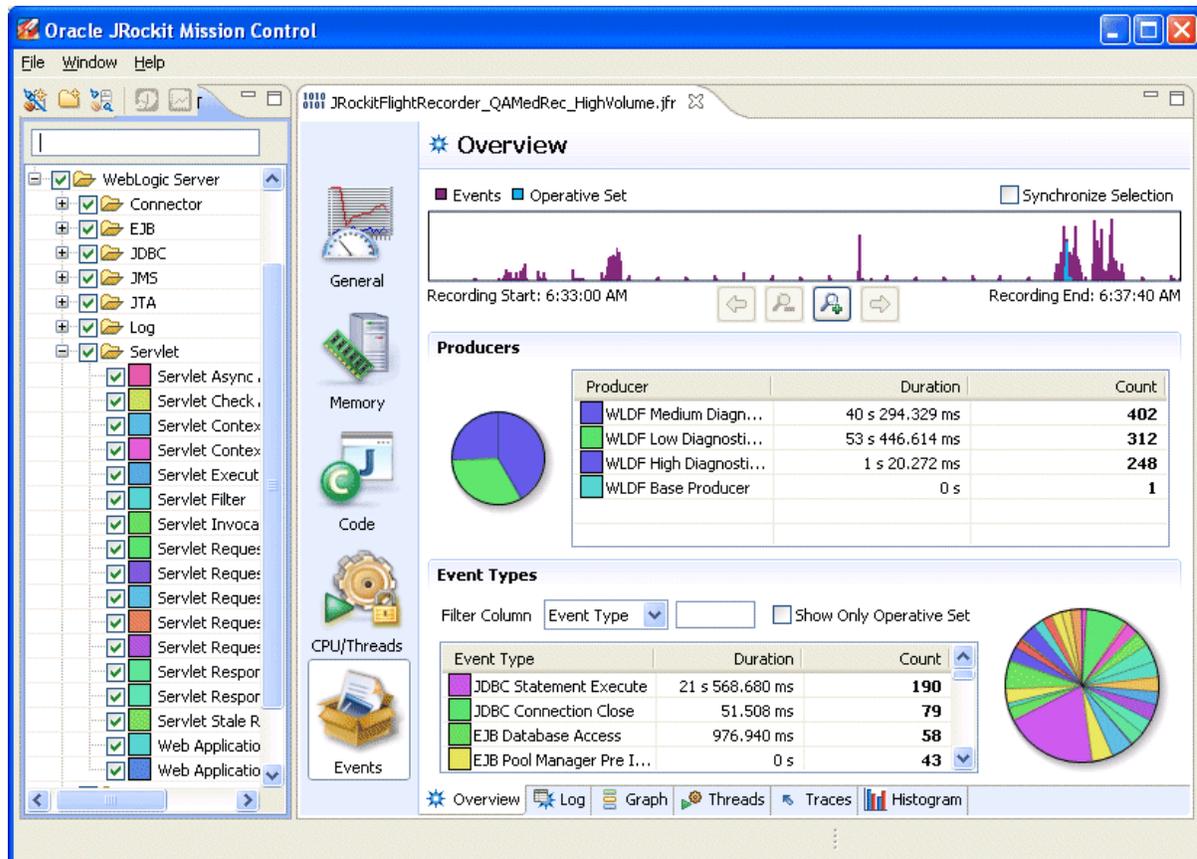
- The Events Type View is available by selecting the Events tab group icon.
- The name of the Flight Recorder file appears at the top of the Overview tab. Note that the JFR is always named `FlightRecording.jfr`, it is useful to rename it descriptively after downloading it from the diagnostic image capture.
- The Event Types Browser, on the left side, is a tree that shows the available event types in a recording. It works in conjunction with the Events tab group to provide a means to select events or groups of events in a recording that might be of interest to you and to obtain more granular information about them.

As you select and deselect entries in the Event Types Browser, the information displayed in the Overview tab is filtered dynamically. For example, by selecting

only WebLogic Server, event data from all non-WebLogic event producers is filtered out.

- The range navigator, which is the graph displayed below the Overview tab title, is a timeline that shows all events in a recording that pertain to the data displayed on the selected tab. A set of buttons are available for adjusting the range of data that is displayed, which can simplify the process of drilling down into the details of Flight Recorder data.
- The Producers section identifies each event producer that generated the data that is displayed. Metrics are included for each producer, indicating the volume of event activity generated by each as a proportion of the total set of event data displayed.
- The Event Types section lists all events represented in the Overview tab, along with key metric data about each event.

Figure 3–2 Example Overview Page of JRockit Flight Recorder File in JRockit Mission Control



3.5.2 Analyzing Execution Flow — A Sample Walkthrough

This section shows an example of the steps that a developer or support engineer might use to identify the event activity associated with a particular request in a Web application hosted on WebLogic Server. This example is not meant to recommend a specific way to diagnose performance problems, but simply shows how the JFR graphical user interface can be used to greatly simplify the process of locating and analyzing performance issues.

The following examples are shown in this section:

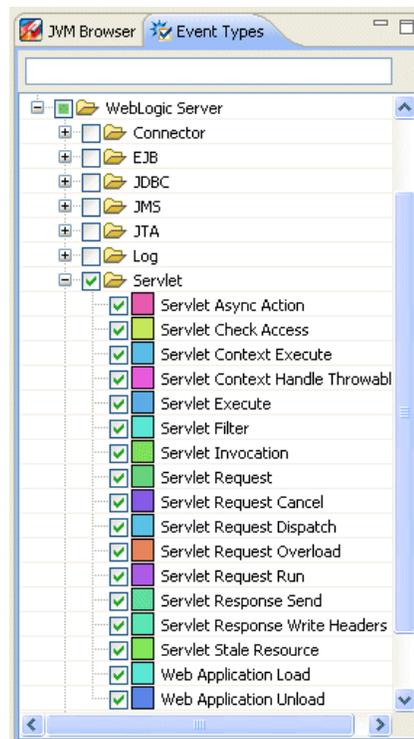
- [Displaying Event Data for a Product Subcomponent](#)
- [Viewing the Event Log to Display Details](#)
- [Tracking Execution Flow by Analyzing an Operative Set](#)
- [Expanding the Operative Set and Viewing Correlated Diagnostic Data](#)

3.5.2.1 Displaying Event Data for a Product Subcomponent

When you start JRockit Mission Control and open a JFR file, you can use the Event Types View to quickly select the specific events you want to analyze. As you select and deselect items in the Event Types Browser (which is available in the Event Types View), the information displayed in the JFR graphical user interface is updated instantly to show information about only the selected event types.

[Figure 3–3](#) shows the Event Types Browser with only servlet event types selected.

Figure 3–3 *Event Types Browser*



3.5.2.2 Viewing the Event Log to Display Details

To view details about the events logged by one or more event types, select the Log tab, which is available at the bottom of the JFR graphical user interface. An example of the Log tab for servlet event types is shown in [Figure 3–4](#).

Figure 3–4 Servlet Event Log

Event Log

Filter Column: Show Only Operative Set

Start Time	Duration	Thread	Event Type
3/5/10 6:36:59 AM 278ms	2 s 140.484 ms	[ACTIVE] ExecuteThrea...	Servlet Invocation
3/5/10 6:37:08 AM 982ms	1 s 710.013 ms	[ACTIVE] ExecuteThrea...	Servlet Request Run
3/5/10 6:37:08 AM 987ms	1 s 703.720 ms	[ACTIVE] ExecuteThrea...	Servlet Invocation
3/5/10 6:37:10 AM 694ms	1 s 674.000 ms	[ACTIVE] ExecuteThrea...	Servlet Request Run
3/5/10 6:37:10 AM 698ms	1 s 667.291 ms	[ACTIVE] ExecuteThrea...	Servlet Invocation
3/5/10 6:37:14 AM 159ms	1 s 420.820 ms	[ACTIVE] ExecuteThrea...	Servlet Request Run
3/5/10 6:37:14 AM 163ms	1 s 416.196 ms	[ACTIVE] ExecuteThrea...	Servlet Invocation
3/5/10 6:36:52 AM 747ms	1 s 354.143 ms	[ACTIVE] ExecuteThrea...	Servlet Request Run
3/5/10 6:36:57 AM 994ms	1 s 279.437 ms	[ACTIVE] ExecuteThrea...	Servlet Request Run
3/5/10 6:36:57 AM 997ms	1 s 274.969 ms	[ACTIVE] ExecuteThrea...	Servlet Invocation

Event Attributes

Name	Value
Start Time	3/5/10 6:36:59 AM 278ms
End Time	3/5/10 6:37:01 AM 418ms
Duration	2 s 140.484 ms
User ID	admin@avitek.com
Return Value	
Method Name	doIt
Class Name	weblogic.servlet.internal.WebAppServletContext\$ServletInvocationAction
Transaction ID	N/A
ECID	b4325c3ce72548e4:34e9fa75:1272e19ce0a:-8000-00000000000013af
URI	/admin/approve.do

Overview Log Graph Threads Traces Histogram

When using the Log tab, you can view details about events as follows:

- You can click on individual column heads in the Event Log table to modify the sort order of the events. For example, by clicking the Duration column, you can quickly identify the events that took the longest time to execute.
- When you select an event in the Event Log table, details about that event are displayed in the Event Attributes table. For example, Figure 3–4 shows the following attributes:
 - Event start, end, and duration times
 - User ID of person who issued the request on the servlet
 - Method, class name, and URI of invoked servlet
 - Execution context ID (ECID)

Different event types have different attributes. For example, if this were a JDBC event, you could scroll among the attributes to see the SQL statement, the JDBC connection pool used, and the stack from which it was called. The interface makes it easy to scan for unexpected behavior that can be analyzed in deeper detail.

Note: The value of the ECID is a unique identifier that can be used to correlate individual events as being part of the same request execution flow. For example, events that are identified as being related to a particular request typically have the same ECID value, as shown in Section 3.5.2.3, "Tracking Execution Flow by Analyzing an Operative Set." However, the format of the ECID string itself is determined by an internal mechanism that is subject to change; therefore, you should not have or place any dependencies on that format.

3.5.2.3 Tracking Execution Flow by Analyzing an Operative Set

The JFR graphical user interface in JRockit Mission Control allows you to analyze the run-time trail of system activity that occurs as the result of a particular event. In this example, the run-time trail is analyzed by first defining an *operative set*. An operative set is any set of events that you choose to work in JRockit Mission Control.

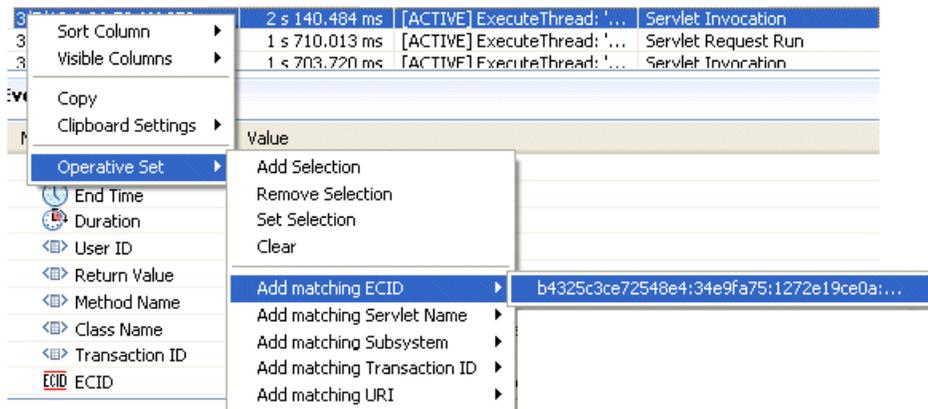
In the example shown in this section, an operative set is created for the events that have the same execution context ID (ECID) attribute as the servlet invocation event selected in the Event Log table, shown in [Figure 3-5](#). The operative set is then analyzed to see the execution flow that resulted from that servlet invocation. (Note that this operative set could be expanded to include events that match on different attributes as well; for example, events containing a specific SQL statement but not necessarily the same ECID.)

Figure 3-5 Operative Set Defined by Execution Context ID (ECID)

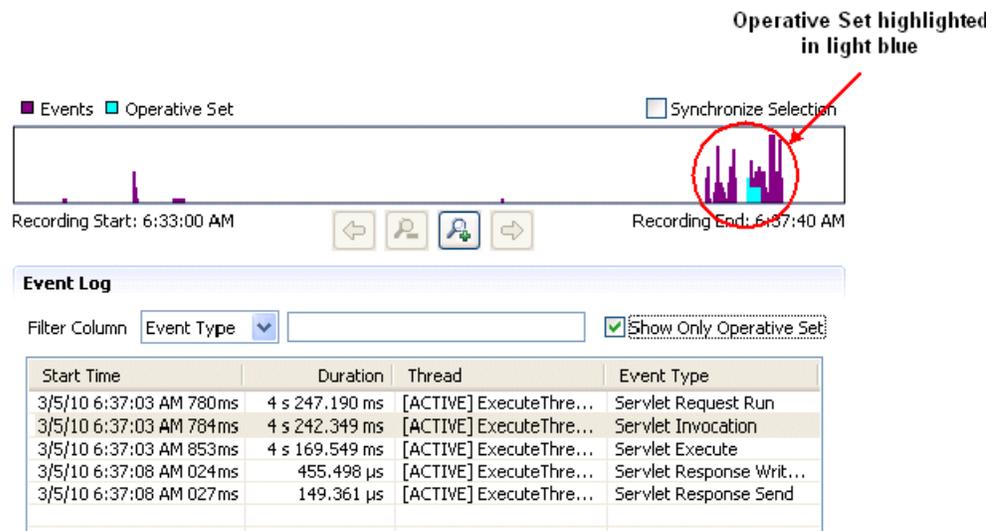
Start Time	Duration	Thread	Event Type
3/5/10 6:36:59 AM 274ms	2 s 145.295 ms	[ACTIVE] ExecuteThrea...	Servlet Request Run
3/5/10 6:36:59 AM 278ms	2 s 140.484 ms	[ACTIVE] ExecuteThrea...	Servlet Invocation
3/5/10 6:36:59 AM 306ms	157.483 μs	[ACTIVE] ExecuteThrea...	EJB Home Create
3/5/10 6:36:59 AM 308ms	0 s	[ACTIVE] ExecuteThrea...	EJB Pool Manager Pre Invoke
3/5/10 6:36:59 AM 334ms	103.893 μs	[ACTIVE] ExecuteThrea...	JDBC Connection Reserve
3/5/10 6:36:59 AM 338ms	0 s	[ACTIVE] ExecuteThrea...	JDBC Transaction Is Same RM
3/5/10 6:36:59 AM 345ms	0 s	[ACTIVE] ExecuteThrea...	JDBC Transaction Is Same RM
3/5/10 6:36:59 AM 349ms	0 s	[ACTIVE] ExecuteThrea...	JTA Transaction Start
3/5/10 6:36:59 AM 351ms	0 s	[ACTIVE] ExecuteThrea...	JDBC Transaction Start

This operative set is defined by right-clicking the desired event in the Event Log, and then selecting **Operative Set > Add matching ECID > ecid**. See [Figure 3-6](#).

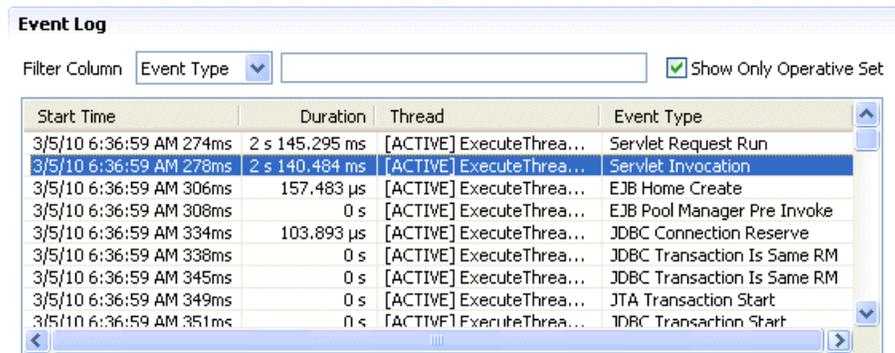
Figure 3-6 Defining an Operative Set by Matching ECID



The operative set is then displayed by selecting **Show Only Operative Set** above the event log table, shown in [Figure 3-7](#). Note how the operative set is indicated in the range navigator.

Figure 3–7 *Displaying an Operative Set*

The run-time trail of execution flow that results from the request that generated the servlet invocation event can be viewed by including additional event types. For example, [Figure 3–8](#) shows the operative set when all WebLogic Server event types are added, using the Event Type Browser, and listing the events in chronological order. (You can sort the events chronologically by selecting the Start Time column head.)

Figure 3–8 *Adding all WebLogic Server Events to Operative Set*

In this example, note a portion of the execution flow shown in the Event Log:

1. The servlet URI is invoked.
2. The servlet uses an EJB, which requires access to the database.
3. A JDBC connection is obtained and a transaction is started.

3.5.2.4 Expanding the Operative Set and Viewing Correlated Diagnostic Data

The operative set can be further analyzed by constraining the time interval of the execution flow and adding correlated events from additional producers. By constraining the time interval for displayed events, you can add events to the Event Log that occurred simultaneously with the operative set. This allows you to see additional details about the execution context that can help diagnose performance issues.

The time interval can be constrained by using the range selection bars in the range navigator. You can grab these bars with your pointer and drag them inward or outward to change the range of events displayed in the Event Log. The range selection bars are activated when you hover your pointer over either end of the navigator, as shown in [Figure 3-9](#).

Figure 3-9 Range Navigator Selection Bars



Events from additional producers, such as the JRockit JVM, can be selected in the Event Types Browser. Note that JVM events do not have ECID attributes, so they cannot be included among the WLDF events in the operative set. So to view the JVM events, you need to de-select **Show Only Operative Set**.

At this point the events that are displayed in the Event Log are those that occurred during the selected time interval but not correlated otherwise. [Figure 3-10](#) shows drilling down into JDBC activity by selecting only JDBC events and JVM socket events. The Event Log is updated and listed in chronological order to show the socket activity that occurred simultaneously to the flow of the JDBC events in the selected time interval.

Figure 3-10 Adding JVM Events to JDBC Event Log

Recording Start: 6:33:00 AM Recording End: 6:37:40 AM

Event Log

Filter Column: Event Type Show Only Operative Set

Start Time	Duration	Thread	Event Type
3/5/10 6:36:59 AM 360ms	120.003 µs	[ACTIVE] ExecuteThrea...	JDBC Statement Creation
3/5/10 6:36:59 AM 365ms	111.274 ms	[ACTIVE] ExecuteThrea...	JDBC Statement Execute
3/5/10 6:36:59 AM 365ms	110.712 ms	[ACTIVE] ExecuteThrea...	Socket Read
3/5/10 6:36:59 AM 478ms	86.123 µs	[ACTIVE] ExecuteThrea...	JDBC Connection Close
3/5/10 6:36:59 AM 897ms	0 s	[ACTIVE] ExecuteThrea...	JDBC Transaction Is Sa...
3/5/10 6:37:00 AM 009ms	37.342 ms	[ACTIVE] ExecuteThrea...	JDBC Connection Prepare
3/5/10 6:37:00 AM 045ms	119.853 µs	[ACTIVE] ExecuteThrea...	JDBC Statement Creation
3/5/10 6:37:00 AM 053ms	112.169 ms	[ACTIVE] ExecuteThrea...	JDBC Statement Execute
3/5/10 6:37:00 AM 054ms	111.639 ms	[ACTIVE] ExecuteThrea...	Socket Read
3/5/10 6:37:00 AM 168ms	89.600 µs	[ACTIVE] ExecuteThrea...	JDBC Connection Close
3/5/10 6:37:00 AM 169ms	0 s	[ACTIVE] ExecuteThrea...	JDBC Transaction End
3/5/10 6:37:00 AM 229ms	0 s	[ACTIVE] ExecuteThrea...	JDBC Transaction Prepare
3/5/10 6:37:00 AM 301ms	109.204 ms	Thread-14	JDBC Statement Execute

3.5.3 Changing the Location of Temporary JFR Files

The temporary JFR files created in the operating system's `temp` directory are managed directly by JRockit. WLDF does not control these files. (By default, WLDF temporary files related to JFR recordings are placed in the `DOMAIN_NAME\servers\SERVER_NAME/server/logs/diagnostic_images` directory.)

However, you can change the location in which JRockit places its temporary files by using the following command-line option when starting JFR, where `path` represents the preferred location:

```
-XX:FlightRecorderOptions=repository=path
```

For more information about JFR configuration settings, see "Starting the Flight Recorder" in *Oracle JRockit Flight Recorder Run Time Guide*.

Understanding WLDF Configuration

This chapter describes the features provided by the WebLogic Diagnostics Framework (WLDF) for generating, gathering, analyzing, and persisting diagnostic data from WebLogic Server instances and from applications deployed to them. For server-scoped diagnostics, some WLDF features are configured as part of the configuration for a server in a 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.

This chapter includes the following sections:

- [Configuration MBeans and XML](#)
- [Tools for Configuring WLDF](#)
- [How WLDF Configuration Is Partitioned](#)
- [Configuring Diagnostic Image Capture and Diagnostic Archives](#)
- [Configuring Diagnostic Image Capture for JRockit Flight Recorder](#)
- [Configuring Diagnostic System Modules](#)
- [Configuring Diagnostic Modules for Applications](#)
- [WLDF Configuration MBeans and Their Mappings to XML Elements](#)

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

4.1 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 `config.xml`. When you modify a configuration by changing the values of MBean attributes, those changes are saved (persisted) in the XML files.

Configuration MBean attributes map directly to configuration XML elements. For example, the `Enable` attribute of the `WLDFInstrumentationBean` maps directly to the `<enabled>` sub-element of the `<instrumentation>` element 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 were to edit an XML element in the configuration file directly (which is not recommended), the change to an MBean value would take effect after the next session is started.

For more information about WLDF Configuration MBeans, see [Section 4.8, "WLDF Configuration MBeans and Their Mappings to XML Elements."](#) 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 for Oracle WebLogic Server*.

4.2 Tools for Configuring WLDF

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

- Use the Administration Console to configure WLDF for server instances and clusters. See "Configure the WebLogic Diagnostics 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 D, "WebLogic Scripting Tool Examples."](#) Also see *Oracle WebLogic Scripting Tool* for general information about using WLST.
- Configure WLDF programmatically using JMX and the WLDF configuration MBeans. See [Chapter 16, "Configuring and Using WLDF Programmatically,"](#) for specific information about programming WLDF. See *Oracle WebLogic Server MBean Reference* and browse or search for specific MBeans for programming reference.
- Edit the XML configuration files directly. This documentation explains many configuration tasks by showing and explaining the XML elements in the configuration files. The XML is easy to understand, and you can edit the configuration files directly, although it is recommended that you do not. (If you have a good reason to edit the files directly, you should first generate the XML files by configuring WLDF in the Administration Console. Doing so provides a blueprint for valid XML.)

Note: If you make changes to a configuration by editing configuration files, you must restart the server for the changes to take effect.

4.3 How WLDF Configuration Is Partitioned

You can use WLDF to perform diagnostics tasks for server instances (and clusters) and for applications.

4.3.1 Server-Level Configuration

You configure the following WLDF components as part of a server instance in a domain. The configuration settings are controlled using MBeans and are persisted in the domain's config.xml file.

- Diagnostic Image Capture
- Diagnostic Archives

See [Section 4.4, "Configuring Diagnostic Image Capture and Diagnostic Archives."](#)

You configure the following WLDF components as the parts of one or more diagnostic system modules, or resources, that can be deployed to one or more server instances (or clusters). These configuration settings are controlled using Beans and are persisted in

one or more diagnostic resource descriptor files (configuration files) that can be targeted to one or more server instances or clusters.

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

See [Section 4.6, "Configuring Diagnostic System Modules."](#)

4.3.2 Application-Level Configuration

You can use the WLDF Instrumentation component with applications, as well as at the server level. The Instrumentation component is configured in a resource descriptor file deployed with the application in the application's archive file. See [Section 4.7, "Configuring Diagnostic Modules for Applications."](#)

4.4 Configuring Diagnostic Image Capture and Diagnostic Archives

In the config.xml file for a domain, you configure the Diagnostic Image Capture component and the Diagnostic Archive component in the <server-diagnostic-config> element, which is a child of the <server> element in a domain, as shown in [Example 4-1](#).

Example 4-1 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 Example 4-2. -->
</domain>
```

Note: If WebLogic Server is configured with Oracle JRockit, and JRockit Flight Recorder is enabled, the diagnostic image capture can optionally include a JRockit Flight Recorder (JFR) file that includes WebLogic Server events. The JFR file can then be viewed in JRockit Mission Control.

For more information, see the following:

- [Chapter 5, "Configuring and Capturing Diagnostic Images"](#)
- [Chapter 6, "Configuring Diagnostic Archives"](#)

4.5 Configuring Diagnostic Image Capture for JRockit Flight Recorder

If WebLogic Server is configured with Oracle JRockit R28 or later, and JRockit Flight Recorder is not explicitly disabled, the JFR file is automatically included in the diagnostic image capture. The JFR file contains data from all event producers that are enabled. However, the amount of WebLogic Server event data that is included in the JFR file is determined by the configuration of the WLDF diagnostic volume.

Note: By default, the WLDF diagnostic volume is set to `Low`.

To include WebLogic Server event data in the JFR file:

1. Ensure that WebLogic Server is configured with Oracle JRockit R28, which is available from the WebLogic Server installation program.

For information, see *Installation Guide for Oracle WebLogic Server*.

2. Ensure that JFR flight recording is not disabled in JRockit.

In a default installation of Oracle JRockit, Flight Recorder operates in Default Data Gathering Mode. No JRockit settings are required to have JVM events included in the JFR file.

For information, see *Oracle JRockit Flight Recorder Run Time Guide*.

3. Set the WLDF diagnostic volume as appropriate. For general use, Oracle recommends the default setting of `Low`. However, you can increase the volume of WebLogic Server event data that is generated, as appropriate, by setting the volume to `Medium` or `High`.

Note that the WLDF diagnostic volume setting has no impact on data recorded for other event producers, such as the JVM.

For information, see "Configure WLDF diagnostic volume" in *Oracle WebLogic Server Administration Console Online Help*.

Note: If the WLDF diagnostic volume is set to `Off`, and Flight Recorder has not been explicitly disabled, the JFR file continues to include JVM event data and is always included in the diagnostic image capture.

4.6 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*, which will contain the configurations for all those components. Keep in mind that:

- System modules are globally available for targeting to servers and clusters configured in a domain.
- In a given domain, you can create multiple diagnostic system modules with distinct configurations.
- At most, one diagnostic system module can be targeted to any given server or cluster.

4.6.1 The Diagnostic System Module and Its Resource Descriptor

You create a diagnostic system module through the Administration Console or the WebLogic Scripting Tool (WLST). It 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://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-diagnostics.xsd>.

For instructions on creating a diagnostic system module, see "Create diagnostic system modules" in the *Oracle WebLogic Server Administration Console Online Help*.

4.6.2 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 *DOMAIN_NAME/config/diagnostics*, and a reference to the module is added to the domain's `config.xml` file.

Note: Oracle recommends that you do not write XML configuration files directly. But if you have a valid reason to do so, you should first create a diagnostic module from the Console. That way, you can start with the valid XML that the Console creates. For instructions, see "Create diagnostic system modules" in the *Oracle WebLogic Server 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.

For example, [Example 4-2](#) shows a `config.xml` file with a module named `myDiagnosticModule` targeted to the server `myserver` and another module named `newDiagnosticMod` targeted to servers `ManagedServer1` and `ManagedServer2`.

Example 4-2 Sample WLDF Configuration Information in the Config.xml File for a Domain

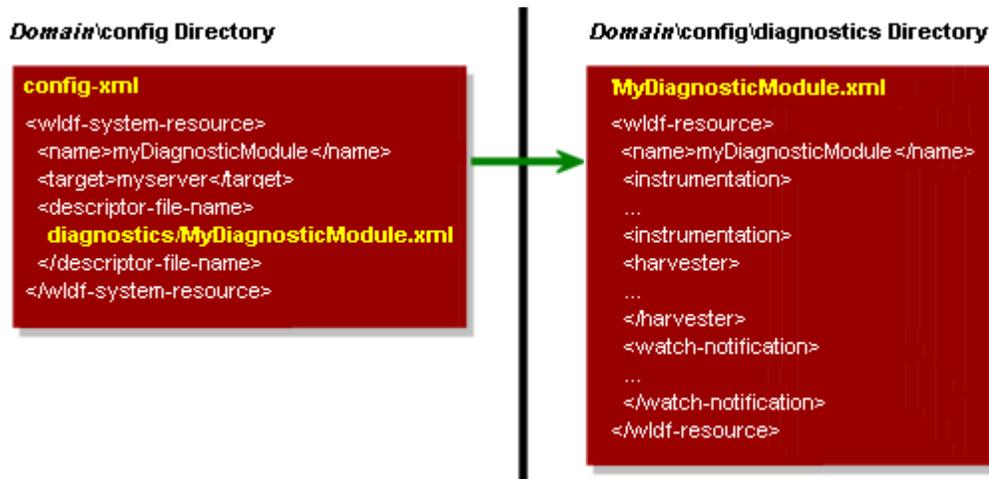
```
<domain>
  <!-- Other domain-level configuration elements -->
  <wldf-system-resource
    xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics">
    <name>myDiagnosticModule</name>
    <target>myserver</target>
    <descriptor-file-name>diagnostics/MyDiagnosticModule.xml
    </descriptor-file-name>
    <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>
  <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 4-1.

Figure 4-1 Relationship of config.xml to System Descriptor File



4.6.3 The DIAG_MODULE.xml 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 system module is saved in its resource descriptor file. Example 4-3 shows portions of the descriptor file for a diagnostic system module named myDiagnosticModule.

Example 4-3 Sample Structure of a Diagnostic System Module Descriptor File, MyDiagnosticModule.xml

```

<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-diagnostics.xsd">
  <name>MyDiagnosticModule</name>
  <instrumentation>
    <!-- Configuration elements for zero or more diagnostic monitors -->
  </instrumentation>
  <harvester>
    <!-- Configuration elements for harvesting metrics from zero 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>
```

4.6.4 Managing Diagnostic System Modules

A diagnostic system module can be targeted to zero, one, or more servers or clusters, although a given server can have only one module targeted to it at a time. You can create multiple modules that monitor different aspects of your system. You can then 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. This gives you considerable flexibility in writing and using diagnostic monitors that address a specific diagnostic goal, without interfering with the operation of the server instances themselves.

4.6.5 More Information About Configuring Diagnostic System Resources

See the following sections for detailed instructions on configuring WLDf system resources:

- [Chapter 7, "Configuring the Harvester for Metric Collection"](#)
- [Chapter 8, "Configuring Watches and Notifications"](#)
- [Chapter 11, "Configuring Instrumentation"](#)
- [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts"](#)

4.7 Configuring Diagnostic Modules for Applications

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

You configure and deploy application-scoped instrumentation as a diagnostic module, which is similar to a diagnostic system module. However, an application module is configured in an XML descriptor (configuration) file named `weblogic-diagnostics.xml`, which is packaged with the application archive in the `ARCHIVE_PATH/META-INF` directory for the deployed application. For example, `D:\bea\wlserver_12.1\samples\server\medrec\dist\standalone\exploded\medrec\META-INF\weblogic-diagnostics.xml`

Note: The DyeInjection monitor, which is used to configure diagnostic context (a way of tracking requests as they flow through the system), can be configured only at the server level. But once a diagnostic context is created, the context attached to incoming requests remains with the requests as they flow through the application. For information about the diagnostic context, see [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts."](#)

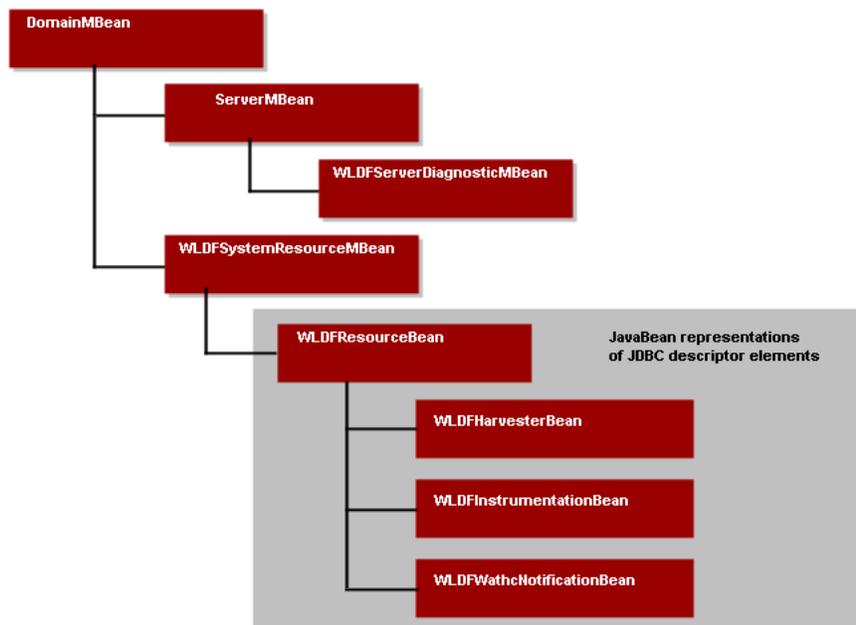
For more information about configuring and deploying diagnostic modules for applications, see:

- [Section 11.5, "Configuring Application-Scoped Instrumentation"](#)
- [Chapter 14, "Deploying WLDF Application Modules"](#)

4.8 WLDF Configuration MBeans and Their Mappings to XML Elements

Figure 4–2 shows the hierarchy of the WLDF configuration MBeans and the diagnostic system module beans for WLDF objects in a WebLogic Server domain.

Figure 4–2 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. (Diagnostic context is a way to uniquely identify requests and track them as they flow through the system. See [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts."](#))

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

- WLDFSystemResourceMBean contains the name of a descriptor file for a diagnostic module in the *DOMAIN_NAME/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 system 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 system module. This bean is represented by a <wldf-resource> element in a *DIAG_MODULE.xml* diagnostics descriptor file in the domain's config/diagnostics directory. (See [Figure 4-1](#) and [Example 4-3](#).) The WLDFResourceBean contains configuration settings for the following components:
 - Harvester: The WLDHHarvesterBean is represented by the <harvester> element in a *DIAG_MODULE.xml* file.
 - Instrumentation: The WLDHInstrumentationBean is represented by the <instrumentation> element in a *DIAG_MODULE.xml* file.
 - Watch and Notification: The WLDHWatchNotificationBean is represented by the <watch-notification> element in a *DIAG_MODULE.xml* file.

If a WLDFResourceBean is linked from a WLDHSystemResourceMBean, the settings for WLDF components apply to the targeted 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 WLDFResourceMBean is not a child of a WLDHSystemResourceMBean.

Configuring and Capturing Diagnostic Images

This chapter describes the Diagnostic Image Capture component of the WebLogic Diagnostics Framework (WLDF) that you can use to create a diagnostic snapshot, or dump, of a server's internal run-time state at the time of the capture. This information helps support personnel analyze the cause of a server failure.

If WebLogic Server is configured with Oracle JRockit, and JRockit Flight Recorder is enabled, the diagnostic image capture includes WebLogic Server diagnostic data that can be viewed in JRockit Mission Control.

This chapter includes the following sections:

- [How to Initiate Image Captures](#)
- [Configuring Diagnostic Image Captures](#)
- [How Diagnostic Image Capture Is Persisted in the Server's Configuration](#)
- [Content of the Captured Image File](#)

5.1 How to Initiate Image Captures

A diagnostic image capture can be initiated by:

- A configured watch notification. See [Chapter 10, "Configuring Notifications."](#)
- A request initiated by a user in the Administration Console (and requests initiated from third-party diagnostic tools). See "Configure and capture diagnostic images" in the *Oracle WebLogic Server Administration Console Online Help*.
- A direct API call, using JMX. See [Example 5-1](#).
- WLST command

5.2 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 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

- WLDF diagnostics volume, which determines the volume of WebLogic Server event information that is captured in the Flight Recorder file.

As with other WLDF components, you can configure Diagnostic Image Capture using the Administration Console (see "Configure and capture diagnostic images" in the *Oracle WebLogic Server Administration Console Online Help*), the WebLogic Scripting Tool (WLST), or programmatically.

Note: It is often useful to generate a diagnostic image capture when a server fails. To do so, set a watch rule to evaluate to true when the server's state changes to FAILED; then associate an image notification with the watch.

The watch rule is as follows:

```
([${weblogic.management.runtime.ServerRuntimeMBean} /  
/State] = 'FAILED')
```

For more information, see [Section 9.3, "Configuring Harvester Watches,"](#) and [Section 10.6, "Configuring Image Notifications."](#) Also see "Configure Watches and Notifications" in the *Oracle WebLogic Server Administration Console Online Help*.

5.2.1 Configuring WLDF Diagnostic Volume

If WebLogic Server is configured with Oracle JRockit, and the JRockit Flight Recorder is enabled, JRockit Flight Recorder data is automatically also captured in the diagnostic image capture. This data can be extracted from the diagnostic image capture and viewed in JRockit Mission Control. If JRockit Flight Recorder is not enabled, or if WebLogic Server is configured with a different JVM, the Flight Recorder data is not captured in the diagnostics image capture.

The volume of Flight Recorder data that is captured can be configured from the WebLogic Server Administration Console, which allows you to specify the following settings:

- `Off` — No data is captured in the Flight Recorder diagnostic image.
- `Low` — (Default). Basic information is captured when messages with the "emergency", "alert", or "critical" levels are recorded.

Note: The default setting for the WLDF diagnostic volume is `Low`.

- `Medium` — Additional information is captured when messages with the "error" level and above are recorded.
- `High` — In-depth information is captured when messages with the "error" level and above are recorded.

For information about how to set the volume of data that is captured, see "Configure WLDF diagnostics volume" in the *Oracle WebLogic Server Administration Console Online Help*.

5.2.2 WLST Commands for Generating an Image Capture

[Example 5-1](#) shows an example of WLST commands for generating an image capture.

Example 5-1 Sample WLST Commands for Generating a Diagnostic Image

```
url='t3://localhost:7001'
username='system'
password='gumby1234'
server='myserver'
timeout=120
connect(username, password, url)
serverRuntime()
cd('WLDFFRuntime/WLDFFRuntime/WLDFImageRuntime/Image')
argTypes = jarray.array(['java.lang.Integer'], java.lang.String)
argValues = jarray.array([timeout], java.lang.Object)
invoke('captureImage', argValues, argTypes)
```

5.3 How Diagnostic Image Capture Is Persisted in the Server's Configuration

The configuration for Diagnostic Image Capture is persisted in the config.xml file for a domain, under the <server-diagnostic-config> sub-element of the <server> element for the server, as shown in [Example 5-2](#):

Example 5-2 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>
```

Note: Oracle recommends that you do not edit the config.xml file directly.

5.4 Content of the Captured Image File

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
- Most recent harvested data

The Diagnostic Image Capture component captures and combines the images produced by the different server subsystems into a single ZIP file. In addition to

capturing the most common sources of the server state, this component captures images from all the server subsystems including, for example, images produced by the JMS, JDBC, EJB, and JNDI subsystems.

If WebLogic Server is configured with Oracle JRockit, and JRockit Flight Recorder is enabled, the diagnostic image capture includes a JRockit Flight Recorder image, `FlightRecording.jfr`, that can be viewed in JRockit Mission Control. The contents of the JRockit Flight Recorder image contains all available data from the Flight Recorder, and the volume of data produced by WLDF depends on the diagnostics volume setting. When JRockit Flight Recorder is enabled, data is always provided by Oracle JRockit, and optionally includes data provided by WebLogic Server. Data from additional Oracle components, such as Oracle Dynamic Monitoring System (DMS), may be included in the Flight Recorder image as well.

Notes:

1. A diagnostic image is a heavyweight artifact meant to serve as a server-level state dump for the purpose of diagnosing significant failures. It enables you to capture a significant amount of important data in a structured format and then to provide that data to support personnel for analysis.
 2. If a non-WebLogic event producer in the WebLogic Server environment, such as DMS, has configured JRockit Flight Recorder to record data, the WLDF diagnostic image capture includes a Flight Recorder image file with the recorded data even if the WLDF diagnostics volume is set to Off.
-
-

5.4.1 Data Included in the Diagnostics Image Capture File

Each image is captured as a single file for the entire server. The default location is `SERVER\logs\diagnostic_images`. Each image instance has a unique name, as follows:

```
diagnostic_image_DOMAIN_SERVER_YYYY_MM_DD_HH_MM_SS.zip
```

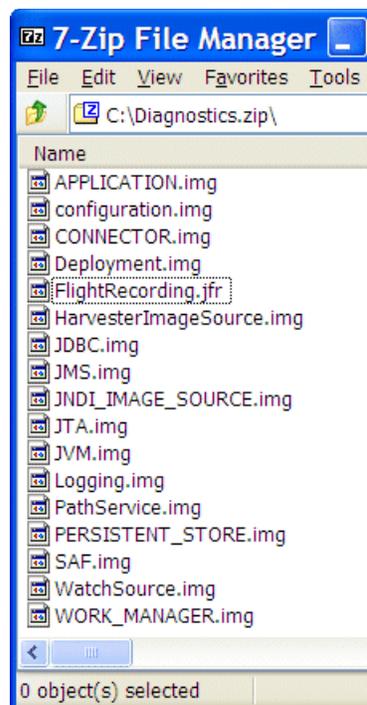
The contents of the file include at least the following information:

- 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

If WLDF is configured with Oracle JRockit as described in [Section 4.5, "Configuring Diagnostic Image Capture for JRockit Flight Recorder,"](#) the image also contains the JRockit Flight Recorder (JFR) file, `FlightRecording.jfr`. The JFR file can be extracted as described in [Section 5.4.2, "WLST Online Commands for Downloading Diagnostics Image Captures,"](#) and viewed in JRockit Mission Control. For more information, see *Oracle JRockit Flight Recorder Run Time Guide*.

[Figure 5–1](#) shows the contents of an image file. You can open most of the files in this ZIP file with a text editor to examine the contents.

Figure 5-1 An Image File



5.4.2 WLST Online Commands for Downloading Diagnostics Image Captures

WLST online provides the following commands for downloading diagnostic image captures from the server to which WLST is connected:

- `getAvailableCapturedImages` — Returns a list of diagnostic images that have been created in the image destination directory configured on the server.
- `saveDiagnosticImageCaptureFile` — Downloads a specified diagnostic image capture file.
- `saveDiagnosticImageCaptureEntryFile` — Downloads a specific entry within a diagnostic image capture. This command is particularly useful for obtaining the Flight Recorder diagnostics data for viewing in JRockit Mission Control.

For information about these commands, and examples of using them, see *WebLogic Scripting Tool Command Reference*. For examples of WLST scripts that return a list of diagnostic images and retrieve JFR files in them, see [Appendix D, "WebLogic Scripting Tool Examples."](#)

Configuring Diagnostic Archives

This chapter describes the Archive component of the WebLogic Diagnostics Framework (WLDF) that captures and persists all data events, log records, and metrics collected by WLDF from server instances and applications running on them. You can subsequently access archived diagnostic data in online mode (that is, on a running server). You can also access archived data in off-line mode using the WebLogic Scripting Tool (WLST).

You can configure WLDF to archive diagnostic data to a file store or a Java Database Connectivity (JDBC) data source, as described in this chapter.

This chapter includes the following sections:

- [Configuring the Archive](#)
- [Configuring a File-Based Store](#)
- [Configuring a JDBC-Based Store](#)

You can also specify when and under what conditions old data will be removed from the archive, as described in [Section 6.4, "Retiring Data from the Archives."](#)

6.1 Configuring the Archive

You configure the diagnostic archive on a per-server basis. The configuration is persisted in the `config.xml` file for a domain, under the `<server-diagnostic-config>` element for the server. Example configurations for file-based stores and JDBC-based stores are shown in [Example 6-1](#) and [Example 6-3](#).

Note: Resetting the system clock while diagnostic data is being written to the archive can produce unexpected results. See [Section 13.5, "Resetting the System Clock Can Affect How Data Is Archived and Retrieved."](#)

6.2 Configuring a File-Based Store

For a file-based store, WLDF creates a file to contain 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 `DOMAIN_NAME/servers/SERVER_NAME/data/store/diagnostics`, where `DOMAIN_NAME` is the home directory for the domain, and `SERVER_NAME` is the home directory for the server instance.

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

An example configuration for a file-based store is shown in [Example 6-1](#).

Example 6-1 Sample Configuration for File-based Diagnostic Archive (in config.xml)

```
<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>
```

6.3 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. For information about how to configure JDBC using the Administration Console, see "Configure database connectivity" in *Oracle WebLogic Server Administration Console Online Help*. For additional information about JDBC configuration, see *Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*.

6.3.1 Creating WLDF Tables in the Database

If they do not 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. [Example 6-2](#) shows the DDL that you can use to create WLDF tables in Apache Derby.

Example 6-2 DDL Definition of the WLDF Tables for Apache Derby

```
-- WLDF Instrumentation and Harvester archive DDLs using Derby

AUTOCOMMIT OFF;

-- DDL for creating wls_events table for instrumentation events

DROP TABLE wls_events;

CREATE TABLE wls_events (
  RECORDID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
  TIMESTAMP BIGINT 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 BIGINT default NULL,
THREADNAME varchar(128) default NULL
);

-- DDL for creating wls_events table for instrumentation events

DROP TABLE wls_hvst;

CREATE TABLE wls_hvst (
  RECORDID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
  TIMESTAMP BIGINT 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.

6.3.2 Configuring JDBC Resources for WLDF

After you create the tables in your database, you must configure JDBC to access the tables. (See *Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*.) Then, as part of your server configuration, you specify that JDBC resource as the data store to be used for a server's archive.

An example configuration for a JDBC-based store is shown in [Example 6-3](#).

Example 6-3 Sample configuration for JDBC-based Diagnostic Archive (in config.xml)

```

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

```

```
</server>
<!-- Other server configurations in this domain -->
</domain>
```

If you specify a JDBC resource but it is configured incorrectly, or if the required tables do not exist in the database, WLDF uses the default file-based persistent store.

6.4 Retiring Data from the Archives

WLDF includes a configuration-based data retirement feature for periodically deleting old diagnostic data from the archives. You can configure size-based data retirement at the server level and age-based retirement at the individual archive level, as described in the following sections.

6.4.1 Configuring Data Retirement at the Server Level

You can set the following data retirement options for a server instance:

- The preferred maximum size of the server instance's data store (<preferred-store-size-limit>) and the interval at which it is checked, on the hour, to see if it exceeds that size (<store-size-check-period>).

When the size of the store is found to exceed the preferred maximum, an appropriate number of the oldest records in the store are deleted to reduce the size below the specified threshold. This is called "size-based data retirement."

Note: Size-based data retirement can be used only for file-based stores. These options are ignored for database-based stores.

- Enable or disable data retirement for the server instance.

For file-based diagnostic stores, this enables or disables the size-based data retirement options discussed above. For both file-based stores and database-based stores, this also enables or disables any age-based data retirement policies defined for individual archives in the store. See [Section 6.4.2, "Configuring Age-Based Data Retirement Policies for Diagnostic Archives,"](#) below.

6.4.2 Configuring Age-Based Data Retirement Policies for Diagnostic Archives

The data store for a server instance can contain the following types of diagnostic data archives whose records can be retired using the data retirement feature:

- Harvested metrics data (logical name: HarvestedDataArchive)
- Instrumentation events data (logical name: EventsDataArchive)
- Custom data (user-defined name)

Note: WebLogic Server log files are maintained both at the server level and the domain level. Data is retired from the current log using the log rotation feature. See "Configuring WebLogic Logging Services" in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

Age-based policies apply to individual archives. The data store for a server instance can have one age-based policy for the `HarvestedDataArchive`, one for the `EventsDataArchive`, and one each for any custom archives.

When records in an archive exceed the age limit specified for records in that archive, those records are deleted.

6.4.3 Sample Configuration

Data retirement configuration settings are persisted in the `config.xml` configuration file for the server's domain, as shown in [Example 6-4](#).

Example 6-4 Data Retirement Configuration Settings in `config.xml`

```
<domain>
<!-- other domain configuration settings -->
  <server>
    <name>MedRecServer</name>
    <!-- other server configuration settings -->
    <server-diagnostic-config>
      <diagnostic-store-dir>data/store/diagnostics</diagnostic-store-dir>
      <diagnostic-data-archive-type>FileStoreArchive
        </diagnostic-data-archive-type>
      <data-retirement-enabled>true</data-retirement-enabled>
      <preferred-store-size-limit>120</preferred-store-size-limit>
      <store-size-check-period>1</store-size-check-period>
      <wldf-data-retirement-by-age>
        <name>HarvestedDataRetirementPolicy</name>
        <enabled>true</enabled>
        <archive-name>HarvestedDataArchive</archive-name>
        <retirement-time>1</retirement-time>
        <retirement-period>24</retirement-period>
        <retirement-age>45</retirement-age>
      </wldf-data-retirement-by-age>
      <wldf-data-retirement-by-age>
        <name>EventsDataRetirementPolicy</name>
        <enabled>true</enabled>
        <archive-name>EventsDataArchive</archive-name>
        <retirement-time>10</retirement-time>
        <retirement-period>24</retirement-period>
        <retirement-age>72</retirement-age>
      </wldf-data-retirement-by-age>
    </server-diagnostic-config>
  </server>
</domain>
```

Configuring the Harvester for Metric Collection

This chapter describes the Harvester component of the WebLogic Diagnostics Framework (WLDF) that gathers metrics from attributes on qualified MBeans instantiated in a running server. The Harvester can collect metrics from WebLogic Server MBeans and from custom MBeans.

This chapter includes the following sections:

- [Harvesting, Harvestable Data, and Harvested Data](#)
- [Harvesting Data from the Different Harvestable Entities](#)
- [Configuring the Harvester](#)

7.1 Harvesting, Harvestable Data, and Harvested Data

Harvesting metrics is the process of gathering data that is useful for monitoring the system state and 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 run-time MBean server. Only simple type attributes of an MBean can be harvestable.
- *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 be currently registered with the JMX server.
 - 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 this MBean, see the description of the

weblogic.management.runtime.WLDFHarvesterRuntimeMBean in the *Oracle WebLogic Server MBean Reference*.

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

7.2 Harvesting Data from the Different Harvestable Entities

You can configure the Harvester to harvest 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 7-1](#).

Table 7-1 Sources of Harvested Data from Different Configurations

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

All WebLogic Server run-time 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 run-time MBean types and attributes. As types are instantiated, those instances also become known and thus 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 for custom MBeans, the list contains only the currently discovered types. See "Configure metrics to collect in a diagnostic system module" in the *Oracle WebLogic Server Administration Console Online Help*.

7.3 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.

[Example 7-1](#) 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 element in the listing.

Example 7-1 Sample Harvester Configuration (in DIAG_MODULE.xml)

```

<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-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>

```

7.3.1 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 $T+A$. If this occurs, the Harvester tries to start the next cycle sooner, to ensure that the average interval is I .

7.3.2 Configuring the Types of Data to Harvest

One or more `<harvested-type>` elements determine the types of data to harvest. 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. However, you can use pattern-matching to specify instance names in non-canonical form, as described in [Section C.1, "Using Wildcards in Harvester Instance Names."](#)
- 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.

The <harvested-attribute> element also supports an expression syntax for "drilling down" into attributes that are complex or aggregate objects, such as lists, maps, simple POJOs (Plain Old Java Objects), and various nestings of these types. See [Section C.2, "Specifying Complex and Nested Harvester Attributes,"](#) for details on this syntax. However, note that the result of these expressions must be a simple intrinsic type (int, boolean, String, and so on) in order to be harvested.

- If no <harvested-attribute> is present, all harvestable attributes defined for the type are collected.
- Attribute and instance lists can be combined in a type.

7.3.3 Specifying Type Names for WebLogic Server MBeans and Custom MBeans

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

There is a difference in how WebLogic Server and customer types are specified. For WebLogic Server types, the type name is the name of the Java interface that defines the MBean. For example, the server run-time MBean's type name is `weblogic.management.runtime.ServerRuntimeMBean`.

For custom MBeans, the Harvester follows these rules:

- If the MBean is not a ModelMBean, the type name is the implementing class name. (For example, see [Example 7-1.](#))
- If the MBean is a ModelMBean, the type name is the value of the MBean Descriptor field `DiagnosticTypeName`.

If neither of these conditions is satisfied (if the MBean is a ModelMBean and there is no value for the MBean Descriptor field `DiagnosticTypeName`) then the MBean can't be harvested.

7.3.4 Harvesting from the DomainRuntime MBeanServer

The <harvested-type> element supports a <namespace> attribute that lets you harvest metrics from MBeans registered in the DomainRuntime MBeanServer. However, Oracle recommends that you limit the usage to harvesting only DomainRuntime-specific MBeans, such as the `ServerLifecycleRuntimeMBean`. Harvesting of remote managed server MBeans through the DomainRuntime MBeanServer is possible, but is discouraged for performance reasons. It is a best practice to use the resident Harvester in each managed server to capture metrics related to that managed server instance.

The <namespace> attribute can have one of two values:

- `ServerRuntime`
- `DomainRuntime`

If the <namespace> attribute is omitted, it defaults to `ServerRuntime`.

Note: Harvesting from the DomainRuntime MBean server is available only on the Administration Server. Attempts to harvest DomainRuntime MBeans on a Managed Server are ignored. For an example, see [Example 7-5](#).

7.3.5 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.

7.3.6 Sample Configurations for Different Harvestable Types

In [Example 7-2](#), the `<harvested-type>` element in the `DIAG_MODULE.xml` configuration file 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, because there is always only one instance of the server run-time 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 for each of the two instances.

Example 7-2 Sample Configuration for Collecting All Instances and All Attributes of a Type (in `DIAG_MODULE.xml`)

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

In [Example 7-3](#), the `<harvested-type>` element in the `DIAG_MODULE.xml` configuration file 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`.

Example 7-3 Sample Configuration for Collecting Specified Attributes of All Instances of a Type (in `DIAG_MODULE.xml`)

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

In [Example 7-4](#), the `<harvested-type>` element in the `DIAG_MODULE.xml` configuration file specifies that a single instance of a custom MBean type is to be harvested. Because this is a custom MBean, the type 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. Because no instances of `<harvested-attribute>` are specified, all attributes will be harvested.

Example 7-4 Sample Configuration for Collecting All Attributes of a Specified Instance of a Type (in DIAG_MODULE.xml)

```
<harvested-type>
  <name>myMBeans.MySimpleStandard</name>
  <harvested-instance>myCustomDomain:Name=myCustomMBean1
</harvested-instance>
  <harvested-instance>myCustomDomain:Name=myCustomMBean2
</harvested-instance>
</harvested-type>
```

In [Example 7-5](#), the `<harvested-type>` element in the `DIAG_MODULE.xml` configuration file specifies that the `ServerLifecycleRuntimeMBean` is to be harvested. The `<namespace>` attribute specifies that this is a `DomainRuntime` MBean, so this configuration will only be honored on the administration server (see the note in [Section 7.3.4, "Harvesting from the DomainRuntime MBeanServer"](#)). The sub-element `<harvested-attribute>` specifies that only the `StateVal` attribute will be harvested.

Example 7-5 Sample configuration for Collecting Specified Attributes of the ServerLifecycleMBean Type (in DIAG_MODULE.xml)

```
<harvested-type>
  <name>weblogic.management.runtime.ServerLifecycleRuntimeMBean</name>
  <namespace>DomainRuntime</namespace>
  <known-type>true</known-type>
  <harvested-attribute>StateVal</harvested-attribute>
</harvested-type>
```

Configuring Watches and Notifications

This chapter describes the Watch and Notification component of the WebLogic Diagnostics Framework (WLDF) that 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 server instances in a domain.

This chapter includes the following sections:

- [Watches and Notifications](#)
- [Overview of Watch and Notification Configuration](#)
- [Sample Watch and Notification Configuration](#)

8.1 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

You must associate a watch 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.

8.2 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 descriptor file, *DIAG_MODULE.xml*, are shown in [Example 8-1](#). As the listing shows, the base 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>`.

Example 8-1 A Skeleton Watch and Notification Configuration (in *DIAG_MODULE.xml*)

```
<wldf-resource>
<!-- ----- Other system resource configuration elements ----- -->
  <watch-notification>
    <log-watch-severity>
      <!-- Threshold severity for a log watch to be evaluated further
           (This can be narrowed further at the watch level.) -->
    </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.

For information about how to configure watches and notifications using the Administration Console, see "Configure Watches and Notifications" in *Oracle WebLogic Server Administration Console Online Help*.

8.3 Sample Watch and Notification Configuration

A complete configuration for a set of watches and notifications in a diagnostic module is shown in [Example 8-2](#). The details of this example are explained in the following two sections:

- [Chapter 9, "Configuring Watches"](#)
- [Chapter 10, "Configuring Notifications"](#)

Example 8-2 Sample Watch and Notification Configuration (in `DIAG_MODULE.xml`)

```
<?xml version='1.0' encoding='UTF-8'?>
<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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 does not have to be configured and enabled for harvester
watches. However, configuring the Harvester can provide advantages;
for example the data will be archived. -->
<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//SocketsOpenedTotalCou
nt} &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>
```

Configuring Watches

This chapter describes the types of watches available in the WebLogic Diagnostics Framework (WLDF) and their configuration options

This chapter includes the following sections:

- [Types of Watches](#)
- [Configuration Options Shared by All Types of Watches](#)
- [Configuring Harvester Watches](#)
- [Configuring Log Watches](#)
- [Configuring Instrumentation Watches](#)
- [Defining Watch Rule Expressions](#)

For information about how to create a watch using the Administration Console, see "Create watches for a diagnostic system module" in *Oracle WebLogic Server Administration Console Online Help*.

9.1 Types of Watches

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

- **Harvester** watches monitor the set of harvestable MBeans in the local run-time 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 Instrumentation watches are triggered in real time, whereas Harvester watches are triggered only after the current harvest cycle completes.

9.2 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 expression 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 10, "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 `WatchNotificationRuntimeMBean` 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).

9.3 Configuring Harvester Watches

A Harvester watch can monitor any run-time MBean in the local run-time 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 7, "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 through a notification. On average, the delay is $\text{SamplePeriod}/2$.

[Example 9-1](#), shows a configuration example of a Harvester watch that monitors several run-time MBeans. When the watch rule (defined in the `<rule-expression>` element) 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.

The watch rule is a logical expression composed of four Harvester variables. The rule has the form:

```
( (A >= 100) AND (B > 0) ) OR C OR D.equals("active")
```

Each variable is of the form:

```
{entityName}/{attributeName}
```

where `{entityName}` is the JMX ObjectName as registered in the run-time MBean server or the type name as defined by the Harvester, and where `{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.

Example 9-1 Sample Harvester Watch Configuration (in `DIAG_MODULE.xml`)

```
<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <harvester>
  <!-- Harvesting does not have to be configured and enabled for harvester
    watches. However, configuring the Harvester can provide advantages;
    for example the data will be archived. -->
  <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=WLDHArvesterRuntime,ServerRuntime=myserver,Type=
    WLDHArvesterRuntime,WLDHRuntime=WLDHRuntime//TotalSamplingTime}
    >= 100
    AND
    ${mydomain:Name=myserver,Type=
    ServerRuntime//OpenSocketsCurrentCount} > 0)
    OR
    ${mydomain:Name=WLDHWatchNotificationRuntime,ServerRuntime=
    myserver,Type=WLDHWatchNotificationRuntime,
    WLDHRuntime=WLDHRuntime//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.

9.4 Configuring Log Watches

Use Log watches to monitor the occurrence of specific messages and/or strings in the server log. Watches of this type are triggered as a result of a log message containing the specified data being issued.

An example configuration for a log watch is shown in [Example 9-2](#).

Example 9-2 Sample Configuration for a Log Watch (in `DIAG_MODULE.xml`)

```

<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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>

```

9.5 Configuring Instrumentation Watches

You use Instrumentation watches to monitor the events from the WLDf Instrumentation component. Watches of this type are triggered as a result of the event being posted.

[Example 9-3](#) shows an example configuration for an Instrumentation watch.

Example 9-3 Sample Configuration for an Instrumentation Watch (in *DIAG_MODULE.xml*)

```

<watch-notification>
  <watch>
    <name>myInstWatch</name>
    <enabled>true</enabled>
    <rule-type>EventData</rule-type>
    <rule-expression>
      (PAYLOAD > 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>

```

9.6 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."](#)

Configuring Notifications

This chapter describes the types of notifications available in the WebLogic Diagnostics Framework (WLDF) and their configuration options.

This chapter includes the following sections:

- [Types of Notifications](#)
- [Configuring JMX Notifications](#)
- [Configuring JMS Notifications](#)
- [Configuring SNMP Notifications](#)
- [Configuring SMTP Notifications](#)
- [Configuring Image Notifications](#)

For information about how to create a notification using the Administration Console, see "Create notifications for watches in a diagnostic system module" in *Oracle WebLogic Server Administration Console Online Help*.

10.1 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 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: To define notifications programmatically, use `weblogic.diagnostics.watch.WatchNotification`.

10.2 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. You can also specify a JMX "notification type" string that a JMX client can use as a filter.

[Example 10-1](#) shows an example of a JMX notification configuration.

Example 10-1 Example Configuration for a JMX Notification

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

  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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>
```

Here is an example of a JMX notification:

```
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
```

10.3 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.

[Example 10-2](#) 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, and the JMS resource must be targeted to this server.

Example 10–2 Example JMS Notifications

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

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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.

10.4 Configuring SNMP Notifications

Simple Network Management Protocol (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 [Example 10–3](#). 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.

Example 10–3 An Example Configuration for an SNMP Notification

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

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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 [Example 10–3](#) 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)

```

10.5 Configuring SMTP Notifications

Simple Mail Transfer Protocol (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, first configure the SMTP session. That configuration is persisted in the config.xml configuration file for the domain. In *DIAG_MODULE.xml*, you 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.

[Example 10-4](#) shows an SMTP notification that causes an SMTP (e-mail) message to be distributed through the configured SMTP session, to the configured recipients. 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.

Example 10-4 Sample Configuration for SMTP Notification (in *DIAG_MODULE.xml*)

```

<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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.

10.6 Configuring Image Notifications

An image notification causes a diagnostic image to be generated in response to the triggering of an associated watch. You can configure two options for image notifications: a directory and a lockout period.

The directory name indicates where images will be generated. The lockout period determines the number of seconds that must elapse before a new image can be generated after the last one. This is useful for limiting the number of images that will be generated when there is a sequence of server failures and recoveries.

You can specify the directory name relative to the *DOMAIN_NAME*\servers*SERVER_NAME*, directory 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.

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

The configuration is persisted in the *DIAG_MODULE.xml* configuration file.

[Example 10-5](#) shows an image notification configuration that specifies that the lockout time will be two minutes and that the image will be generated to the *DOMAIN_NAME*\servers*SERVER_NAME*\images directory.

Example 10-5 Sample Configuration for Image Notification (in *DIAG_MODULE.xml*)

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

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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>
```

For more information about Diagnostic Images, see [Chapter 5, "Configuring and Capturing Diagnostic Images."](#)

Configuring Instrumentation

This chapter describes the Instrumentation component of the WebLogic Diagnostics Framework (WLDF) that provides a mechanism for adding diagnostic code to 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 that is inserted into server or application code at specific locations. You define monitors 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 unique request identifier and flags that indicate the presence of certain request properties such as originating IP address or user identity. The diagnostic context provides a means for tracking program execution and for controlling when monitors trigger their diagnostic actions. See [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts."](#)

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

This chapter includes the following sections:

- [Concepts and Terminology](#)
- [Instrumentation Configuration Files](#)
- [XML Elements Used for Instrumentation](#)
- [Configuring Server-Scoped Instrumentation](#)
- [Configuring Application-Scoped Instrumentation](#)
- [Creating Request Performance Data](#)

11.1 Concepts and Terminology

This section introduces instrumentation concepts and terminology. The following topics are included:

- [Instrumentation Scope](#)
- [Configuration and Deployment](#)
- [Joinpoints, Pointcuts, and Diagnostic Locations](#)

- [Diagnostic Monitor Types](#)
- [Diagnostic Actions](#)

11.1.1 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 levels. However, there are differences, which are discussed throughout this document. The term **server-scoped instrumentation** refers to instrumentation configuration and features specific to WebLogic Server instances and clusters. By contrast, **application-scoped instrumentation** refers to configuration and features specific to applications deployed on WebLogic servers. The scope is built in to each diagnostic monitor; you cannot modify a monitor's scope.

11.1.2 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 in the *ARCHIVE_PATH/META-INF* directory for the deployed application.

11.1.3 Joinpoints, Pointcuts, and Diagnostic Locations

Instrumentation code is inserted (or **woven**) 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, the entry point, or exit point, or both, of 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 that specifies a pointcut is `<pointcut>`. Pointcuts are described in [Section 11.5.4.1, "Defining Pointcuts for Custom Monitors."](#)
- A **diagnostic location** is the position relative to a joinpoint where the diagnostic activity will take place. Diagnostic locations are **Before**, **After**, and **Around**. The XML element that identifies a diagnostic location is `<location-type>`.

11.1.4 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, *Servlet_Around_Service* is an application-scoped delegating monitor that can be used to trigger diagnostic actions at the entry to and exit from specific servlet and JSP methods.

There are three types of diagnostic monitors:

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

The only standard server-scoped monitor is the DyeInjection monitor, which you can use to create diagnostic context and to configure dye injection at the server level. For more information, see [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts."](#)

The only standard application-scoped monitor is HttpSessionDebug, which you can use to inspect an HTTP Session object.

- A **delegating monitor** has its scope, pointcuts, and locations hard-coded in the monitor, but you select the actions that the monitor performs. That is, 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. To have a delegating monitor perform useful work, you must assign at least one action to it.

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 configure a delegating monitor using WLST or by editing a descriptor file manually, you must make sure that the actions are compatible with that monitor. WLDF validates a delegating monitor when its XML configuration 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 delegating monitor that:
 - Is available only for application-scoped instrumentation
 - Does not have a predefined pointcut or location

To configure a custom monitor, you assign it a name, define the pointcut and the diagnostics location that the monitor uses, and assign actions from the set of predefined diagnostic actions. The <pointcut> and <location type> elements are mandatory for a custom monitor.

[Table 11–1](#) summarizes the differences among the types of monitors.

Table 11–1 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

You can restrict when a diagnostic action is triggered by setting a **dye mask** on a monitor. This mask determines the dye flags in the diagnostic context that trigger actions. See [Section 11.3.2, "<wldf-instrumentation-monitor> XML Elements,"](#) for information about setting a dye mask for a monitor.

Note: Diagnostic context, dye injection, and dye filtering are described in [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts."](#)

11.1.5 Diagnostic Actions

Diagnostic actions execute diagnostic code that is appropriate for the associated delegating or custom monitor (standard monitors have predefined actions). For a delegating or custom monitor to perform any useful work, you must configure at least one action for that 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 of the `DIAG_MODULE.xml` configuration file:

- `DisplayArgumentsAction`
- `MethodInvocationStatisticsAction`
- `MethodMemoryAllocationStatisticsAction`
- `StackDumpAction`
- `ThreadDumpAction`
- `TraceAction`
- `TraceElapsedTimeAction`
- `TraceMemoryAllocationAction`

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.

11.2 Instrumentation Configuration Files

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

- System-level instrumentation configuration is stored in one or more diagnostics descriptors in the following directory:

```
DOMAIN_NAME/config/diagnostics
```

This directory can contain multiple system-level diagnostic descriptor files. File names are arbitrary but must be terminated with `.xml`; for example, `myDiag.xml`. Each file can contain configuration information for one or more of the following deployable diagnostic components:

- Harvester
- Instrumentation
- Watch and Notification

The configuration of one or more diagnostic monitors can be defined in an `<instrumentation>` section in the descriptor file. Server-scoped instrumentation can be enabled, disabled, and reconfigured without restarting the server.

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

```
DOMAIN_NAME/config/config.xml
```

For more information about configuring diagnostic system modules, see [Section 4.6, "Configuring Diagnostic System Modules."](#) For general information about the creation, content, and parsing of configuration files in WebLogic Server, see *Understanding Domain Configuration for Oracle WebLogic Server*.

- 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 descriptor can contain only instrumentation configuration information.

Note: For instrumentation to be available for an application, instrumentation must be enabled on the server to which the application is deployed. (Server-scoped instrumentation is enabled and disabled in the <instrumentation> element of the diagnostics descriptor for the server.)

You can enable and disable diagnostic monitors without redeploying an application. However, you may need to redeploy the application after modifying other instrumentation features; for example, defining pointcuts or adding or removing monitors. Whether you need to redeploy depends on how you configure the instrumentation and how you deploy the application. There are three options:

- Define and change the instrumentation configuration for the application directly, without using a JSR-88 deployment plan
- Configure and deploy the application using a deployment plan that has placeholders for instrumentation settings
- Enable the HotSwap feature when starting the server, and deploy using a deployment plan that has placeholders for instrumentation settings

For more information about these choices, see [Section 14.2, "Using Deployment Plans to Dynamically Control Instrumentation Configuration."](#)

For more information about deploying and modifying diagnostic application modules, see [Chapter 14, "Deploying WLDF Application Modules."](#)

The diagnostics XML schema is located at:

```
http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-diagnostics.xsd
```

Each diagnostics descriptor file must begin with the following lines:

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

For an overview of WLDF resource configuration, see [Chapter 4, "Understanding WLDF Configuration."](#)

11.3 XML Elements Used for Instrumentation

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

- [<Instrumentation> XML Elements](#) describes the top-level elements used within an `<instrumentation>` element.
- [<wldf-instrumentation-monitor> XML Elements](#) describes the elements used within a `<wldf-instrumentation-monitor>` element.
- [Mapping <wldf-instrumentation-monitor> XML Elements to Monitor Types](#) identifies the instrumentation elements that apply to each monitor.

11.3.1 <Instrumentation> XML Elements

[Table 11–2](#) describes the `<instrumentation>` elements in the `DIAG_MODULE.xml` file. The following configuration fragment illustrates the use of those elements:

```
<wldf-resource>
  <name>MyDiagnosticModule</name>
  <instrumentation>
    <enabled>true</enabled>
    <!-- The following <include> element would apply only to an
         application-scoped Instrumentation descriptor -->
    <include>foo.bar.com.*</include>
    <!-- &lt;wldf-instrumentation-monitor&gt; elements to define diagnostic
         monitors for this diagnostic module -->
  </instrumentation>
  <!-- Other elements to configure this diagnostic module -->
</wldf-resource>
```

Table 11–2 *<instrumentation> XML Elements in the DIAG_MODULE.xml Configuration File*

Element	Description
<code><instrumentation></code>	The element that begins an instrumentation configuration.
<code><enabled></code>	<p>If true, instrumentation is enabled. If false, no instrumented code is inserted in classes in this instrumentation scope, and all diagnostic monitors within this scope are disabled. The default value is false.</p> <p>You must enable instrumentation at the server level to enable instrumentation for the server and for any applications deployed to it. You must further enable instrumentation at the application level to enable instrumentation for the application (that is, in addition to enabling the server-scoped instrumentation).</p>

Table 11-2 (Cont.) <instrumentation> XML Elements in the DIAG_MODULE.xml Configuration File

Element	Description
<include>	<p>An optional element specifying the list of classes where instrumented code can be inserted. Wildcards (*) are supported. You can specify multiple <include> elements. If specified, a class must satisfy an <include> pattern for it to be instrumented.</p> <p>Applies only to application-scoped instrumentation. Any specified <include> or <exclude> patterns are applied to the application scope as a whole.</p> <p>Note: You can also specify <include> and <exclude> patterns for specific diagnostic monitors. See the entries for <include> and <exclude> in Table 11-1.</p> <p>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.</p> <p>Note: Instrumentation is inserted in applications at class load time. A large application that is loaded often may benefit from a judicious use of <include> elements, <exclude> elements, or both. You can probably ignore these elements for small applications or for medium-to-large applications that are loaded infrequently.</p>
<exclude>	<p>An optional element specifying the list of classes where instrumented code cannot be inserted. Wildcards (*) are supported. You can specify multiple <exclude> elements. If specified, classes satisfying an <exclude> pattern are not instrumented.</p> <p>Applies only to application-scoped instrumentation. See the preceding description of the <include> element.</p>

11.3.2 <wldf-instrumentation-monitor> XML Elements

Diagnostic monitors are defined in <wldf-instrumentation-monitor> elements, which are children of the <instrumentation> element in the following descriptor:

- The *DIAG_MODULE.xml* descriptor for server-scoped instrumentation
- The *META-INF/weblogic-diagnostics.xml* descriptor for application-scoped instrumentation

The following fragment shows the configuration for a delegating monitor and a custom monitor in an application. (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 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts,"](#) for information about configuring the `DyeInjection` monitor.

[Table 11–3](#) describes the `<wldf-instrumentation-monitor>` elements.

Table 11–3 `<wldf-instrumentation-monitor>` XML Elements in the `DIAG_MODULE.xml` or `weblogic-diagnostics.xml` file

Element	Description
<code><wldf-instrumentation-monitor></code>	The element that begins a diagnostic monitor configuration.
<code><enabled></code>	If true, the monitor is enabled. If false, the monitor is disabled. You enable or disable each monitor separately. The default value is true.
<code><name></code>	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. The name for a custom monitor must be unique; that is, it cannot duplicate the name of any monitor in the library.
<code><description></code>	An optional element describing the monitor.
<code><action></code>	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 <code><action></code> 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."
<code><dye-filtering-enabled></code>	An optional element. If true, dye filtering is enabled for the monitor. If false, dye-filtering is disabled. The default value is false. In order to use dye filtering, the <code>DyeInjection</code> monitor must be configured appropriately at the server level.
<code><dye-mask></code>	An optional element. If dye filtering is enabled, the dye mask, when compared with the values in the diagnostic context, determines whether actions are taken. See Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts," for information about dyes and dye filtering.
<code><properties></code>	An optional element. Sets name=value pairs for dye flags. Currently applies only to the <code>DyeInjection</code> monitor.

Table 11-3 (Cont.) <wldf-instrumentation-monitor> XML Elements in the DIAG_MODULE.xml or weblogic-diagnostics.xml file

Element	Description
<location-type>	<p>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.</p> <p>Applies only to custom monitors; standard and delegating monitors have predefined location types. A custom monitor must define a location type and a pointcut.</p>
<pointcut>	<p>An optional element. A pointcut element contains an expression that defines joinpoints where diagnostic code will be inserted.</p> <p>Applies only to custom monitors; standard and delegating monitors have predefined pointcuts. A custom monitor must define a location type and a pointcut.</p> <p>Pointcut syntax is documented in Section 11.5.4.1, "Defining Pointcuts for Custom Monitors."</p>
<include>	<p>An optional element specifying the list of classes where instrumented code can be inserted. Wildcards (*) are supported. You can specify multiple <include> elements. If specified, a class must satisfy an <include> pattern for it to be instrumented.</p> <p>Applies only to application-scoped instrumentation. Any specified <include> or <exclude> patterns are applied only to the monitor defined in the parent <wldf-instrumentation-monitor> element.</p> <p>Note: You can also specify <include> and <exclude> patterns for an entire instrumented application scope. See the entries for <include> and <exclude> in Table 11-1.</p> <p>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.</p> <p>Note: Instrumentation is 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.</p>
<exclude>	<p>An optional element specifying the list of classes where instrumented code cannot be inserted. Wildcards (*) are supported. You can specify multiple <exclude> elements. If specified, classes satisfying an <exclude> pattern are not instrumented.</p> <p>Applies only to diagnostic monitors in application-scoped instrumentation. See the <include> description, above.</p>

Note the following additional information about the <dye-filtering-enabled> and <dye-mask> elements:

- 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 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts,"](#) for detailed information about diagnostic contexts and dye vectors.

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

[Table 11-4](#) identifies the <wldf-instrumentation-monitor> elements that apply to each monitor type. An X indicates that an element applies to the corresponding monitor; N/A indicates that it does not.

Table 11-4 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>	N/A	X	X
<dye-filtering-enabled>	N/A	X	X
<dye-mask>	N/A	X	X
<properties>	X ¹	N/A	N/A
<location-type>	N/A	N/A	X
<pointcut>	N/A	N/A	X

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

11.4 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 WLDF system resources you want to create.

You can have multiple *DIAG_MODULE.xml* diagnostic descriptor files in a domain. But for each server or cluster in a domain, you can deploy only one diagnostic descriptor file at any given time. One reason for creating more than one file is for flexibility. For example, you could have 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 the descriptor file that corresponds to the particular monitors you want active.

2. Decide which server-scoped monitors you want to include in a configuration:
 - If you plan to use dye filtering on a server, or on any applications 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 and configure the configuration file(s).
 - If you use the Administration Console to create the `DIAG_MODULE.xml` file (recommended), for delegating monitors the console displays only the 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 Oracle WebLogic Server* for information about configuring `config.xml`.
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.

Example 11-1 contains a sample server-scoped instrumentation configuration file that 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.

Example 11-1 Sample Server-Scoped Instrumentation (in `DIAG_MODULE.xml`)

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

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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>
```

11.5 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](#)
- [Overview of the Steps Required to Instrument an Application](#)
- [Creating a Descriptor File for a Delegating Monitor](#)
- [Creating a Descriptor File for a Custom Monitor](#)
- [Defining Pointcuts for Custom Monitors](#)
- [Annotation-based Pointcuts](#)

11.5.1 Comparing System-Scoped to Application-Scoped Instrumentation

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

- Applications can use standard, delegating, and custom monitors.
 - The only server-scoped standard monitor is DyeInjection. The only application-scoped standard monitor is HttpSessionDebug. For more information, see the entry for HttpSessionDebug in [Section B.1, "Diagnostic Monitor Library."](#)
 - 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 server instrumentation is enabled at the time of deployment, instrumentation will be available for the application. If instrumentation is not enabled on the server at the time of deployment, 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 the instrumentation, and put the file in the application's archive. When the archive is deployed, the instrumentation is automatically inserted when the application is loaded.
- You can use a *deployment plan* to dynamically update configuration elements without redeploying the application. See [Section 14.2, "Using Deployment Plans to Dynamically Control Instrumentation Configuration."](#)

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 also create your own custom monitors; however, the diagnostic actions that you attach to these monitors must be taken from the WLDF Instrumentation Library.

[Table 11–5](#) compares the function and scope of system and application diagnostic modules.

Table 11–5 Comparing System and Application Modules

Module Type	Add or Remove Objects Dynamically	Add or Remove Objects with Console	Modify with JMX Remotely	Modify with JSR-88 (non-remote)	Modify with Console	Enable/Disable Dye Filtering and Dye Mask Dynamically
System Module	Yes	Yes	Yes	No	Yes (via JMX)	Yes
Application Module	Yes, when HotSwap is enabled No, when HotSwap is not enabled: module must be redeployed	Yes	No	Yes	Yes (via plan)	Yes

11.5.2 Overview of the Steps Required to Instrument an Application

Note: As of WebLogic Server 10.3, you are not required to create a `weblogic-diagnostics.xml` file in the application's META-INF directory, as was the case in previous WLS releases. However, you can still use this method to initially configure diagnostic monitors for your application.

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

1. Make sure that instrumentation is enabled on the server. See [Section 11.4, "Configuring Server-Scoped Instrumentation."](#)
2. Create a well formed META-INF/weblogic-diagnostics.xml descriptor file for the application. If you want to add any monitors that will be automatically enabled each time the application is deployed:
 - 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 [Section 11.5.3, "Creating a Descriptor File for a Delegating Monitor,"](#) and [Section 11.5.4, "Creating a Descriptor File for a Custom Monitor,"](#) for samples of well-formed descriptor files.

See [Section 11.5.4.1, "Defining Pointcuts for Custom Monitors,"](#) for information about creating a pointcut expression.

3. Put the descriptor file in the application archive.
4. Deploy the application. See [Chapter 14, "Deploying WLDF Application Modules."](#)

Keep the following points in mind:

- The diagnostic monitors defined in `weblogic-diagnostics.xml` will be listed on the **Deployments: <server_name>: Configuration: Instrumentation** page of the Administration Console.

- If the `META-INF/weblogic-diagnostics.xml` descriptor in the application archive defines a monitor, it can't be removed using the Administration Console. However, it can be disabled or enabled using the Administration Console.
- You can add additional monitors from the Administration Console. Any monitors you add from the Administration Console will *not* be persisted to `weblogic-diagnostics.xml`; they will be saved in the application's deployment plan. Any monitors that were added in this way can be deleted using the Administration Console.

11.5.3 Creating a Descriptor File for a Delegating Monitor

The following example shows a well-formed `META-INF/weblogic-diagnostics.xml` descriptor file for an application-scoped delegating monitor. At a minimum, this file must contain the lines shown in bold. In this example, there is only one monitor defined (`Servlet_Before_Service`). However, you can define multiple monitors in the descriptor file.

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

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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 hard coded with a pointcut that sets joinpoints at method entry for several servlet or 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 as per the `DyeInjection` monitor configuration when the request enters the server. For example, if the `DyeInjection` monitor is configured with property `USER1=weblogic` and the request was originated by user `weblogic`, the `USER1` dye flag in the dye vector will be set.

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.

11.5.4 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. At a minimum, the file must contain the lines shown in bold.

Example 11–2 Sample Custom Monitor Configuration (in DIAG_MODULE.xml)

```

<?xml version="1.0" encoding="UTF-8"?>
<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-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 `TraceAction` action will be invoked before (`<location-type>before</location-type>`) any methods defined by the pointcut expression is invoked. [Table 11–6](#) shows how the pointcut expression from [Example 11–2](#) is parsed. (Note the use of wildcard characters.)

Table 11–6 Description of a Sample Pointcut Expression

Pointcut Expression	Description
<code>call(* com.foo.bar.* get* (...)</code>	<code>call()</code> : Trigger any defined actions when the methods whose joinpoints are defined by the remainder of this pointcut expression are invoked.
<code>call(* com.foo.bar.* get* (...)</code>	<code>*</code> : Return value. The wildcard indicates that the methods can have any type of return value.
<code>call(* com.foo.bar.* get* (...)</code>	<code>com.foo.bar.*</code> : Methods from class <code>com.foo.bar</code> and its sub-packages are eligible.
<code>call(* com.foo.bar.* get* (...)</code>	<code>get*</code> : Any methods whose name starts with the string <code>get</code> is eligible.
<code>call(* com.foo.bar.* get* (...)</code>	<code>(...)</code> : The ellipsis indicates that the methods can have any number of arguments.

This pointcut expression matches all methods in all classes in package `com.foo.bar` and its sub-packages. 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 these methods are called, and, just before those methods are called, the `TraceAction` action will be invoked.

See [Section 11.5.4.1, "Defining Pointcuts for Custom Monitors,"](#) for a description of the grammar used to define pointcuts.

11.5.4.1 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 a specific, well-defined location in a program. A pointcut is an expression that specifies a set of joinpoints. This section describes how you define expressions for pointcuts using the following 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 syntax for defining a pointcut expression is as follows:

```
pointcutExpr := orExpr ( 'OR' orExpr ) *
orExpr := andExpr ( 'AND' andExpr ) *
andExpr := 'NOT' ? termExpr
termExpr := exec_pointcut | call_pointcut | '(' pointcutExpr ')'
exec_pointcut := 'execution' '(' modifiers?
                returnSpec
                classSpecWithAnnotations
                methodSpec '(' parameterList ')'
                ')'
call_pointcut := 'call' '(' returnSpec
                classSpec
                methodSpec '(' parameterList ')'
                ')'
modifiers := modifier ( 'OR' modifier ) * modifier := 'public' | 'protected' |
'private' | 'static'
returnSpec := '*' | typeSpec
classSpecWithAnnotations := '@' IDENTIFIER ( 'OR' IDENTIFIER ) * | classSpec
classSpec := '+' ? classOrMethodPattern | '*'
typeSpec := '%' ? ( primitiveType | classSpec ) ( '[' ] ) *
methodSpec := classOrMethodPattern
parameterList := param ( ',' param ) *
param := typeSpec | '...'
primitiveType := 'byte' | 'char' | 'boolean' | 'short' | 'int' | 'float' | 'long'
| 'double' | 'void'
classOrMethodPattern := '*' ? IDENTIFIER '*'? | '*'
```

The following rules apply:

- The asterisk wildcard character (*) 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 percent character (%) prefix designates the value of a non-static class instantiation, parameter, or return specification as not containing nor exposing sensitive information. The use of this operator is particularly useful with the DisplayArgumentsAction action, which captures method arguments or return values. If this prefix character is not explicitly used, an asterisk string is substituted for the value that is returned; this behavior ensures that sensitive data in your application is not inadvertently transmitted when an instrumentation event captures input arguments to, or return values from, a joinpoint.

Note: The % operator cannot be applied to an ellipsis or to a wildcarded type within a pointcut expression.

- A plus sign (+) prefix to a class type identifies all subclasses, sub-interfaces 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 initialize methods in all classes in package `com.foo.bar` and its sub-packages. The initialize 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 (call sites) 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>
```

11.5.4.2 Annotation-based Pointcuts

You can use JDK-style annotations in class and method specifiers of execution points. A class or method specifier starting with `@` is interpreted as an annotation name.

When used as a class specifier, the annotation matches all classes that are annotated with it. While performing the match, only annotation names are considered. Annotation attributes are ignored.

For example, consider the following pointcut:

```
execution(public void @Service @Invocation (...))
```

The preceding pointcut matches methods that:

- Are public method
- Return void
- Are contained in a class that is annotated with `@Service`
- Have a method annotated with `@Invocation`
- Contain any number of arguments.

Note: Annotation-based specifiers can be used only with execution pointcuts. They cannot be used with call pointcuts.

Annotation-based class and method specifiers can use the following wildcard characters:

- The asterisk wildcard (*) matches everything.
- The asterisk wildcard (*) at the beginning matches class/interface or method names that end with the given string. For example, *Bean matches with `weblogic.management.configuration.ServerMBean`.
- The asterisk wildcard (*) at the end matches class/interface or method names that end with the given string. For example, `weblogic.*` matches all classes and interfaces that are in `weblogic` and its sub-packages.
- You can specify a pointcut based on names of inner classes. For example:

```
public class Foo {
    class Bar {
        public int getValue() {...}
    }
}
```

You can define a pointcut that covers the `getValue` method of the inner class `Bar` using the following specification:

```
execution (public int Foo$Bar getValue(...));
```

You can also use wildcard characters as follows. The following pointcut matches only the getter methods in the inner class `Bar` of class `Foo`:

```
execution ( * Foo$Bar get*(...));
```

You can also use leading and trailing wildcard characters. The following examples also match the getter methods in class `Foo$Bar`:

```
execution ( * Foo$Ba* get*(...));
execution ( * *oo$Bar get*(...));
execution ( * *oo$Ba* get*(...));
```

11.6 Creating Request Performance Data

If you have configured server-scoped or application-scoped instrumentation, you can display request performance data in the WebLogic Server Administration Console. The Request Performance page displays information about the real-time and historical views of method performance that has been captured by means of the WebLogic Diagnostics Framework instrumentation capabilities.

To create request performance data, the following criteria must be met:

- A WLDF system resource must be created and targeted to the server. Create the system resource as described in [Section 11.2, "Instrumentation Configuration Files."](#) You can do this using the WebLogic Server Administration Console or the WebLogic Scripting Tool (WLST).
- Instrumentation in the targeted WLDF system resource must be enabled.
- Application instrumentation must be enabled with a `weblogic-diagnostics.xml` descriptor, which you create in the application's `META-INF` directory, as described in [Section 11.2, "Instrumentation Configuration Files."](#)

- Application instrumentation descriptors must use `TraceElapsedTimeAction` diagnostic actions attached to `Around` diagnostic monitor types. For example, a descriptor could contain the following:

```
<instrumentation>
  <enabled>true</enabled>
  <wldf-instrumentation-monitor>
    <name>Connector_Around_Inbound</name>
    <action>TraceElapsedTimeAction</action>
  </wldf-instrumentation-monitor>
</instrumentation>
```

Note: WebLogic Server does not require the `weblogic-diagnostics.xml` descriptor to be pre-bundled in your application's archive in order to make instrumentation changes to a deployed application.

- If your application uses a deployment plan, and you enable Java HotSwap before deploying your application, you can make instrumentation changes at run time without redeploying your application.
- If your deployed application does not have a deployment plan and you modify the application's instrumentation configuration, the Administration Console automatically creates a deployment plan for you and prompts you for the location in which to save it.
- If Java HotSwap is not enabled in your deployment plan, or if you do not use a deployment plan, changes to some instrumentation settings require redeployment.

For more information, see [Chapter 14, "Deploying WLDf Application Modules."](#)

See [Appendix B, "WLDf Instrumentation Library,"](#) for a list of "Around" type monitors.

For information about creating and analyzing request performance data in the WebLogic Server Administration Console, see "Analyze request performance" in the *Oracle WebLogic Server Administration Console Online Help*.

Configuring the DyeInjection Monitor to Manage Diagnostic Contexts

This chapter describes the Instrumentation component of the WebLogic Diagnostics Framework (WLDF) that provides a way to uniquely identify requests (such as HTTP or RMI requests) and track them as they flow through the system. You can configure WLDF to check for certain characteristics (such as the originating user or client address) of every request that enters the system and attach a *diagnostic context* to the request. This allows you to take measurements (such as elapsed time) of specific requests to get an idea of how all requests are being processed as they flow through the system.

The diagnostic context consists of two pieces: a unique Context ID and a 64-bit dye vector that represents the characteristics of the request. The Context ID associated with a given request is recorded in the Event Archive and can be used to:

- *Throttle* instrumentation event generation, that is determine how often events are generated when specified conditions are met
- Associate log records with a request
- Filter searches of log or event records using the WLDF Accessor component (see [Chapter 13, "Accessing Diagnostic Data With the Data Accessor"](#)).

This chapter includes the following sections:

- [Contents, Life Cycle, and Configuration of a Diagnostic Context](#)
- [Overview of the Process](#)
- [Configuring the Dye Vector via the DyeInjection Monitor](#)
- [Configuring Delegating Monitors to Use Dye Filtering](#)
- [How Dye Masks Filter Requests to Pass to Monitors](#)
- [Using Throttling to Control the Volume of Instrumentation Events](#)
- [Using `weblogic.diagnostics.context`](#)

12.1 Contents, Life Cycle, and Configuration of a Diagnostic Context

A diagnostic context contains a unique *Context ID* and a 64-bit *dye vector*. The dye vector contains flags which are set to identify the characteristics of the diagnostic context associated with a request. Currently, 32 bits of the dye vector are used, one for each available dye flag (see [Table 12-1](#)).

12.1.1 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 diagnostic 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 a Context ID that is unique in the domain. Because the Context ID travels with the request, it is possible to determine the events and log entries associated with a given request as it flows through the system.

12.1.2 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, and so on.

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 flag ADDR1 is configured to indicate a request that originated from IP address 127.0.0.1.
- the flag ADDR2 is configured to indicate a request that originated from IP address 127.0.0.2.
- the flag USER1 is configured to indicate a request that originated from user admin@avitek.com.

If a request from IP address 127.0.0.1 enters the system from a user other than admin@avitek.com, the ADDR1 flag in the dye vector for the request is set. The ADDR2 and USER1 dye flags remain unset.

If a request from admin@avitek.com enters the system from an IP address other than 127.0.0.1 or 127.0.0.2, the USER1 flag in the dye vector for the request is set. The ADDR1 and ADDR2 dye flags remain unset.

If a request from admin@avitek.com from IP address 127.0.0.2 enters the system, both the USER1 and ADDR2 flags in the dye vector for this request are set. The ADDR1 flag remains unset.

Diagnostic and monitoring features that take advantage of the diagnostic context can examine the dye vector to determine if one or more attributes are present (that is, the associated flag is set). In the example above, you could configure a diagnostic monitor to trace every request that is dyed with ADDR1; that is, every request originating from IP address 127.0.0.1. You could also configure a diagnostic monitor that traces every request that is dyed with both ADDR1 and USER1; that is, every request originating from user admin@avitek.com at IP address 127.0.0.1 (requests from other users at 127.0.0.1 would *not* be traced).

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 [Section 12.3.3, "THROTTLE Dye Flag."](#)

For a list of the available dyes and the attributes they represent, see [Section 12.3.1, "Dyes Supported by the DyeInjection Monitor."](#) The process of configuring dye vectors and using them is discussed throughout the rest of this chapter.

12.1.3 Where Diagnostic Context Is Configured

Diagnostic context is configured as part of a diagnostic module. You use the DyeInjection monitor at the server level to configure the diagnostic context. The DyeInjection monitor is a *standard* diagnostic monitor, so you cannot modify its behavior. 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 diagnostic context to the request, setting the dye flags as appropriate. If the dye flags that are set for a request match the dye flags that are configured for a downstream diagnostic monitor, an event with the request's associated Context ID is added to the Event Archive. So, for example, if a request has only the USER1 and ADDR1 dye flags set, and there is a diagnostic monitor configured to trace requests with both the USER1 and ADDR1 flags set (but no other flags set), an event is added to the Event Archive.

For information about diagnostic monitor types, pointcuts (which define the joinpoints), and diagnostic actions, see [Chapter 11, "Configuring Instrumentation."](#)

12.2 Overview of the Process

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

1. Configure a dye vector via the DyeInjection Module. See [Section 12.3, "Configuring the Dye Vector via the DyeInjection Monitor."](#)
2. When any request enters the system, WLDF creates and instantiates a diagnostic context for the request. The context includes a unique Context ID and a dye vector.
3. The DyeInjection monitor, if enabled at the server level within a WLDF diagnostic module, examines the request to see if any of the configured dye values in the dye vector match attributes of the request. For example, it checks to see if the request originated from the user associated with USER1 or USER2, and it checks to see if the request came from the IP address associated with ADDR1 or ADDR2.
4. For each dye value that matches a request attribute, the DyeInjection monitor sets the associated dye bits within the diagnostic context. For example, if the DyeInjection monitor is configured with USER1=*weblogic*, USER2=*admin@avitek.com*, ADDR1=*127.0.0.1*, ADDR2=*127.0.0.2*, and the request originated from user *weblogic* at IP address *127.0.0.2*, it will set the USER1 and ADDR2 dye bits within the dye vector.
5. As the request flows through the system, the diagnostic context (which includes the dye vector) flows with it as well. 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 (USER1 and ADDR2). It does not contain the actual user name and IP address associated with USER1 and ADDR2.

Note: All dye vectors also contain one of the implicit PROTOCOL dyes, as explained in [Section 12.3, "Configuring the Dye Vector via the DyeInjection Monitor."](#)

6. The administrator configures a diagnostic monitor (either application-scoped or server-scoped) to be active within downstream code, setting the monitor's dye mask as USER1 and ADDR2. See [Section 12.4, "Configuring Delegating Monitors to Use Dye Filtering,"](#) for more information.

- The diagnostic monitor will perform its associated action(s) if the dye flags that are set in the diagnostic context's dye vector match the dye mask of the diagnostic monitor. See [Section 12.5, "How Dye Masks Filter Requests to Pass to Monitors,"](#) for more details. In this example, the monitor will perform its action(s) if the USER1 and ADDR2 flags are set in the dye vector. In addition, an event associated with the request will be written to the Event Archive.

12.3 Configuring the Dye Vector via the DyeInjection Monitor

To create diagnostic contexts for all requests coming into the system, you must:

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

You configure the DyeInjection monitor by assigning values to dyes. The available dye flags are described in [Table 12-1](#).

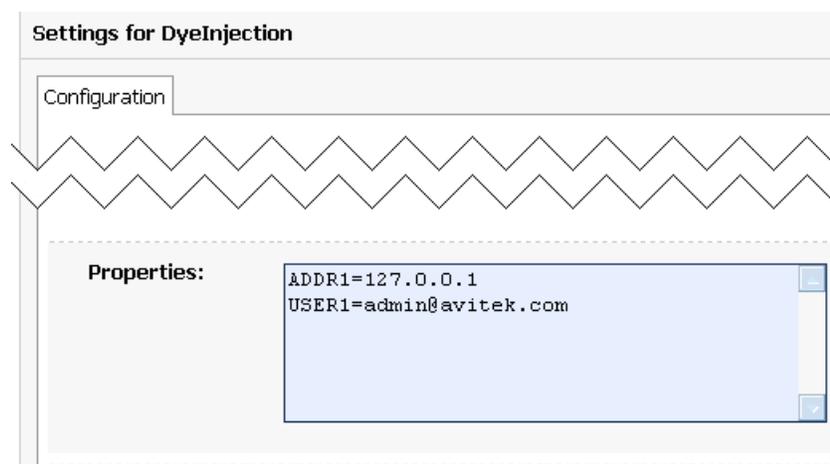
For example, you could set the flags as follows: USER1=*weblogic*, USER2=*admin@avitek.com*, ADDR1=*127.0.0.1*, ADDR2=*127.0.0.2*, and so forth.

Basically, you want to set the values of one or more flags to the user(s), IP address(es) whose requests you want to monitor.

For example, to monitor all requests initiated by a user named *admin@avitek* from a client at IP address *127.0.0.1*, assign the value *admin@avitek* to USER1 and assign the value *127.0.0.1* to ADDR1.

In the Administration Console, you assign values to dyes by typing them into the **Properties** field of the **Settings for DyeInjection** page. For instructions, see "Configure diagnostic monitors in a diagnostic system module" in the *Oracle WebLogic Server Administration Console Online Help*.

Figure 12-1 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.

Example 12–1 Sample DyeInjection Monitor Configuration, in DIAG_MODULE.xml

```

<wldf-resource>
  <name>MyDiagnosticModule</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>

```

12.3.1 Dyes Supported by the DyeInjection Monitor

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

Table 12–1 Request Protocols for Supported Diagnostic Context Dyes

Dye Flags	Description
ADDR1 ADDR2 ADDR3 ADDR4	Use the ADDR1, ADDR2, ADDR3 and ADDR4 dyes 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 originated from an IP address specified by the respective property (ADDR1, ADDR2, ADDR3, ADDR4) of the DyeInjection monitor. These dyes cannot be used to specify DNS names.
CONNECTOR1 CONNECTOR2 CONNECTOR3 CONNECTOR4	Use the CONNECTOR1, CONNECTOR2, CONNECTOR3 and CONNECTOR4 dyes 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 Section 12.7, "Using weblogic.diagnostics.context."

Table 12–1 (Cont.) Request Protocols for Supported Diagnostic Context Dyes

Dye Flags	Description
PROTOCOL_HTTP	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_IIOB	PROTOCOL_HTTP is set in the diagnostic context of a request if the request uses HTTP or HTTPS protocol.
PROTOCOL_JRMP	PROTOCOL_IIOB is set in the diagnostic context of a request if it uses Internet Inter-ORB Protocol (IIOB).
PROTOCOL_RMI	PROTOCOL_JRMP is set in the diagnostic context of a request if it uses the Java Remote Method Protocol (JRMP).
PROTOCOL_SOAP	PROTOCOL_RMI is set in the diagnostic context of a request if it uses the Java Remote Method Invocation (RMI) protocol.
PROTOCOL_SSL	PROTOCOL_SOAP is set in the diagnostic context of a request if it uses the Secure Sockets Layer (SSL) protocol.
PROTOCOL_T3	PROTOCOL_SSL 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	Use the USER1, USER2, USER3 and USER4 dyes 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.
USER2	
USER3	
USER4	

12.3.2 PROTOCOL Dye Flags

You must explicitly set the values for the dye flags `USERn`, `ADDRn`, `COOKIEn`, and `CONNECTORn` in the DyeInjection monitor. However, the flags `PROTOCOL_HTTP`, `PROTOCOL_IIOB`, `PROTOCOL_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.

12.3.3 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. See [Section 12.6, "Using Throttling to Control the Volume of Instrumentation Events,"](#) for more information.

12.3.4 When Diagnostic Contexts Are Created

When the DyeInjection monitor is enabled in a diagnostic module, a diagnostic context is created for every incoming request. The DyeInjection monitor is enabled by default when you enable instrumentation in a diagnostic module. This ensures that a diagnostic Context ID is available so that events can be correlated. Even if no properties are explicitly set in the DyeInjection monitor, the diagnostic 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 disabled, no diagnostic contexts will be created for any incoming requests.

12.4 Configuring Delegating Monitors to Use Dye Filtering

Note: For information about how to implement a diagnostic monitor for an application (such as a web application), see [Section 11.5.2, "Overview of the Steps Required to Instrument an Application."](#)

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 and a diagnostic event is generated only for those requests that meet the criteria set by the mask.

[Figure 12–2](#) shows an example of diagnostic events that were generated when a configured diagnostic action was triggered. Notice that the Context ID is the same for all of the events, indicating that they are related to the same request. You can use this Context ID to query for log records that are associated with the request. Note that the user ID associated with a request may not always be the same as the USER value you configured in the DyeInjection monitor; as a request is processed through the system, the user associated with the request may change to allow the system to perform certain functions (for example, the User ID may change to kernel).

Figure 12–2 Example of Diagnostic Events Associated with a Request

Date	Context ID	User ID	Type	Monitor	Class	Method
06/20/08 07:52:55 509	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-Before-1	Servlet_Around_Session	javax.servlet.http.HttpServletRequest	getSession
06/20/08 07:52:55 509	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-After-1	Servlet_Around_Session	javax.servlet.http.HttpServletRequest	getSession
06/20/08 07:53:18 272	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-Before-2	Servlet_Around_Service	jsp_servlet__index	_jspService
06/20/08 07:53:18 272	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceAction	Servlet_Before_Service	jsp_servlet__index	_jspService
06/20/08 07:53:18 272	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-Before-3	Servlet_Around_Session	javax.servlet.http.HttpServletRequest	getSession
06/20/08 07:53:18 272	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-After-3	Servlet_Around_Session	javax.servlet.http.HttpServletRequest	getSession
06/20/08 07:53:18 442	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-Before-4	Servlet_Around_Session	javax.servlet.http.HttpSession	setAttribute
06/20/08 07:53:18 442	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-After-4	Servlet_Around_Session	javax.servlet.http.HttpSession	setAttribute
06/20/08 07:53:18 442	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-After-2	Servlet_Around_Service	jsp_servlet__index	_jspService
06/20/08 07:53:18 552	513bb54e27d6cc3a:30c3eff2:11aa5c7b97a-7#2-00000000000000bb	turmel	TraceElapsedTimeAction-Before-5	Servlet_Around_Session	javax.servlet.http.HttpServletRequest	getSession

Example configuration

Consider a Servlet_Around_Service application-scoped diagnostic monitor that has a TraceElapsedTimeAction action attached to it. Without dye filtering, any request that is handled by Servlet_Around_Service will trigger a TraceElapsedTimeAction.

However, you could use dye filtering to trigger `TraceElapsedTimeAction` only for requests that originated from user `admin@avitek.com` at IP address `127.0.0.1`.

1. Configure the `DyeInjection` monitor so that `USER1=admin@avitek.com` and `ADDR1=127.0.0.1`, and enable the `DyeInjection` monitor. For instructions, see "Configure diagnostic monitors in a diagnostic system module" in the *Oracle WebLogic Server Administration Console Online Help*.
2. Configure a *dye mask* and enable dye filtering for the `Servlet_Before_Service` diagnostic monitor. In the Administration Console:
 - a. Add the `Servlet_Around_Service` monitor from the WLDf instrumentation library to your application as described in "Configure instrumentation for applications" in the *Oracle WebLogic Server Administration Console Online Help*.
 - b. After adding the monitor, click **Save** on the **Settings for <application_name>** page.
 - c. Click the `Servlet_Around_Service` link to display the **Settings for Servlet_Around_Service** page.
 - d. Select the **Enabled** check box to enable the monitor.
 - e. Under **Actions**, move `TraceElapsedTimeAction` from the **Available** list to the **Chosen** list.
 - f. In the **Dye Mask** section, move `USER1` and `ADDR1` from the **Available** list to the **Chosen** list.
 - g. Select the **EnableDyeFiltering** check box.
 - h. Click **Save**.
3. Redeploy the application.

Configurations added via the Administration Console are *not* persisted to the `weblogic-diagnostics.xml` file in the application's META-INF directory or to the `DIAG_MODULE.xml` file; they are saved in the application's deployment plan.

You can also manually update your `DIAG_MODULE.xml` file to add diagnostic monitors, as shown in [Example 12–2](#), but this is not recommended. It is better to change the configuration via the Administration Console on a running server. Any changes you make to `DIAG_MODULE.xml` will not take effect until you redeploy the application.

Example 12–2 Sample Configuration for Using Dye Filtering in a Delegating Monitor, in `DIAG_MODULE.xml`

```
<wldf-resource>
  <name>MyDiagnosticModule</name>
  <instrumentation>
    <enabled>true</enabled>
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <enabled>true</enabled>
      <properties>ADDR1=127.0.0.1 USER1=admin@avitek.com</properties>
    </wldf-instrumentation-monitor>
    <wldf-instrumentation-monitor>
      <name>Servlet_Around_Service</name>
      <dye-mask>ADDR1 USER1</dye-mask>
      <dye-filtering-enabled>true</dye-filtering-enabled>
      <action>TraceElapsedTimeAction</action>
    </wldf-instrumentation-monitor>
  <!-- Other elements to configure instrumentation -->
```

```

</instrumentation>
<!-- Other elements to configure this diagnostic monitor -->
<wldf-resource>

```

With this configuration, the `TraceElapsedTimeAction` action will be triggered for the `Servlet_Around_Service` diagnostic monitor only for those requests that originate from IP address `127.0.0.1` and user `admin@avitek.com`.

The flags that are enabled in the diagnostic monitor must exactly match the bits set in the request's dye vector for an action to be triggered and an event to be written to the Event Archive. For example, if the diagnostic monitor has both the `USER1` and `ADDR1` flags enabled, and only the `USER1` flag is set in the request's dye vector, no action will be triggered and no event will be generated.

Note: When configuring a diagnostic monitor, do not enable multiple flags of the same type. For example, don't enable both the `USER1` and `USER2` flags, as the dye vector for a given request will never have both the `USER1` and `USER2` flags set.

12.5 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:

- Dye filtering for the delegating or custom diagnostic monitor is enabled in the application's `weblogic-diagnostics.xml` descriptor, or is enabled via the Administration Console.
- The request's dye vector contains all of the dyes that are defined in the monitor's dye mask. (The dye vector can also contain dyes that are not in the dye mask.)

12.5.1 Dye Filtering Example

[Figure 12-3](#) illustrates how dye filtering works, using a diagnostic module with three diagnostic monitors:

- The `DyeInjection` monitor is configured as follows:

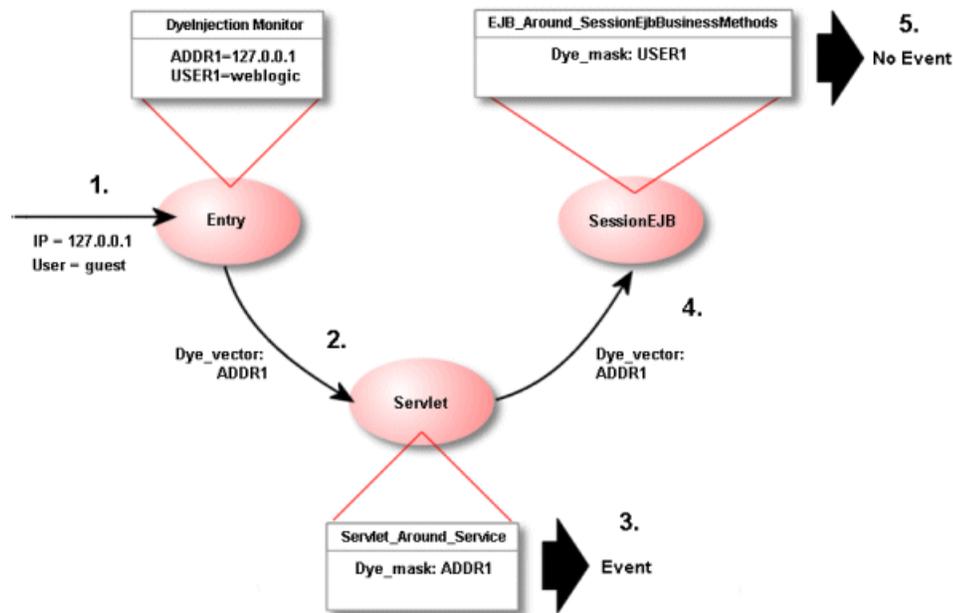
```

ADDR1=127.0.0.1
USER1=weblogic

```

- The `Servlet_Around_Service` monitor is configured with a dye mask containing only `ADDR1`.
- The `EJB_Around_SessionEjbBusinessMethods` monitor is configured with a dye mask containing `USER1` only.

Figure 12-3 Dye Filtering Example



1. A request initiated by user guest from IP address 127.0.0.1 enters the system. The user guest does not match the value for USER1 in the DyeInjection monitor, so the request is *not* dyed with the dye vector USER1. The originating IP address (127.0.0.1) matches the value for ADDR1 defined in the DyeInjection monitor, so the request *is* dyed with the dye vector ADDR1.
2. The request (dyed with ADDR1) enters the Servlet component, where the diagnostic monitor Servlet_Around_Service is woven into the code. (Servlet_Around_Service triggers diagnostic actions at the entry of and exit of certain servlet and JSP methods.) Dye monitoring is enabled for the monitor, and the dye mask is defined with the single value ADDR1.
3. When the request enters or exits a method instrumented with Servlet_Around_Service, the diagnostic monitor checks the request for dye vector ADDR1, which it finds. Therefore, the monitor triggers a diagnostic action, which generates a diagnostic event, for example, writing data to the Events Archive.
4. The request enters the SessionEJB component, where the diagnostic monitor EJB_Around_SessionEjbBusinessMethods is woven into the code. (EJB_Around_SessionEjbBusinessMethods triggers diagnostic actions at the entry and exit of all SessionBean methods). Dye monitoring is enabled for the monitor, and the dye mask is defined with the single value USER1.
5. When the request enters or exits a SessionBean method (instrumented with EJB_Around_SessionEjbBusinessMethods), the diagnostic monitor checks the request for dye vector USER1, which it does not find. Therefore, the monitor does not trigger a diagnostic action, and therefore does not generate a diagnostic event.

12.6 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.

Note: The `USER n` and `ADDR n` dyes allow inspection of requests from specific users or IP addresses. However, they do not provide a means to look at arbitrary user transactions. The `THROTTLE` dye provides that functionality by allowing sampling of requests.

12.6.1 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`.

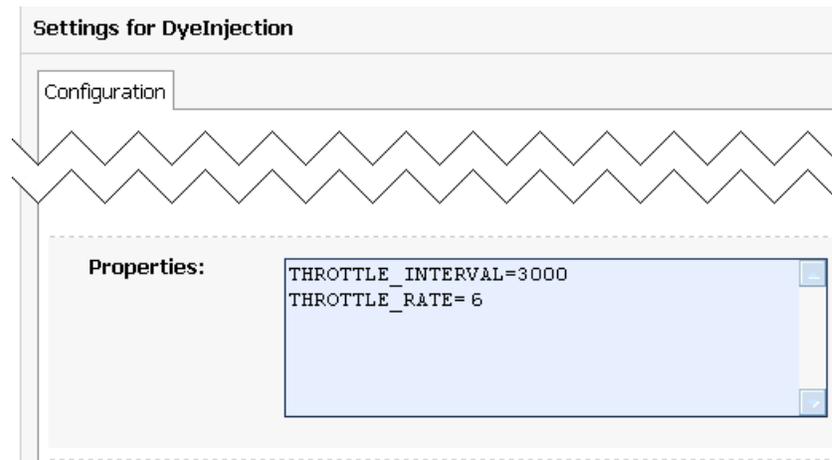
- `THROTTLE_RATE` sets the rate (in terms of the number of incoming requests) by which new incoming requests are dyed with the `THROTTLE` dye.

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`.

You can use `THROTTLE_INTERVAL` and `THROTTLE_RATE` together. If either condition is satisfied, the request is dyed with the `THROTTLE` dye.

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 12–4 Configuring the `THROTTLE` Dye



[Example 12–3](#) shows the resulting configuration in the descriptor file for the diagnostics module.

Example 12–3 Sample THROTTLE Configuration in the DyeInjection Monitor, in DIAG_MODULE.xml

```

<wldf-resource>
  <name>MyDiagnosticModule</name>
  <instrumentation>
    <wldf-instrumentation-monitor>
      <name>DyeInjection</name>
      <properties>
        THROTTLE_INTERVAL=3000
        THROTTLE_RATE=6
      </properties>
    </wldf-instrumentation-monitor>
  </instrumentation>
  <!-- Other elements to configure this diagnostic monitor -->
</wldf-resource>

```

Example 12–4 shows the configuration for a JDBC_Before_Start_Internal delegating monitor where the THROTTLE dye is set in the dye mask for the monitor.

Example 12–4 Sample Configuration for Setting THROTTLE in a Dye Mask of a Delegating Monitor, in DIAG_MODULE.xml

```

<wldf-resource>
  <name>MyDiagnosticModule</name>
  <instrumentation>
    <wldf-instrumentation-monitor>
      <name>JDBC_Before_Start_Internal</name>
      <enabled>true</enabled>
      <dye-mask>THROTTLE</dye-mask>
    </wldf-instrumentation-monitor>
  </instrumentation>
  <!-- Other elements to configure this diagnostic monitor -->
</wldf-resource>

```

12.6.2 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 [Section 12.3, "Configuring the Dye Vector via the DyeInjection Monitor."](#) One of those dyes can be the THROTTLE dye, so that you can filter on THROTTLE, just like any other dye.

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.

12.7 Using `weblogic.diagnostics.context`

The `weblogic.diagnostics.context` package provides applications with 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.)
- 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 than the eight flags reserved for applications.
- Prevent another application from 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.

Note: The diagnostic context payload can be viewed by other code in the same execution context; it can flow out of the process along with the `Work` instance; and it can be overwritten by other code running in the same execution context. Therefore, you should ensure the following behavior in your applications:

- Avoid including any sensitive data in the payload that, for example, could be returned by the `getPayload()` method.
 - Do not create a dependency on any particular data being available in the context payload. For example, applications should not rely on a particular context ID being present. If an application uses the contents of the payload, the application should first verify that the contents match what is expected.
-
-

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.

[Example 12–5](#) 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.

Example 12–5 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));
    }
}
```

Accessing Diagnostic Data With the Data Accessor

This chapter describes the Data Accessor component of the WebLogic Diagnostics Framework (WLDF) that is used to access 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. This chapter also describes how to use the Data Accessor online (when a server is running) and offline (when a server is not running).

This chapter includes the following sections:

- [Data Stores Accessed by the Data Accessor](#)
- [Accessing Diagnostic Data Online](#)
- [Accessing Diagnostic Data Offline](#)
- [Accessing Diagnostic Data Programmatically](#)
- [Resetting the System Clock Can Affect How Data Is Archived and Retrieved](#)

13.1 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 for Oracle WebLogic Server*.

13.2 Accessing Diagnostic Data Online

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

13.2.1 Accessing Data Using the Administration Console

You do not 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 *Oracle WebLogic Server Administration Console Online Help*.

13.2.2 Accessing Data Programmatically Using Run-Time MBeans

The Data Accessor provides the following run-time 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 `WLDFFDataAccessRuntimeMBean` 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 *Oracle WebLogic Server MBean Reference*.

- Use the `WLDFFDataAccessRuntimeMBean` 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 metadata about the columns of the data set and the earliest and latest timestamp of the records in the data store.

Data Accessor run-time 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 `WLDFDataAccessRuntime`'s 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");
rmbms.getAttribute(objName, "WLDFDataAccessRuntimes");
```

See "WLDFDataAccessRuntimeMBean" in the *Oracle WebLogic Server MBean Reference*.

13.2.3 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 "Diagnostics Commands" in the *WebLogic Scripting Tool Command Reference*.

13.2.4 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."](#)

13.3 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 *WebLogic Scripting Tool Command Reference*.

Note: 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.

13.4 Accessing Diagnostic Data Programmatically

[Example 13-1](#) shows the source Java code for a utility that uses the Accessor to query the different archive data stores.

Example 13-1 Sample Code to Use the WLDF Accessor

```
/*
 * WLAccessor.java
 *
 * Demonstration utility that allows query of the different ARCV data stores
 * via the WLDF Accessor.
 *
 */

import javax.naming.Context;
import weblogic.jndi.Environment;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Properties;
```

```

import weblogic.management.ManagementException;
import weblogic.management.runtime.WLDFAccessRuntimeMBean;
import weblogic.management.runtime.WLDFDataAccessRuntimeMBean;
import weblogic.diagnostics.accessor.ColumnInfo;
import weblogic.diagnostics.accessor.DataRecord;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.management.MBeanServerConnection;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.management.ObjectName;
import weblogic.management.mbeanservers.runtime.RuntimeServiceMBean;
import weblogic.management.runtime.ServerRuntimeMBean;
import weblogic.management.jmx.MBeanServerInvocationHandler;
import weblogic.management.configuration.ServerMBean;

/**
 * Demonstration utility that allows query of the different ARCV data stores
 * via the WLDF Accessor. The class looks up the appropriate accessor and
 * executes the query given the specified query parameters.
 *
 * To see information about it's usage, compile this file and run
 *
 * java WLAccessor usage
 */
public class WLAccessor {

    /** Creates a new instance of WLAccessor */
    public WLAccessor(Properties p) {
        initialize(p);
    }

    /**
     * Retrieve the specified WLDFDataAccessRuntimeMBean instance for querying.
     */
    public WLDFDataAccessRuntimeMBean getAccessor(String accessorType)
        throws Throwable
    {
        // Get the runtime MBeanServerConnection
        MBeanServerConnection runtimeMBS = this.getRuntimeMBeanServerConnection();

        // Lookup the runtime service for the connected server
        ObjectName rtSvcObjName = new ObjectName(RuntimeServiceMBean.OBJECT_NAME);
        RuntimeServiceMBean rtService = null;

        rtService = (RuntimeServiceMBean)
            MBeanServerInvocationHandler.newProxyInstance(
                runtimeMBS, rtSvcObjName
            );

        // Walk the Runtime tree to the desired accessor instance.
        ServerRuntimeMBean srt = rtService.getServerRuntime();

        WLDFDataAccessRuntimeMBean ddar =
            srt.getWLDFRuntime().getWLDFAccessRuntime().
            lookupWLDFDataAccessRuntime(accessorType);
    }
}

```

```

        return ddar;
    }

    /**
     * Execute the query using the given parameters, and display the formatted
     * records.
     */
    public void queryEventData() throws Throwable
    {
        String logicalName = "EventsDataArchive";
        WLDFDataAccessRuntimeMBean accessor = getAccessor(accessorType);

        ColumnInfo[] colinfo = accessor.getColumns();
        inform("Query string: " + queryString);

        int recordsFound = 0;
        Iterator actualIt =
            accessor.retrieveDataRecords(beginTime, endTime, queryString);
        while (actualIt.hasNext()) {
            DataRecord rec = (DataRecord)actualIt.next();
            inform("Record[" + recordsFound + "]: {");
            Object[] values = rec.getValues();
            for (int colno=0; colno < values.length; colno++) {
                inform("[ " + colno + " ] "
                    + colinfo[colno].getColumnName() +
                    " (" + colinfo[colno].getColumnTypeName() + "): " +
                    values[colno]);
            }
            inform("}");
            inform("");
            recordsFound++;
        }
        inform("Found " + recordsFound + " results");
    }

    /**
     * Main method that implements the tool.
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        try {
            WLAccessor acsr = new WLAccessor(handleArgs(args));
            acsr.queryEventData();
        } catch (UsageException uex) {
            usage();
        } catch (Throwable t) {
            inform("Caught exception, " + t.getMessage(), t);
            inform("");
            usage();
        }
    }

    public static class UsageException extends Exception {}

    /**
     * Process the command line arguments, which are provided as name/value pairs.
     */
    public static Properties handleArgs(String[] args) throws Exception
    {

```

```

        Properties p = checkForDefaults();
        for (int i = 0; i < args.length; i++) {
            if (args[i].equalsIgnoreCase("usage"))
                throw new UsageException();

            String[] nvpair = new String[2];
            int token = args[i].indexOf('=');
            if (token < 0)
                throw new Exception("Invalid argument, " + args[i]);
            nvpair[0] = args[i].substring(0,token);
            nvpair[1] = args[i].substring(token+1);
            p.put(nvpair[0], nvpair[1]);
        }
        return p;
    }

    /**
     * Look for a default properties file
     */
    public static Properties checkForDefaults() throws IOException {
        Properties defaults = new Properties();
        try {
            File defaultprops = new File("accessor-defaults.properties");
            FileInputStream defaultsIS = new FileInputStream(defaultprops);
            //inform("loading options from accessor-defaults.properties");
            defaults.load(defaultsIS);
        } catch (FileNotFoundException fnfex) {
            //inform("No accessor-defaults.properties found");
        }
        return defaults;
    }

    public static void inform(String s) {
        System.out.println(s);
    }

    public static void inform(String s, Throwable t) {
        System.out.println(s);
        t.printStackTrace();
    }

    private MBeanServerConnection getRuntimeMBeanServerConnection()
        throws IOException
    {
        // construct jmx service url

        // "service:jmx:[url]/jndi/[mbeanserver-jndi-name]"
        JMXServiceURL serviceURL =
            new JMXServiceURL(
                "service:jmx:" + getServerUrl() +
                "/jndi/" + RuntimeServiceMBean.MBEANSERVER_JNDI_NAME
            );

        // specify the user and pwd. Also specify weblogic provide package
        inform("user name [" + username + "]);
        inform("password [" + password + "]);
        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");
        // get jmx connector
    }

```

```

JMXConnector connector = JMXConnectorFactory.connect(serviceURL, h);

inform("Using JMX Connector to connect to " + serviceURL);
return connector.getMBeanServerConnection();
}

private void initialize(Properties p) {
    serverUrl = p.getProperty("url", "t3://localhost:7001");
    username = p.getProperty("user", "weblogic");
    password = p.getProperty("pass", "weblogic");
    queryString = p.getProperty("query", "SEVERITY IN
('Error', 'Warning', 'Critical', 'Emergency')");
    accessorType = p.getProperty("type", "ServerLog");

    try {
        beginTime = Long.parseLong(p.getProperty("begin", "0"));

        String end = p.getProperty("end");
        endTime = (end==null) ? Long.MAX_VALUE : Long.parseLong(end);
    } catch (NumberFormatException nfex) {
        throw new RuntimeException("Error formatting time bounds", nfex);
    }
}

private static void usage() {
    inform("");
    inform("");
    inform("Usage: ");
    inform("");
    inform(" java WLAcessor [options]");
    inform("");
    inform("where [options] can be any combination of the following: ");
    inform("");
    inform("    usage                Prints this text and exits");
    inform("    url=<url>            default: 't3://localhost:7001'");
    inform("    user=<username>      default: 'weblogic'");
    inform("    pass=<password>      default: 'weblogic'");
    inform("    begin=<begin-timestamp> default: 0");
    inform("    end=<end-timestamp>  default: Long.MAX_VALUE");
    inform("    query=<query-string> default: \"SEVERITY IN
('Error', 'Warning', 'Critical', 'Emergency')\"");
    inform("    type=<accessor-type>  default: 'ServerLog'");
    inform("");
    inform("Example:");
    inform("");
    inform("    java WLAcessor user=system pass=gumbyl234 url=http://myhost:8000
\\");
    inform("        query=\"SEVERITY = 'Error'\" begin=1088011734496
type=ServerLog");
    inform("");
    inform("");
    inform("");
    inform("All properties (except \"usage\") can all be specified in a file ");
    inform("in the current working directory. The file must be named: ");
    inform("");
    inform("        \"accessor-defaults.properties\"");
    inform("");
    inform("Each property specified in the defaults file can still be ");
    inform("overridden on the command-line as shown above");
    inform("");
}

```

```
    }

    /** Getter for property serverUrl.
     * @return Value of property serverUrl.
     *
     */
    public java.lang.String getServerUrl() {
        return serverUrl;
    }

    /** Setter for property serverUrl.
     * @param serverUrl New value of property serverUrl.
     *
     */
    public void setServerUrl(java.lang.String serverUrl) {
        this.serverUrl = serverUrl;
    }

    protected String serverName = null;
    protected String username = null;
    protected String password = null;
    protected String queryString = "";
    private String serverUrl = "t3://localhost:7001";
    private String accessorType = null;

    private long endTime = Long.MAX_VALUE;
    private long beginTime = 0;

    private WLDFAccessRuntimeMBean dar = null;
}

```

13.5 Resetting the System Clock Can Affect How Data Is Archived and Retrieved

Resetting the system clock to an earlier time while diagnostic data is being written to the WLDF Archive or logs can cause unexpected results when you query that data based on a timestamp. For example, consider the following sequence of events:

1. At 2:00 p.m., a diagnostic event is archived as RECORD_200, with a timestamp of 2:00:00 PM.
2. At 2:30 p.m., a diagnostic event is archived as RECORD_230, with a timestamp of 2:30:00 PM.
3. At 3:00 p.m., the system clock is reset to 2:00 p.m.
4. At 2:15 p.m. (after the clock was reset), a diagnostic event is archived as RECORD_215, with a timestamp of 2:15:00 PM.
5. You issue a query to retrieve records generated between 2:00 and 2:20 p.m.

The query will not retrieve RECORD_215, because the 2:30:00 PM timestamp of RECORD_230 ends the query.

Deploying WLDF Application Modules

This chapter describes how to deploy WebLogic Diagnostics Framework (WLDF) application modules. The only WLDF component you can use with applications is Instrumentation (see [Section 11.5, "Configuring Application-Scoped Instrumentation"](#)).

You configure and manage instrumentation for an application as a diagnostics application module, which is an application-scoped resource. The configuration is persisted in a descriptor file which you deploy with the application. A diagnostic module deployed in this way is available only to the enclosing application. Using application-scoped resources ensures that an application always has access to required resources and simplifies the process of deploying the application to new environments.

You can deploy an application using a deployment plan, which permits dynamic configuration updates.

Note: For instrumentation to be available for an application, instrumentation must be enabled on the server to which the application is deployed. (Server-scoped instrumentation is enabled and disabled in the <instrumentation> element of the diagnostics descriptor for the server.)

This chapter includes the following sections:

- [Deploying a Diagnostic Module as an Application-Scoped Resource](#)
- [Using Deployment Plans to Dynamically Control Instrumentation Configuration](#)
- [Using a Deployment Plan: Overview](#)
- [Creating a Deployment Plan Using `weblogic.PlanGenerator`](#)
- [Sample Deployment Plan for Diagnostics](#)
- [Enabling Java HotSwap](#)
- [Deploying an Application with a Deployment Plan](#)
- [Updating an Application with a Modified Plan](#)

14.1 Deploying a Diagnostic Module as an Application-Scoped Resource

To deploy a diagnostic module as an application-scoped resource, you configure the module in a descriptor file named `weblogic-diagnostics.xml`. You then package the descriptor file with the application archive in the `ARCHIVE_PATH/META-INF` directory for the deployed application. For example:

```
D:\bea\wlserver_
12.1\samples\server\medrec\dist\standalone\exploded\medrec\META-INF\weblogic-diagnostics.xml
```

You can deploy the diagnostic module in both exploded and unexploded archives.

Note: If the EAR archive contains WAR, RAR or EJB modules that have the weblogic-diagnostics.xml descriptors in their META-INF directory, those descriptors are 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).

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

Because of the different ways that diagnostic application modules and diagnostic system modules are deployed, there are some differences in how you can reconfigure them and when those changes take place, as shown in [Table 14–1](#). The details of how to work with diagnostic application modules is described throughout this section. See [Chapter 11, "Configuring Instrumentation,"](#) for information about working with diagnostic system modules.

Table 14–1 Comparing System and Application Modules

Monitor 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	Yes, when HotSwap ¹ is enabled No, when HotSwap is not enabled: module must be redeployed	Yes	No	Yes	Yes - via plan

¹ See [Section 14.2, "Using Deployment Plans to Dynamically Control Instrumentation Configuration,"](#) for information about HotSwap.

14.2 Using Deployment Plans to Dynamically Control Instrumentation Configuration

WebLogic Server supports deployment plans, as specified in the Java EE Deployment Specification API (JSR-88). With deployment plans, you can modify an application's configuration after the application is built, without having to modify the application archives. For complete documentation on using deployment plans in WebLogic Server, see "Configuring Applications for Production Deployment" in *Deploying Applications to Oracle WebLogic Server*.

If you want to reconfigure an application that was deployed without a deployment plan, you must undeploy, unarchive, reconfigure, re-archive, and then redeploy the application. With a configuration plan, you can dynamically change many

configuration options simply by updating the plan, without modifying the application archive.

If you enable a feature called Java HotSwap (see [Section 14.6, "Enabling Java HotSwap"](#)) before deploying your application with a deployment plan, you can dynamically update all instrumentation settings without redeploying the application. If you do not enable HotSwap, or if you do not use a deployment plan, changes to some instrumentation settings require redeployment, as shown in [Table 14-2](#).

Table 14-2 When Application Instrumentation Configuration Changes Take Effect

Scenario / Settings to Use =>	Add and remove monitors	Attach and detach actions	Enable and disable monitors
Application deployed with a deployment plan, HostSwap enabled	Dynamic	Dynamic	Dynamic
Application deployed with a deployment plan, HotSwap not enabled	Must redeploy application ¹	Dynamic	Dynamic
Application deployed without a deployment plan	Must redeploy application	Must redeploy application	Must redeploy application

¹ If HotSwap is not enabled, you can "remove" a monitor, but that just disables it. The instrumentation code is still woven into the application code. You cannot re-enable it through a modified plan.

You can use a deployment plan to dynamically update configuration elements without redeploying the application.

- <enabled>
- <dye-filtering-enabled>
- <dye-mask>
- <action>

14.3 Using a Deployment Plan: Overview

The general process for creating and using a deployment plan is as follows:

1. Create a well-formed `weblogic-diagnostics.xml` descriptor file for the application.

Oracle recommends that you create an empty descriptor. This provides full flexibility for dynamically modifying the configuration. It is possible to create monitors in the original descriptor file and then use a deployment plan to override the settings. However, you will be unable to completely remove monitors without redeploying. If you add monitors using a deployment plan to an empty descriptor, all such monitors can be removed. For information about configuring diagnostic application modules, see [Section 11.5, "Configuring Application-Scoped Instrumentation."](#)

The schema for `weblogic-diagnostics.xml` is available at <http://xmlns.oracle.com/weblogic/weblogic-diagnostics/1.0/weblogic-diagnostics.xsd>.

2. Place the descriptor file `weblogic-diagnostics.xml`, in the top-level META-INF directory of the appropriate archive.
3. Create a deployment plan, for example by using `weblogic.PlanGenerator`. See [Section 14.4, "Creating a Deployment Plan Using `weblogic.PlanGenerator`."](#)

4. Start the server, optionally enabling Java HostSwap. See [Section 14.6, "Enabling Java HotSwap."](#)
5. Deploy the application using the deployment plan. See [Section 14.7, "Deploying an Application with a Deployment Plan"](#)).
6. When needed, edit the plan and update the application with the plan. See [Section 14.8, "Updating an Application with a Modified Plan."](#)

14.4 Creating a Deployment Plan 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.

The PlanGenerator tool inspects all Java EE 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.

To create the plan, use the following syntax:

```
java weblogic.PlanGenerator -plan output-plan.xml [options]
    application-path
```

For example:

```
java weblogic.PlanGenerator -plan foo.plan -dynamics /test/apps/mywar
```

Note: The -dynamics options specifies that the plan should be generated to include only those options that can be dynamically updated.

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

For more information about using PlanGenerator, see weblogic.PlanGenerator Command Line Reference" and "Exporting an Application for Deployment to New Environments" in *Deploying Applications to Oracle WebLogic Server*

14.5 Sample Deployment Plan for Diagnostics

[Example 14-1](#) shows a simple deployment plan generated using weblogic.PlanGenerator. (For readability, some information has been removed.) The plan enables the Servlet_Before_Service monitor and attaches to it the actions DisplayArgumentsAction and StackDumpAction.

Example 14-1 Sample Deployment Plan

```
<?xml version='1.0' encoding='UTF-8'?>
<deployment-plan xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
global-variables="false">
  <application-name>jsp_expr_root</application-name>

  <variable-definition>
    <!-- Add two additional actions to Servlet_Before_Service monitor -->
```

```

<variable>
  <name>WLDFInstrumentationMonitor_Servlet_Before_Service_Actions_113050559713922</name>
  <value>"DisplayArgumentsAction", "StackDumpAction"</value>
</variable>
<!-- Enable the Servlet_Before_Service monitor -->
<variable>
  <name>WLDFInstrumentationMonitor_Servlet_Before_Service_Enabled_113050559713927</name>
  <value>true</value>
</variable>
</variable-definition>

<module-override>
  <module-name>jspExpressionWar</module-name>
  <module-type>war</module-type>
  <module-descriptor external="false">
    <root-element>weblogic-web-app</root-element>
    <uri>WEB-INF/weblogic.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>web-app</root-element>
    <uri>WEB-INF/web.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>wldf-resource</root-element>
    <uri>META-INF/weblogic-diagnostics.xml</uri>
    <variable-assignment>
      <name>WLDFInstrumentationMonitor_Servlet_Before_Service_Actions_113050559713922</name>
      <xpath>/wldf-resource/instrumentation/wldf-instrumentation-monitor/[name="Servlet_Before_
Service"]/action</xpath>
    </variable-assignment>
    <variable-assignment>
      <name>WLDFInstrumentationMonitor_Servlet_Before_Service_Enabled_113050559713927</name>
      <xpath>/wldf-resource/instrumentation/wldf-instrumentation-monitor/[name="Servlet_Before_
Service"]/enabled</xpath>
    </variable-assignment>
  </module-descriptor>
</module-override>
<config-root xsi:nil="true"></config-root>
</deployment-plan>

```

For a list and documentation of diagnostic monitors and actions that you can specify in the deployment plan, see [Appendix B, "WLDF Instrumentation Library."](#)

14.6 Enabling Java HotSwap

To enable Java HotSwap, start the server with the following command line switch:

```
-javaagent:$WL_HOME/server/lib/diagnostics-agent.jar
```

14.7 Deploying an Application with a Deployment Plan

To take advantage of the dynamic control provided by a deployment plan, you must deploy the application with the plan.

You can use any of the standard WebLogic Server tools for controlling deployment, including the Administration 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. 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 or after the deployment process without having to modify the application archive.

14.8 Updating an Application with a Modified Plan

You change configuration settings by modifying the deployment plan and then updating or redeploying the application, depending on whether or not HotSwap is enabled. (See [Table 14–2](#) to see when you can simply update the application and when you must redeploy it.) You can use any of the standard WebLogic Server tools for updating or redeploying, including the Administration Console or the WebLogic Scripting Tool (WLST).

If you enabled HotSwap, you can update the configuration for the application with the modified plan values by *updating* the application with the plan. For example, the following WLST command updates an application with a plan:

```
wls:/mydomain/serverConfig> updateApplication('BigApp',  
    'c:/myapps/BigApp/newPlan/plan.xml', stageMode='STAGE',  
    testMode='false')
```

If you did not enable HotSwap, you must *redeploy* the application for certain changes to take effect. For example, the following WLST command redeploys an application using a plan:

```
wls:/mydomain/serverConfig> redeploy('myApp' 'c:/myapps/plan.xml')
```

Using the Monitoring Dashboard

This chapter describes the Monitoring Dashboard, which provides views and tools for graphically presenting diagnostic data about servers and applications running on them. The underlying functionality for generating, retrieving, and persisting diagnostic data is provided by the WebLogic Diagnostics Framework (WLDF). The Monitoring Dashboard provides additional tools for presenting that data in charts and graphs.

This chapter includes the following sections:

- [Running the Monitoring Dashboard](#)
- [Scope of the Diagnostic Information Displayed](#)
- [About the Monitoring Dashboard Interface](#)
- [Understanding How Metrics Are Collected and Presented](#)
- [The Parts of a Chart](#)

15.1 Running the Monitoring Dashboard

You can launch the Monitoring Dashboard from the WebLogic Server Administration Console, or you can run it separately in a web browser. The Monitoring Dashboard is always displayed in its own tab, or window, depending on the preferences you have set for your browser. You do not need to be logged in to the Administration Console to use the Monitoring Dashboard; but if you are not logged in, you are prompted for your username and password credentials.

For more information, see "Launch the Monitoring Dashboard" in *Oracle WebLogic Server Administration Console Online Help*.

15.2 Scope of the Diagnostic Information Displayed

The diagnostic data displayed by the Monitoring Dashboard consists of run-time MBean attributes with numeric or Boolean values that are useful to measure, either as their current values or as their changes over time. These values, referred to in the Monitoring Dashboard as *metrics*, originate from one or more run-time MBean instances from one or more servers in the domain.

The Monitoring Dashboard obtains metrics from two sources:

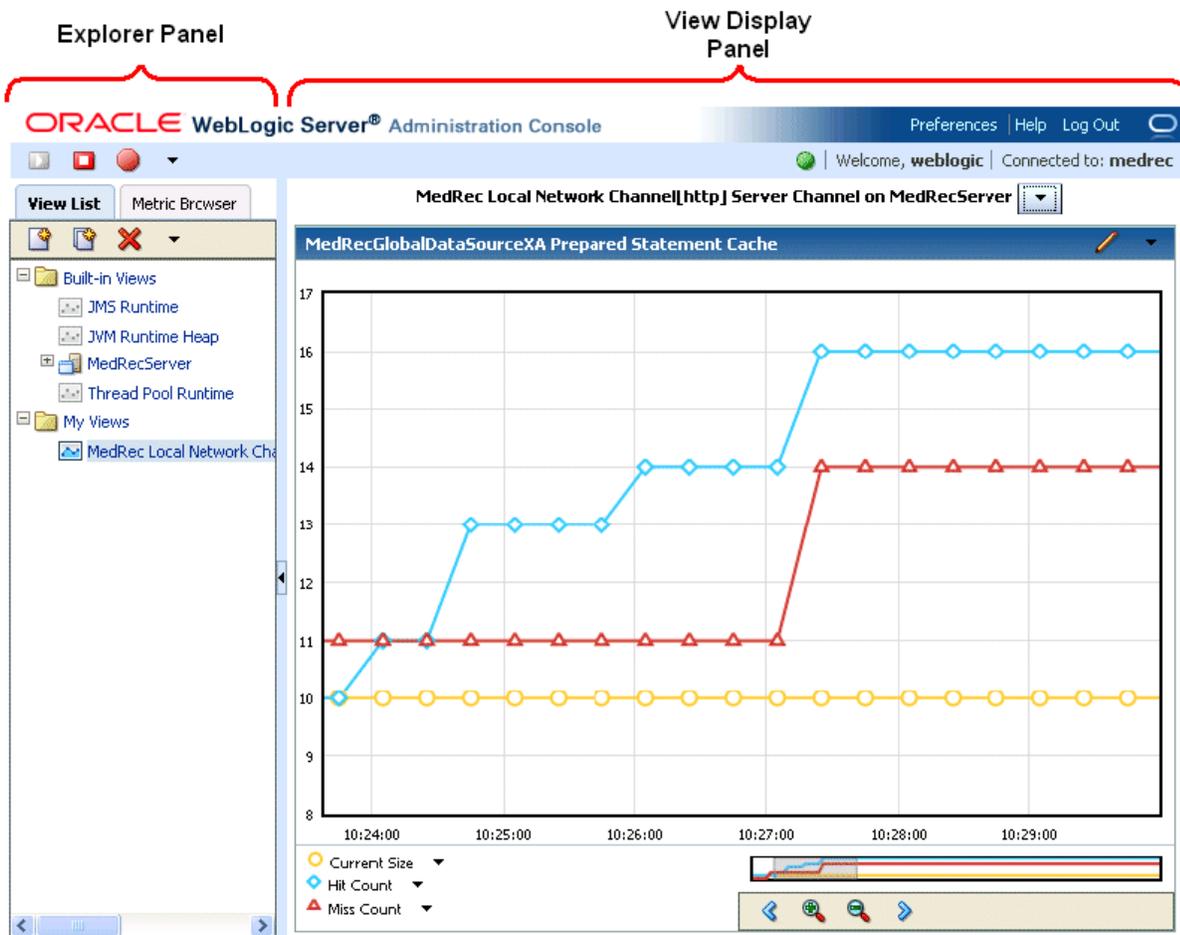
- Directly from active run-time MBean instances — these metrics are sometimes called *polled metrics* in this chapter.
- From the Archive that have been collected by the Harvester — these metrics are also known as *collected metrics* to distinguish them from metrics whose values are

obtained directly from active run-time MBean instances and returned to the Monitoring Dashboard.

15.3 About the Monitoring Dashboard Interface

The Monitoring Dashboard has two main panels: the explorer panel and the view display panel, as shown in the following figure.

Figure 15–1 Monitoring Dashboard Panels

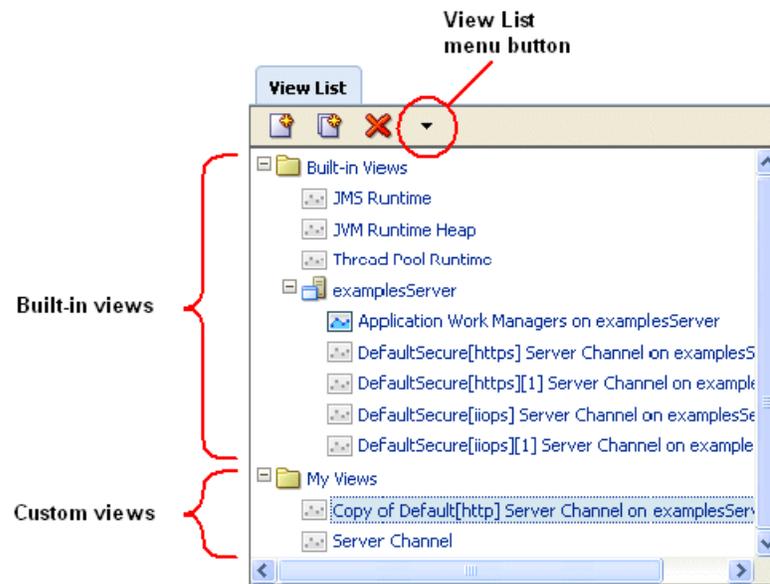


The explorer panel provides access to the following:

- View List — Set of existing built-in and custom views. It also contains controls for creating, copying, renaming, and deleting views. For details, see [Section 15.3.1, "View List."](#)
- Metric Browser — Provides a means to navigate to and select the specific MBean instance attributes whose metric values you want to display in a chart in a view. For details, see [Section 15.3.2, "Metric Browser."](#)

15.3.1 View List

To display a view, select it from the View List, shown in [Figure 15–2](#).

Figure 15–2 Built-in and Custom Views Displayed in the View List

Views are presented in two primary categories:

- Built-in views

The Built-in views are a set of predefined views of available run-time metrics for all running WebLogic Server instances in the domain. These views surface some of the more critical run-time WebLogic Server performance metrics and serve as examples of the Monitoring Dashboard's view and charting capabilities.

Note the following about built-in views:

- Built-in views are dynamic. For example, if four servers are running, the set of available built-in views and its charts are related to those four servers. If five servers are running, then the set of built-in views and its charts expands for each additional server. In addition, if the number of running server instances changes while you are using dashboard (for example, a server is started or stopped), and you want to see the new built-in views for the current set of running server instances, refresh the view list by selecting **Refresh** from the View List menu.
- Built-in views are automatically available with every WebLogic Server installation and can be used by every user logged into Administration Console or Monitoring Dashboard.
- You cannot modify a built-in view, but you can copy it. Once copied, the view can be modified, renamed, saved, and deleted.

- Custom views

A custom view is any view created by a user. Custom views are available only to the user who created them. Custom views are automatically persisted for the user and are in effect only for that user account and only in the current domain. (However, note that polled metric values that are displayed in custom views are *not* persisted if you close the Monitoring Dashboard window, just as they are not persisted for built-in views either.)

No custom views are available by default.

For more information, see the following topics in *Oracle WebLogic Server Administration Console Online Help*:

- "Work with views in the Monitoring Dashboard"
- "Start and stop views"
- "Create custom views"
- "Copy a view"
- "Delete a view"

15.3.2 Metric Browser

Charts display metrics, which are attributes of MBean instances. Metrics can be either of the following:

- Metrics whose values are obtained from active MBean instances in a running WebLogic Server instance.

The running server instances are polled at regular intervals, and the charts that display the metric values that are returned are continually updated (see [Section 15.4.1.1, "Current Time Range Charts"](#)).
- Collected metrics whose values are obtained from the Archive.

Collected metrics have been previously captured by the WLDF Harvester and placed in the Archive, and they provide a record of past state. Charts that display only collected metrics are not updated (see [Section 15.4.1.2, "Custom Time Range Charts"](#)).

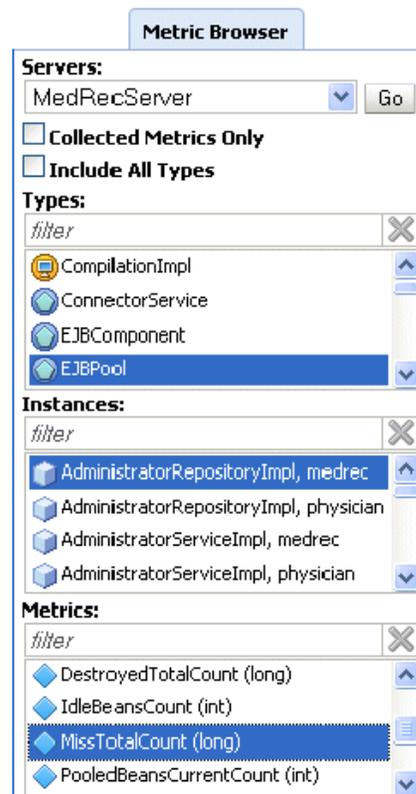
You use the Metric Browser to select the metrics that you want to add to a chart. The Metric Browser, shown in [Figure 15-3](#), displays:

- Currently registered WebLogic MBean types
- Currently registered instances of MBean types
- Attributes of the listed registered instances

As a convenience for selecting metrics that have been collected by the Harvester, the Metric Browser includes the **Collected Metrics Only** button. When you select this button, the Metric Browser displays only collected metrics.

To see metrics for all run-time MBean types regardless of whether instances of them are currently active, select **Include All Types**. To determine whether a metric was collected by the harvester, select the metric, or leave the mouse positioned over it. A note window is displayed that provides information about the metric, including whether or not it is a collected metric (that is, collected by the harvester).

Figure 15–3 Metric Browser



To use the Metric Browser, select the server instance containing the metric values you want to display. The Metric Browser can optionally constrain the list of MBean types, registered instances, and metrics that are displayed to only those for which metric data has been collected, or display all MBean types for the server even if they have no active instances.

In addition, you do not need to find a metric by first selecting its MBean type and then the instance in which it exists. You can select a metric in any order; for example, you can start by first selecting a metric, or by first selecting the MBean instance if you prefer. In addition, you can apply filters to each list to further constrain the items that are displayed.

You can select and filter in any order. Selecting an item in one list may make a selection in another and may also constrain other lists. Note the following behavior:

- Initially the Types list box shows all MBean types (as determined by the settings of the **Collected Metrics Only** and **Include All Types** checkboxes), the Instances list box shows all MBean instances, and the Metrics list box shows all metrics.
- Selecting a specific MBean type causes the MBean instances list to be constrained to instances of that type and the metrics list to be constrained to metrics of that type.
- Selecting **(none)** in the Types list specifies that no type is selected, which causes the entries in the Instances and Metrics lists to be unconstrained.
- Selecting a specific MBean instance, either before or after making any other selection, causes:
 - The corresponding MBean type in the Types list box to become selected.

- The entries in the Metrics list to become constrained to only those metrics for that MBean instance.
- Selecting a specific entry in the Metrics list box, either before or after making any other selection, causes:
 - The specific MBean type to which the metric corresponds to become selected in the Types list.
 - The Instances list to be constrained to the MBean instances to which the metric corresponds.
- When you enter a filter string into any of the list boxes, you constrain the list contents to include only the items that match the filter. The behaviors described in the preceding items that are used in combination with the filter result in a behavior similar to a "logical and."

The effect of these behaviors is to reinforce the relationships among MBean types, MBean instances, and metrics. Each MBean instance is of a specific MBean type, and each metric corresponds to a particular MBean type. The MBean type determines both all the instances of that type as well as all the metrics that the type has.

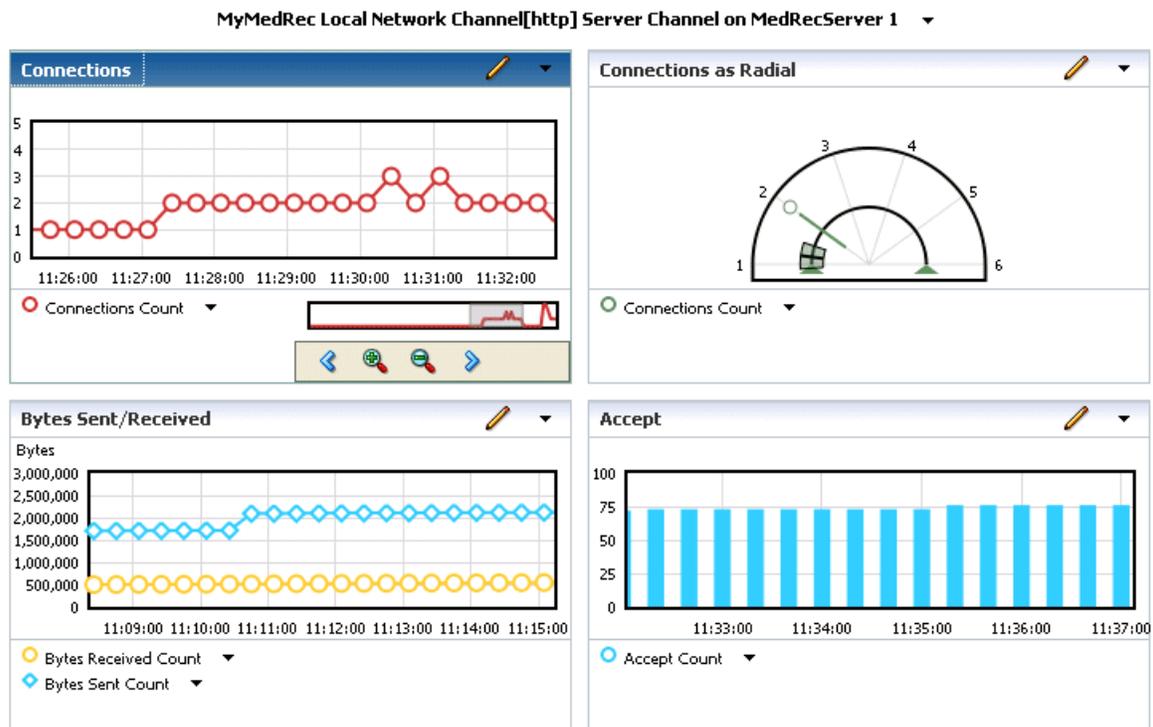
For information about using the Metric Browser, see the following topics in *Oracle WebLogic Server Administration Console Online Help*:

- "Work with the Metric Browser"
- "Select the server to monitor"
- "Display items in the Metric Browser"
- "Display summary notes about MBean instances and metrics in the Metric Browser"

15.3.3 View Display Panel

A view is a collection of one or more charts that display captured metric values, as shown in [Figure 15-4](#). Only one view is displayed at a time in the Monitoring Dashboard; however, multiple views can be running simultaneously.

Figure 15–4 View Containing Four Charts



Each chart in the view contains a legend, labels, and controls for identifying and displaying the data. The following chart styles can be included in a view:

- Time-series charts, such as a line plot or bar graph that show changes in each metric's value over a period of time
- Gauges, which show the current or most recent value of a metric along with the following statistics that have been collected for the metric's values:
 - Minimum
 - Maximum
 - Average
 - Standard deviation

Charts can show the metrics for a current time range, meaning that the chart is updated continually as the Monitoring Dashboard obtains new values for the metric at regular intervals. Or, for charts for which you specify a custom time range that has already passed, charts can display collected metrics obtained from the Archive that were captured by the Harvester.

For information about displaying and starting views, and arranging charts in them, see the following topics in *Oracle WebLogic Server Administration Console Online Help*:

- "Display or create views, charts, and metrics: main steps"
- "Work with views in the Monitoring Dashboard"
- "Display views"
- "Start and stop views"

For general details about Monitoring Dashboard charts, see [Section 15.5, "The Parts of a Chart."](#)

15.4 Understanding How Metrics Are Collected and Presented

As mentioned in [Section 15.2, "Scope of the Diagnostic Information Displayed,"](#) the Monitoring Dashboard displays metrics from two sources:

- Real-time, polled metric values that are obtained at regular intervals from running WebLogic Server instances and returned to the Monitoring Dashboard.
- Metrics collected by the Harvester and placed into the Archive.

To view real-time, polled metrics in the Monitoring Dashboard, it is not necessary to configure the Harvester. When a view is started with charts that contain one or more real-time, polled metrics, the run-time MBean instances corresponding to those metrics are polled at each configured interval, and the requested metric values are returned to the Monitoring Dashboard. A polled metric is stored only once in the Monitoring Dashboard, even if that metric has been added to multiple charts or multiple views. The run-time MBean instance corresponding to that metric is also polled only once at each interval, regardless of the number of charts or views in which its metric values are displayed. So when an updated value for a metric arrives in the Monitoring Dashboard, all charts containing that metric are updated simultaneously. This enables the Monitoring Dashboard to minimize the performance overhead on your system and maximize its overall efficiency.

To be able to view collected metrics, you must first configure the Harvester to collect the data you want to monitor and have it available in the Archive. In a view with one or more custom time range charts containing collected metrics, the values for those metrics that correspond to the specific custom time ranges are fetched once from the Archive and displayed in those charts. Note that collected metrics data is also available for programmatic access, and it is written to a standard log, `HarvestedDataArchive`, which you can view using the standard Administration Console as well as the Monitoring Dashboard. For information about configuring the Harvester to collect metrics, see [Chapter 7, "Configuring the Harvester for Metric Collection."](#)

15.4.1 About Metrics and Chart Types

The way in which the Monitoring Dashboard presents metrics depends upon the chart in which they are displayed. After you add a chart to a view, you can use the Chart Properties dialog box to specify either of the following time ranges:

- **Current**
- **Custom**

The following sections provide key information about how metrics are presented in each chart type.

15.4.1.1 Current Time Range Charts

This is the default time range for charts in the Monitoring Dashboard. Use this time range for displaying real-time, polled metrics, which can be displayed only in current time range charts. These charts are updated at regular intervals, which by default is every 20 seconds. (The sample interval can be customized in the Dashboard Preferences dialog box.)

When you add a metric to a current time range chart, the Monitoring Dashboard fetches a small number of historical values for that metric from the Archive, if they are available. Note the following about metric values obtained from the Archive for current time range charts:

- The number of values fetched is derived from the amount of time over which the stored samples can range, in which the sample interval is multiplied by the maximum samples for the chart. (The default sampling interval is 20 seconds and the default sample maximum is 100, which yields a time range of 2000 seconds, or approximately 33.3 minutes.)
- If the sampling interval used by the Harvester is different from the one configured for the Monitoring Dashboard, some distortion may be evident in the graphing of that metric.

15.4.1.2 Custom Time Range Charts

Charts configured with a custom time range display collected metrics only. When you specify a custom time range for a chart and add a collected metric, the Monitoring Dashboard fetches the metric's values from the Archive that match the specified time range. These charts are static: once the Monitoring Dashboard displays collected metrics in a custom time range chart, the values of those metrics are never updated.

Note the following:

- Custom time range charts never include real-time, polled metric values.
- As a convenience for creating custom time range charts, the Metric Browser includes a button labeled **Collected Metrics Only**. When you select this button, the Metric Browser displays only collected metrics.

15.4.2 Sequence in which Metrics Data is Displayed

If the Harvester is configured to collect run-time MBean metrics, collection can begin independently of whether the Monitoring Dashboard is running. This section shows the sequence of activity that occurs when the Monitoring Dashboard collects and displays metrics in current time range and custom time range charts.

1. If the Harvester is configured to collect data for a metric, it starts to harvest that data after the server is started. The data is persisted in the Archive.
2. When the Monitoring Dashboard is launched, the list of available built-in and custom views is displayed. However, the real-time polling of metric values directly by JMX does not begin until one or more views are started.
3. When a view containing a current time range chart is started:
 - The Monitoring Dashboard begins polling the run-time MBean instances corresponding to the metrics contained in the chart.
 - If the Harvester has collected data for this metric in the Archive, that data added to the chart immediately. The number of samples that the Monitoring Dashboard obtains from the Archive corresponds to the time range for the chart.
 - If the Harvester was not configured to harvest data for this metric, no historical data is retrieved from the Archive for the metric and therefore none is displayed.
4. When a view containing a custom time range chart is created, the Monitoring Dashboard fetches from the Archive the set of values for the metric that match the custom time range specified for that chart. Once the values are displayed in the chart, the chart is never updated. The view in which a custom time range chart has been added does not need to be started in order to have the values of its collected metrics displayed.

5. As polled data values for a metric arrive in the Monitoring Dashboard, the new values are added to the chart. The oldest values obtained from the Archive, if available, are purged.

The chart always displays the most current data. The maximum samples for a chart determines how many samples can be saved for metrics, in both current and custom time range charts. After a chart reaches its maximum samples threshold, the oldest metric values are removed as newest arrive.

15.4.3 Notes about Metric Data Retention

If you exit from the Monitoring Dashboard, either by closing the Monitoring Dashboard window or by logging out, the browser prompts you to confirm your choice because all metric values captured by the Monitoring Dashboard during the session will be lost. Exiting from the Monitoring Dashboard has no effect on collected metrics persisted in the Archive. However, note that the Archive may have a data retirement policy in effect that limits how long data is retained there. For more information, see [Section 6.4, "Retiring Data from the Archives."](#)

15.5 The Parts of a Chart

A chart consists of the following:

- Chart name
- Chart viewport, which shows the data values of one or more metrics that are displayed according to the chart type. The type can be a time-series chart that plots individual data points over a specified time span, or a gauge that shows the current or most recent value of a metric along with statistics indicating maximum, minimum, average, and standard deviation values.
- X- and Y-axes for plotting diagnostic data
 - For time-series charts, data point plots against a time-based X-axis. You can zoom in or out to see a larger or smaller time segment in the viewport.
 - The Y-axis has a range and, by default, the range is automatically set to include all the data points in the chart.
 - You can optionally configure minimum and maximum values for the Y-axis.
- A legend for each metric that includes the name of the metric and the colored marker symbol that is used for that metric in the chart viewport.

The metric legend includes a button that, when selected, provides access to operations that can be performed with the metric, such as:

- Changing the name that is displayed for the metric in the chart, as well as the shape and color used for the metric data points displayed in the chart viewport
 - Copying or moving the metric to another chart, moving the legend within the current chart, or deleting the metric from the chart
- Chart series overview

The chart series overview, which is available for time-series charts, indicates the portion of metrics data currently visible in the chart in relation to the whole set of data that has been collected for the corresponding metrics for the represented period of time. You can "drag-select" in either the viewport or the chart series overview to zoom in or out of the chart's data.

The display of the chart series overview can optionally be suppressed, which can be useful for reducing the number of UI artifacts that are displayed simultaneously in the Monitoring Dashboard and also improving performance on slower systems or browsers.

For information about customizing the display settings for the chart series overview, see "Set dashboard preferences" in *Oracle WebLogic Server Administration Console Online Help*.

- Buttons for panning the and zooming the data displayed on the chart's X-axis. These buttons are part of the chart series overview, so the display properties set for the chart series overview also apply to these buttons.
- Optional Y-axis units label
- Chart menu, available by selecting the chart menu button

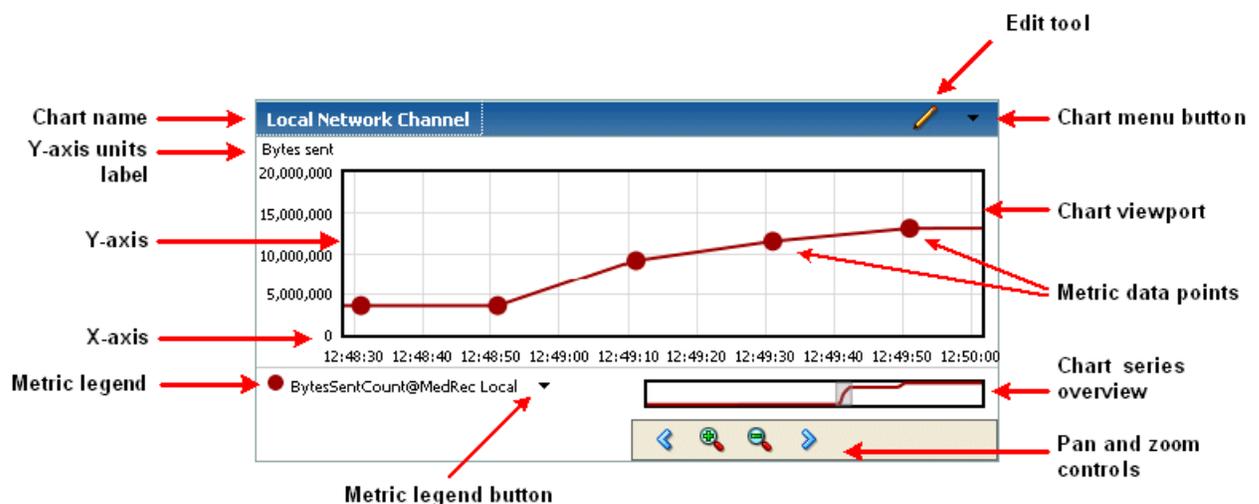
You can use the chart menu to add metrics, change the chart type, pan and zoom data shown in the viewport, and set various chart properties.

- Edit tool

Select the edit tool to modify the chart name, Y-axis units label, and names used to identify each metric added to the chart.

Figure 15-5 shows each of these parts as they appear in a line plot chart.

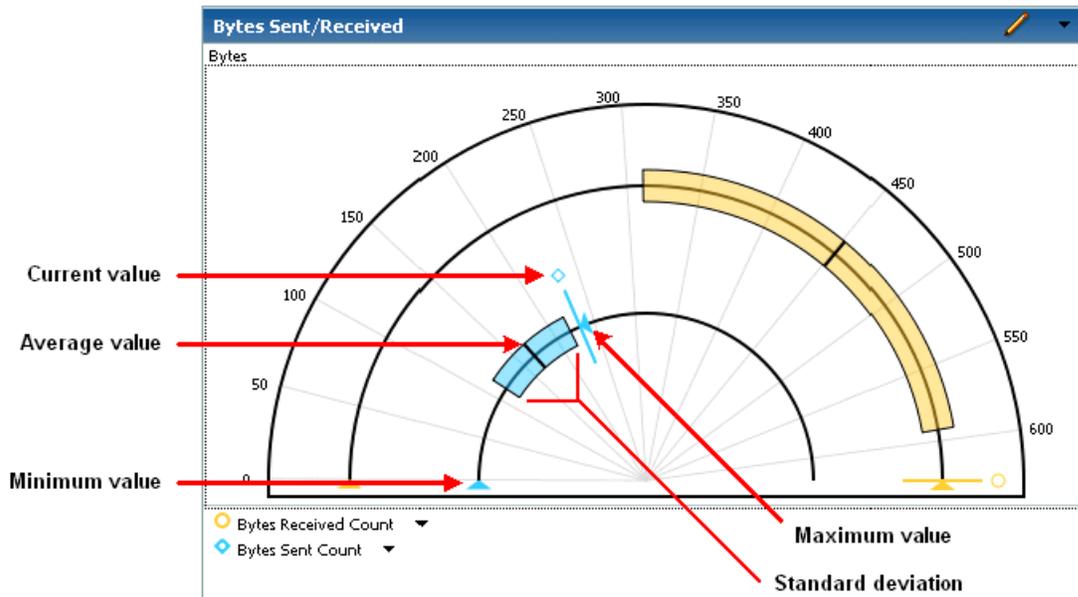
Figure 15-5 Parts of a Chart



A gauge chart, shown in Figure 15-6, contains the following additional information about each metric that has been added to it:

- Minimum and maximum values
- Average value
- Standard deviation

Figure 15–6 Data Values Shown in Gauge Chart Types



To display the numeric values indicated by each of these artifacts associated with a particular metric in a gauge chart, position the mouse pointer over that metric's marker symbol, in a gauge chart, in [Table 15–6](#) by the label **Current value**.

For information about how to create, modify, and work with charts in the Monitoring Dashboard, see the following topics in *Oracle WebLogic Server Administration Console Online Help*:

- "Work with metrics in charts"
- "Add charts to a view"
- "Choose the chart type"
- "Display summary information about metrics in charts"
- "Pan and zoom the metrics data shown in a chart"
- "Reset gauge statistics"
- "Copy or move charts"
- "Set chart time range"
- "Control the Y-axis range"
- "Display thresholds in charts"

Configuring and Using WLDF Programmatically

This chapter describes how to enable, configure, and monitor the WebLogic Diagnostics Framework (WLDF) programmatically, using the JMX API and the WebLogic Scripting Tool (WLST), as an alternative to performing these tasks using the WebLogic Server Administration Console

This chapter includes the following sections:

- [How WLDF Generates and Retrieves Data](#)
- [Mapping WLDF Components to Beans and Packages](#)
- [Programming Tools](#)
- [WLDF Packages](#)
- [Programming WLDF: Examples](#)

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

- *Developing Applications for Oracle WebLogic Server*
- *Developing Manageable Applications With JMX for Oracle WebLogic Server*
- *Developing Custom Management Utilities With JMX for Oracle WebLogic Server*
- *Deploying Applications to Oracle WebLogic Server*
- *Oracle WebLogic Scripting Tool*

16.1 How WLDF Generates and Retrieves Data

In general, diagnostic data is generated and retrieved by WLDF components following this process:

- 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.

Configuration is primarily an administrative task, accomplished either through the Administration 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 about configuring WLDF resources, see [Chapter 4, "Understanding WLDF Configuration."](#))

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

16.2 Mapping WLDF Components to Beans and Packages

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 WLST. Because weblogic.WLST is a JMX client; any task you can perform using WLST you can also perform programmatically through JMX.

[Table 16–1](#) lists the beans and packages associated with WLDF and its components. [Figure 16–1](#) groups the beans by type.

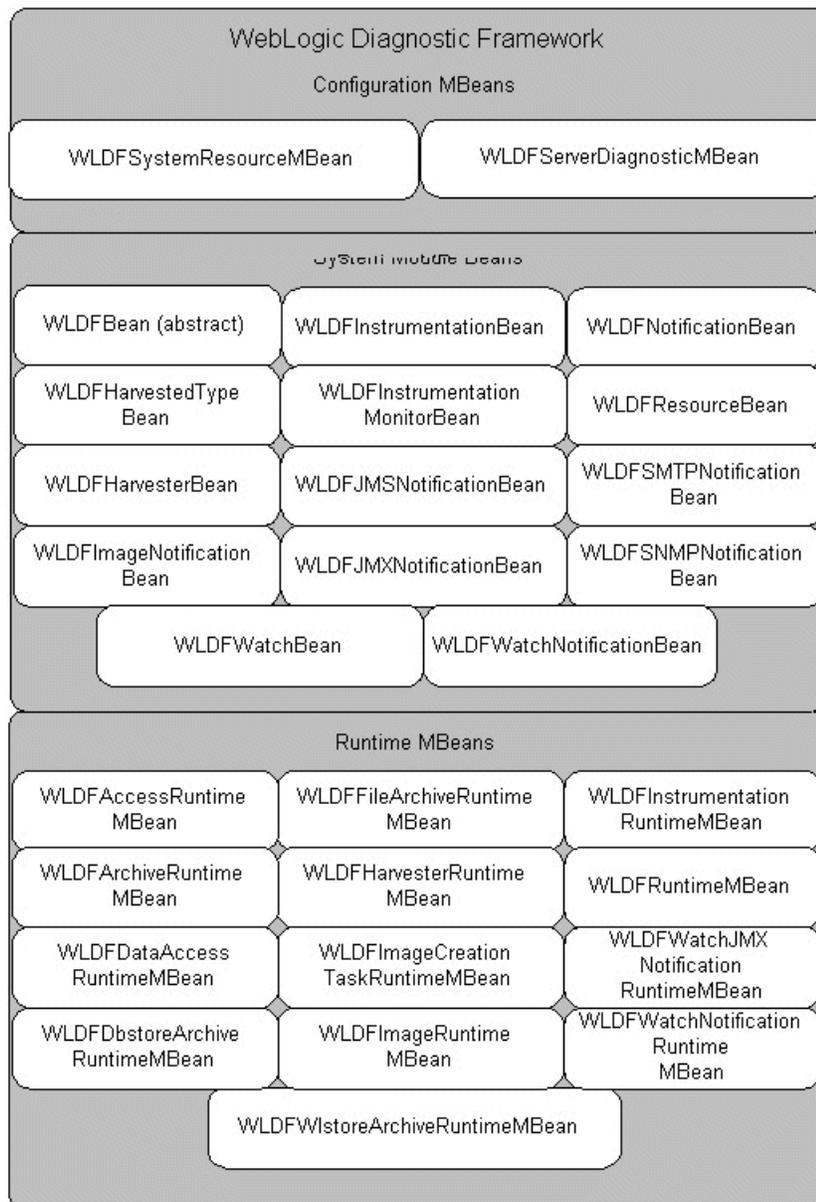
Table 16–1 Mapping WLDF Components to Beans and Packages

Component	Beans / Packages
WLDF	WLDFServerDiagnosticMBean WLDFSystemResourceMBean WLDFBean (abstract) WLDFResourceBean WLDFRuntimeMBean
Diagnostic Image	WLDImageNotificationBean WLDImageCreationTaskRuntimeMBean WLDImageRuntimeMBean
Instrumentation	WLDInstrumentationBean WLDInstrumentationMonitorBean WLDInstrumentationRuntimeMBean
Diagnostic Context	Package: weblogic.diagnostics.context DiagnosticContextHelper DiagnosticContextConstants
Harvester	WLDHarvesterBean WLDHarvestedTypeBean WLDHarvesterRuntimeMBean

Table 16–1 (Cont.) Mapping WLDF Components to Beans and Packages

Component	Beans / Packages
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 16–1 WLDF Configuration MBeans, Run-Time MBeans, and System Module Beans



16.3 Programming Tools

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

- Create and modify diagnostic descriptor files to configure the WLDF Harvester, Instrumentation, and Watch and Notification components at the server level.
- Use JMX to access WLDF operations and attributes.
- Use JMX to create custom MBeans that contain harvestable data. You can then configure the Harvester to collect that data and configure a watches and notifications to monitor the values.
- 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.)

16.3.1 Configuration and Run-Time APIs

The configuration and run-time APIs configure and monitor WLDF. Both the configuration and run-time APIs are exposed as MBeans.

- The configuration MBeans and system module Beans create and configure WLDF resources, and determine their run-time behavior.
- The run-time MBeans monitor the run-time 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.

16.3.1.1 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 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

16.3.1.2 Run-Time APIs

The run-time APIs define interfaces that are used to monitor the run-time state of the WLDf components. Instances of these APIs are instantiated on instances of individually managed servers. These APIs are defined as run-time MBeans, so JMX clients can easily access them.

The run-time APIs encapsulate all other run-time interfaces for the individual WLDf components. These APIs are included in the `weblogic.management.runtime` package.

You can use the run-time APIs to monitor the following WLDf components:

- Data Collectors—You can use the run-time 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 run-time 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 run-time APIs to monitor the Watches and Notifications and Archive components.
 - For the Watches and Notifications component, you can reset watch alarms 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 run-time APIs to retrieve the diagnostic data persisted in the different archives. The run-time 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).

16.4 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.
 - `DiagnosticContextHelper`, 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.

16.5 Programming WLDF: Examples

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

- [Example: DiagnosticContextExample.java](#)
- [Example: HarvesterMonitor.java](#)
- [Example: JMXAccessorExample.java](#)

In addition, see the WLST and JMX examples in [Appendix D, "WebLogic Scripting Tool Examples."](#)

16.5.1 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 about diagnostic contexts, see [Chapter 12, "Configuring the DyeInjection Monitor to Manage Diagnostic Contexts."](#))

To compile and run the program:

1. Copy the DiagnosticContextExample.java example ([Example 16–1](#)) 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
```

Example 16–1 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));
    }
}
```

```

    }
}

```

16.5.2 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. A description of notification listeners and the HarvesterMonitor.java code are provided in the following sections:

- [Notification Listeners](#)
- [HarvesterMonitor.java](#)

For information about the Harvester component, see [Chapter 7, "Configuring the Harvester for Metric Collection."](#)

16.5.2.1 Notification Listeners

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

Note: You can develop plug-ins that propagate events generated by the WebLogic Diagnostics 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 16–2](#) describes each notification listener type that is provided with WebLogic Server and the relevant configuration settings for each type.

Table 16–2 Notification Listener Types

Notification Medium	Description	Configuration Parameter Requirements
JMS	Propagated via JMS Message queues or topics.	Required: Destination JNDI name. Optional: Connection factory JNDI name (use the default JMS connection factory if not present).
JMX	Propagated via standard JMX notifications.	None required. 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.

16.5.2.2 HarvesterMonitor.java

To compile and run the HarvesterMonitor program:

1. Copy the HarvesterMonitor.java example (Example 16–2) 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 that are 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 Example 16–3 for an example of output from the HarvesterMonitor.

Example 16–2 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 =
        getObjectNames(accessorRuntimeMBeanName);
    harvRuntimeMBeanName = getCanonicalName(
        "com.bea:ServerRuntime=" + serverName +
        ",Name=WLDFHarvesterRuntime,Type=WLDFHarvesterRuntime" +
        ",WLDFRuntime=WLDFRuntime");
    harvRuntimeMBeanObjectName = getObjectNames(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();}
}
}
}
}

```

[Example 16-3](#) contains sample output from the HarvesterMonitor program:

Example 16-3 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=WLDFHarveste
rRuntime,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

```

16.5.3 Example: JMXAccessorExample.java

The following example program uses JMX to print log entries to standard out. All access is performed through JMX. (For information about the Accessor component, see [Chapter 13, "Accessing Diagnostic Data With the Data Accessor."](#))

To compile and run the program:

1. Copy the JMXAccessorExample.java example ([Example 16-4](#)) 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.

Example 16-4 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.RuntimeServiceMBean.OB
JECT_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
}
```

WLDF Query Language

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](#)
- [Supported Operators](#)
- [Operator Precedence](#)
- [Numeric Relational Operations Supported on String Column Types](#)
- [Supported Numeric Constants and String Literals](#)
- [Creating Watch Rule Expressions](#)
- [Creating Data Accessor Queries](#)
- [Creating Log Filter Expressions](#)
- [Building Complex Expressions](#)

A.1 Components of a Query Expression

A query expression may include:

- Operators. (See [Section A.2, "Supported Operators."](#))
- Literals. (See [Section A.5, "Supported Numeric Constants and String Literals."](#))
- Variables. The supported variables differ for each type of expression. (See [Section A.6, "About Variables in Expressions."](#))

The query language is case-sensitive.

A.2 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.

Table A-1 (Cont.) WLDL Query Language Operators

Operator	Operator Type	Supported Operand Types	Definition
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.
&	Bitwise binary	Numeric, Dye flag	<p>Performs the bitwise AND function on each parallel pair of bits in each operand. If <i>both</i> operand bits are 1, the & function sets the resulting bit to 1. Otherwise, the resulting bit is set to 0.</p> <p>Examples of both the & and the operators are:</p> <p>1010 & 0010 = 0010</p> <p>1010 0001 = 1011</p> <p>(1010 & (1100 1101)) = 1000</p>
	Bitwise binary	Numeric, Dye flag	<p>Performs the bitwise OR function on each parallel pair of bits in each operand. If <i>either</i> operand bit is 1, the function sets the resulting bit to 1. Otherwise, the resulting bit is set to 0.</p> <p>For examples, see the entry for the bitwise & operator, above.</p>
=	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	<p>Evaluates to true when a character string matches a specified pattern that can include wildcards.</p> <p>LIKE supports two wildcard characters:</p> <p>A percent sign (%) matches any string of zero or more characters</p> <p>A period (.) matches any single character</p>
MATCH ES	Match	String	Evaluates to true when a target string matches the regular expression pattern in the operand String.
IN	Search	String	<p>Evaluates to true when the value of a variable exists in a predefined set, for example:</p> <p>SUBSYSTEM IN ('A','B')</p>

A.3 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. &, |
4. =, !=, <, >, <=, >=, LIKE, MATCHES, IN
5. AND
6. OR

A.4 Numeric Relational Operations Supported on String Column Types

Numeric relational operations can be performed on String column types when they hold numeric values. For example, if STATUS is a String type, while performing relational operations with a numeric operand, the column value is treated as a numeric value. For instance, in the following comparisons:

STATUS = 100

STATUS != 100

STATUS < 100

STATUS <= 100

STATUS > 100

STATUS >= 100

the query evaluator attempts to convert the string value to appropriate numeric value before comparison. When the string value cannot be converted to a numeric value, the query fails.

A.5 Supported Numeric Constants and String Literals

Rules for numeric constants 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.

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.
- An underscore character (_) can be used as a wildcard to stand for any single character.
- 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.

A.6 About Variables in Expressions

Variables represent the dynamic portion of a query expression that is evaluated at run time. 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](#)
- [Creating Data Accessor Queries](#)
- [Creating Log Filter Expressions](#)

A.7 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](#)
- [Creating Instrumentation Event Watch Rule Expressions](#)
- [Creating Harvester Watch Rule Expressions](#)

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

- [Chapter 8, "Configuring Watches and Notifications"](#)
- [Chapter 9, "Configuring Watches"](#)

A.7.1 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
SUBSYSTEM	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')
```

A.7.2 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 monitor.	String
PAYLOAD	Payload of instrumentation event.	String
RECORDID	The number of the record in the log.	Long
RETVAL	Return value of joinpoint.	String
SCOPE	Name of instrumentation scope.	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
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')
```

A.7.3 Creating Harvester Watch Rule Expressions

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

Instance-based and type-based expressions can contain an optional *namespace* component, which is the namespace of the metric being watched. It can be set to either Server Runtime or DomainRuntime. If omitted, it defaults to ServerRuntime.

If included and set to DomainRuntime, you should limit the usage to monitoring only DomainRuntime-specific MBeans, such as the ServerLifecycleRuntimeMBean. Monitoring remote managed server MBeans through the DomainRuntime MBeanServer is possible, but is discouraged for performance reasons. It is a best practice to use the resident watcher in each managed server to monitor metrics related to that managed server instance.

You can also use wildcards in instance names in Harvester watch rule expressions, as well as specify complex attributes in Harvester watch rule expressions. See [Appendix C, "Using Wildcards in Expressions."](#)

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:

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

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

```
${com.bea:namespace//instance_name//attribute_name}
```

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

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

Note: The *domain_name* is not required for a WebLogic Server domain 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
```

A.8 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 [Section A.8.1, "Data Store Logical Names."](#)

- Optionally, the name(s) of one or more columns from which to retrieve data, as described in [Section A.8.2, "Data Store Column Names."](#)

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

A.8.1 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 logs with extended format, the number of columns are user-defined.
JMSMessageLog	The name of the JMS Server.	JMSMessageLog/MyJMSServer
ServerLog	None	ServerLog
WebAppLog	Web server name + Root servlet context name	WebAppLog/MyWebServer/MyRootServletContext

A.8.2 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

Table A-5 (Cont.) Column Names for Log Types

Log Type	Column Names
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, THREADNAME, TIMESTAMP, TXID, TYPE, USERID
HarvestedDataArchive	ATTRNAME, ATTRTYPE, ATTRVALUE, DOMAIN, NAME, RECORDID, SERVER, TIMESTAMP, TYPE
HTTPAccessLog	AUTHUSER, BYTECOUNT, HOST, RECORDID, REMOTEUSER, REQUEST, STATUS, TIMESTAMP
JDBCLog	Same as DomainLog
JMSMessageLog	CONTEXTID, DATE, DESTINATION, EVENT, JMSCORRELATIONID, JMSMESSAGEID, MESSAGE, MESSAGECONSUMER, NANOTIMESTAMP, RECORDID, SELECTOR, TIMESTAMP, TXID, USERID
ServerLog	Same as DomainLog
WebAppLog	Same as DomainLog

An example of a Data Accessor query is:

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

In this example, the Accessor retrieves all messages that include the string "Failed" from the Deployer subsystem.

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 13, "Accessing Diagnostic Data With the Data Accessor."](#)

A.9 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 you use 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 for Oracle WebLogic Server*.

A.10 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 (MSGID = '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.

WLDF Instrumentation Library

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

- [Diagnostic Monitor Library](#)
- [Diagnostic Action Library](#)

For information about using items from the Instrumentation Library, see [Chapter 11](#), "Configuring Instrumentation."

B.1 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. You use the latter 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 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 [Table B-1](#).

Table B-1 *Diagnostic Monitors for Use Within Server Scope*

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
Connector_Before_Inbound	Before	Stateless	At entry of methods handling inbound connections.
Connector_After_Inbound	Server	Stateless	At exit of methods handling inbound connections.
Connector_Around_Inbound	Around	Around	At entry and exit of methods handling inbound connections.
Connector_Before_Outbound	Before	Stateless	At entry of methods handling outbound connections.

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
Connector_After_Outbound	After	Stateless	At exit of methods handling outbound connections.
Connector_Around_Outbound	Around	Around	At entry and exit of methods handling outbound connections.
Connector_Before_Tx	Before	Stateless	Entry of transaction register, unregister, start, rollback and commit methods.
Connector_After_Tx	After	Stateless	At exit of transaction register, unregister, start, rollback and commit methods.
Connector_Around_Tx	Around	Around	At entry and exit of transaction register, unregister, start, rollback and commit methods.
Connector_Before_Work	Before	Stateless	At entry of methods related to scheduling, starting and executing connector work items.
Connector_After_Work	After	Stateless	At exit of methods related to scheduling, starting and executing connector work items.
Connector_Around_Work	Around	Around	At entry and exit of methods related to scheduling, starting and executing connector work items.
DyeInjection	Before	Built-in	At points where requests enter the server.
JDBC_Before_Commit_Internal	Before	Stateless	JDBC subsystem internal code
JDBC_After_Commit_Internal	After	Stateless	JDBC subsystem internal code
JDBC_Before_Connection_Internal	Before	Stateless	Before calls to methods: Driver.connect DataSource.getConnection
JDBC_After_Connection_Internal	Before	Stateless	JDBC subsystem internal code
JDBC_Before_Rollback_Internal	Before	Stateless	JDBC subsystem internal code
JDBC_After_Rollback_Internal	After	Stateless	JDBC subsystem internal code
JDBC_Before_Start_Internal	Before	Stateless	JDBC subsystem internal code
JDBC_After_Start_Internal	After	Stateless	JDBC subsystem internal code
JDBC_Before_Statement_Internal	Before	Stateless	JDBC subsystem internal code
JDBC_After_Statement_Internal	After	Stateless	JDBC subsystem internal code
JDBC_After_Reserve_Connection_Internal	After	Stateless	After a JDBC connection is reserved from the connection pool.
JDBC_After_Release_Connection_Internal	After	Stateless	After a JDBC connection is released back to the connection pool.

Table B-2 lists the diagnostic monitors that can be used within application scopes; that is, in deployed applications. The **Compatible Action Type** column identifies the diagnostic action type that is compatible with each monitor.

Table B-2 Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_After_EntityEjbBusiness Methods	After	Stateless	At exits of all EntityBean methods, which are not standard ejb methods.
EJB_Around_EntityEjbBusinessMethods	Around	Around	At entry and exits of all EntityBean methods that are not standard ejb methods.
EJB_After_EntityEjbMethods	After	Stateless	At exits of methods: EnitivityBean.setEntityContext EnitivityBean.unsetEntityContext EnitivityBean.ejbRemove EnitivityBean.ejbActivate EnitivityBean.ejbPassivate EnitivityBean.ejbLoad EnitivityBean.ejbStore
EJB_Around_EntityEjbMethods	Around	Around	At exits of methods: EnitivityBean.setEntityContext EnitivityBean.unsetEntityContext EnitivityBean.ejbRemove EnitivityBean.ejbActivate EnitivityBean.ejbPassivate EnitivityBean.ejbLoad EnitivityBean.ejbStore
EJB_After_EntityEjbSemantic Methods	After	Stateless	At exits of methods: EnitivityBean.set* EnitivityBean.get* EnitivityBean.ejbFind* EnitivityBean.ejbHome* EnitivityBean.ejbSelect* EnitivityBean.ejbCreate* EnitivityBean.ejbPostCreate*
EJB_Around_EntityEjbSemanticMethods	Around	Around	At entry and exits of methods: EnitivityBean.set* EnitivityBean.get* EnitivityBean.ejbFind* EnitivityBean.ejbHome* EnitivityBean.ejbSelect* EnitivityBean.ejbCreate* EnitivityBean.ejbPostCreate*

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_After_SessionEjbMethods	After	Stateless	At exits of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_Around_SessionEjbMethods	Around	Around	At entry and exits of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_After_SessionEjbBusinessMethods	After	Stateless	At exits of all SessionBean methods, which are not standard ejb methods.
EJB_Around_SessionEjbBusinessMethods	Around	Around	At entry and exits of all SessionBean methods, which are not standard ejb methods.
EJB_After_SessionEjbSemanticMethods	After	Stateless	At exits of methods: SessionBean.ejbCreateSessionBean.ejbPostCreate
EJB_Around_SessionEjbSemanticMethods	Around	Around	At entry and exits of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
EJB_Before_EntityEjbBusinessMethods	Before	Stateless	At entry of all EntityBean methods, which are not standard ejb methods.
EJB_Before_EntityEjbMethods	Before	Stateless	At entry of methods: EntityBean.setEntityContext EntityBean.unsetEntityContext EntityBean.ejbRemove EntityBean.ejbActivate EntityBean.ejbPassivate EntityBean.ejbLoad EntityBean.ejbStore
EJB_Before_EntityEjbSemanticMethods	Before	Stateless	At entry of methods: EntityBean.set* EntityBean.get* EntityBean.ejbFind* EntityBean.ejbHome* EntityBean.ejbSelect* EntityBean.ejbCreate* EntityBean.ejbPostCreate*
EJB_Before_SessionEjbBusinessMethods	Before	Stateless	At entry of all SessionBean methods, which are not standard ejb methods.

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_Before_SessionEjbMethods	Before	Stateless	At entry of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate
EJB_Before_SessionEjbSemanticMethods	Before	Stateless	At entry of methods: SessionBean.ejbCreate SessionBean.ejbPostCreate
HttpSessionDebug	Around	Built-in	getSession - Inspects returned HTTP session Before and after calls to methods: getAttribute setAttribute removeAttribute At inspection points, the approximate session size is computed and stored as the payload of a generated event. The size is computed by flattening the session to a byte-array. If an error is encountered while flattening the session, a negative size is reported.
JDBC_Before_CloseConnection	Before	Stateless	Before calls to methods: Connection.close
JDBC_After_CloseConnection	After	Stateless	After calls to methods: Connection.close
JDBC_Around_CloseConnection	Around	Around	Before and after calls to methods: Connection.close
JDBC_Before_CommitRollback	Before	Stateless	Before calls to methods: Connection.commit Connection.rollback
JDBC_After_CommitRollback	After	Stateless	After calls to methods: Connection.commit Connection.rollback
JDBC_Around_CommitRollback	Around	Around	Before and after calls to methods: Connection.commit Connection.rollback
JDBC_Before_Execute	Before	Stateless	Before calls to methods: Statement.execute* PreparedStatement.execute*
JDBC_After_Execute	After	Stateless	After calls to methods: Statement.execute* PreparedStatement.execute*

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JDBC_Around_Execute	Around	Around	Before and after calls to methods: Statement.execute* PreparedStatement.execute*
JDBC_Before_GetConnection	Before	Stateless	Before calls to methods: Driver.connect DataSource.getConnection
JDBC_After_GetConnection	After	Stateless	After calls to methods: Driver.connect DataSource.getConnection
JDBC_Around_GetConnection	Around	Around	Before and after calls to methods: Driver.connect DataSource.getConnection
JDBC_Before_Statement	Before	Stateless	Before calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch ResultSet.setCommand
JDBC_After_Statement	After	Stateless	After calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch ResultSet.setCommand
JDBC_Around_Statement	Around	Around	Before and after calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch ResultSet.setCommand
JMS_Before_AsyncMessageReceived	Before	Stateless	At entry of methods: MessageListener.onMessage
JMS_After_AsyncMessageReceived	After	Stateless	At exits of methods: MessageListener.onMessage
JMS_Around_AsyncMessageReceived	Around	Around	At entry and exits of methods: MessageListener.onMessage
JMS_Before_MessageSent	Before	Stateless	Before call to methods: QueueSender send
JMS_After_MessageSent	After	Stateless	After call to methods: QueueSender send
JMS_Around_MessageSent	Around	Around	Before and after call to methods: QueueSender send

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
JMS_Before_SyncMessageReceived	Before	Stateless	Before calls to methods: MessageConsumer.receive*
JMS_After_SyncMessageReceived	After	Stateless	After calls to methods: MessageConsumer.receive*
JMS_Around_SyncMessageReceived	Around	Around	Before and after calls to methods: MessageConsumer.receive*
JMS_Before_TopicPublished	Before	Stateless	Before call to methods: TopicPublisher.publish
JMS_After_TopicPublished	After	Stateless	After call to methods: TopicPublisher.publish
JMS_Around_TopicPublished	Around	Around	Before and after call to methods: TopicPublisher.publish
JNDI_Before_Lookup	Before	Stateless	Before calls to javax.naming.Context lookup methods Context.lookup*
JNDI_After_Lookup	After	Stateless	After calls to javax.naming.Context lookup methods: Context.lookup*
JNDI_Around_Lookup	Around	Around	Before and after calls to javax.naming.Context lookup methods Context.lookup*
JTA_Before_Commit	Before	Stateless	At entry of methods: UserTransaction.commit
JTA_After_Commit	After	Stateless advice	At exits of methods: UserTransaction.commit
JTA_Around_Commit	Around	Around	At entry and exits of methods: UserTransaction.commit
JTA_Before_Rollback	Before	Stateless	At entry of methods: UserTransaction.rollback
JTA_After_Rollback	After	Stateless advice	At exits of methods: UserTransaction.rollback
JTA_Around_Rollback	Around	Around	At entry and exits of methods: UserTransaction.rollback
JTA_Before_Start	Before	Stateless	At entry of methods: UserTransaction.begin
JTA_After_Start	After	Stateless advice	At exits of methods: UserTransaction.begin
JTA_Around_Start	Around	Around	At entry and exits of methods: UserTransaction.begin

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

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
MDB_Before_MessageReceived	Before	Stateless	At entry of methods: MessageDrivenBean.onMessage
MDB_After_MessageReceived	After	Stateless	At exits of methods: MessageDrivenBean.onMessage
MDB_Around_MessageReceived	Around	Around	At entry and exits of methods: MessageDrivenBean.onMessage
MDB_Before_Remove	Before	Stateless	At entry of methods: MessageDrivenBean.ejbRemove
MDB_After_Remove	After	Stateless	At exits of methods: MessageDrivenBean.ejbRemove
MDB_Around_Remove	Around	Around	At entry and exits of methods: MessageDrivenBean.ejbRemove
MDB_Before_SetMessageDriven Context	Before	Stateless	At entry of methods: MessageDrivenBean.setMessageDrivenContext
MDB_After_SetMessageDriven Context	After	Stateless	At exits of methods: MessageDrivenBean.setMessageDrivenContext
MDB_Around_SetMessageDriven Context	Around	Around	At entry and exits of methods: MessageDrivenBean.setMessageDrivenContext
Servlet_Before_Service	Before	Stateless	At method entries of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter
Servlet_After_Service	After	Stateless	At method exits of servlet/jsp methods: HttpJspPage._jspService Servlet.service HttpServlet.doGet HttpServlet.doPost Filter.doFilter
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 (Cont.) Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
Servlet_Before_Session	Before	Stateless	Before calls to servlet methods: HttpServletRequest.getSession HttpSession.setAttribute/ putValue HttpSession.getAttribute/ getValue HttpSession.removeAttribute/ removeValue HttpSession.invalidate
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_After_Session	After	Stateless	After 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
Servlet_After_Tags	After	Stateless	After calls to jsp methods: Tag.doStartTag Tag.doEndTag
Servlet_Around_Tags	Around	Around	Before and after calls to jsp methods: Tag.doStartTag Tag.doEndTag

B.2 Diagnostic Action Library

The Diagnostic Action Library includes the following actions:

- [TraceAction](#)
- [DisplayArgumentsAction](#)
- [TraceElapsedTimeAction](#)
- [TraceMemoryAllocationAction](#)
- [StackDumpAction](#)
- [ThreadDumpAction](#)
- [MethodInvocationStatisticsAction](#)
- [MethodMemoryAllocationStatisticsAction](#)

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. Some actions (for example, `TraceElapsedTimeAction`) generate an event payload.

B.2.1 TraceAction

`TraceAction` is a stateless action that is compatible with Before and After monitor types.

`TraceAction` generates a trace event at the 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
- Module name
- Location in code from where the action was called. The location information includes:
 - Class name
 - Method name
 - Method signature
 - Line number
 - Thread name
- Payload carried by the diagnostic context, if any

B.2.2 DisplayArgumentsAction

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

DisplayArgumentsAction generates an instrumentation event at the affected location in the program execution to capture method arguments or a return value.

When executed, this action causes an instrumentation event that is dispatched to the events archive. When attached to Before monitors, the instrumentation event captures input arguments to the joinpoint (for example, method arguments). When attached to After monitors, the instrumentation event captures the return value from the joinpoint. The event carries the following information:

- Timestamp
- Context identifier from the diagnostic context that 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
- Module name
- Location in code from where the action was called. The location information includes:
 - Class name
 - Method name
 - Method signature
 - Line number
 - Thread name
- 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

B.2.3 TraceElapsedTimeAction

TraceElapsedTimeAction is an Around action that is compatible with Around monitor types.

TraceElapsedTimeAction generates two events: one before and one after the location in the program execution.

When executed, this action captures the timestamps before and after the execution of an 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 carries the following information:

- Timestamp

- Context identifier from the diagnostic context that 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
- Module name
- Location in code from where the action was called. This location information consists of:
 - Class name
 - Method name
 - Method signature
 - Line number
 - Thread name
- Payload carried by the diagnostic context, if any
- Elapsed time processing the joinpoint, as event payload, in nanoseconds

B.2.4 TraceMemoryAllocationAction

`TraceMemoryAllocationAction` uses the JRocket API to trace the number of bytes allocated by a thread during a method call. This action is very similar to `TraceElapsedTimeAction`, with the exception that the memory allocated within a method call is traced.

The `TraceMemoryAllocationAction` action:

- Creates an instrumentation event that is persisted.
- Can be used from delegating and custom monitors.

B.2.5 StackDumpAction

`StackDumpAction` is a stateless action that is compatible with Before and After monitor types.

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

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

- Timestamp
- Context identifier from the diagnostic context that 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
- Module name
- Location in code from where the action was called. This location information consists of:
 - Class name
 - Method name
 - Method signature
 - Line number
 - Thread name
- Payload carried by the diagnostic context, if any
- Stack trace as an event payload

B.2.6 ThreadDumpAction

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

ThreadDumpAction generates an instrumentation event at the affected location in the program execution to capture a thread dump, if the underlying VM supports it. JDK 1.6 (Oracle JRockit and Sun) supports this action.

This action generates an instrumentation event that is dispatched to the events archive. This action may be used only with the JRockit JVM. It is ignored when used with other JVMs. It captures the thread dump as event payload. The event carries the following information:

- Timestamp
- Context identifier from the diagnostic context that 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
- Module name
- Location in code from where the action was called. This location information consists of:
 - Class name
 - Method name

- Method signature
- Line number
- Thread name
- Payload carried by the diagnostic context, if any
- Thread dump as an event payload

B.2.7 MethodInvocationStatisticsAction

MethodInvocationStatisticsAction is an Around action that is compatible with Around monitor types.

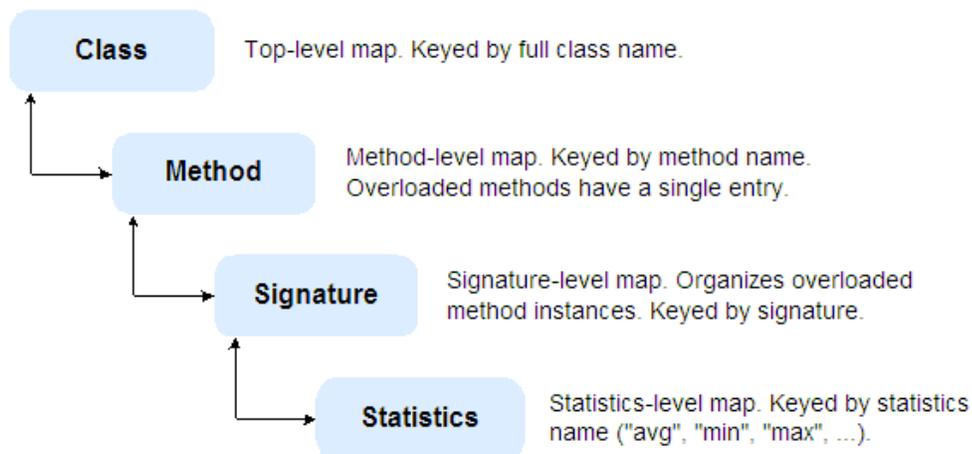
MethodInvocationStatisticsAction captures performance metrics around a joinpoint in memory without persisting an event in the Archive for each invocation. The statistics are collected and made available through the WLDFInstrumentationRuntimeMBean. The collected statistics are also consumable by the Harvester and the Watch-Notifications components. This makes it possible to create watch rules that can combine request information from the instrumentation system and metric information from other run-time MBeans.

Some of the statistics that can be captured include the following:

- Number of invocations
- Average execution time (in nanoseconds)
- Standard deviation in observed execution time
- Minimum execution time
- Maximum execution time

The WLDFInstrumentationRuntimeMBean instance for a given scope exposes the data collected from MethodInvocationStatisticsAction instances, which are attached to configured Diagnostic Around monitors, using the MethodInvocationStatistics attribute. The MethodInvocationStatistics attribute contains a hierarchy of Map objects, keyed as shown in [Figure B-1](#).

Figure B-1 Structure of MethodInvocationStatistics Attribute



The following semantics are used in the MethodInvocationStatistics attribute::

```

MethodInvocationStatistics ::= Map<className, MethodMap>
MethodMap ::= Map<methodName, MethodParamsSignatureMap>
  
```

```

MethodParamsSignatureMap ::= Map<MethodParamsSignature, MethodDataMap>
MethodDataMap ::= <MetricName, Statistic>
MetricName ::= min | max | avg | count | sum | sum_of_squares | std_deviation

```

Because the `MethodInvocationStatisticsAction` only captures information in memory, and does not persist that information in the Archive, this action does not incur the I/O overhead of other instrumentation actions. This makes this action a lightweight mechanism for capturing performance statistics and helping identify bottlenecks in your application. You can navigate through the map structures and identify the low performing parts of your application.

B.2.7.1 Instrumenting an Application with `MethodInvocationStatisticsAction` and Querying the Results

This section shows an example of instrumenting the Avitek Medical Records (MedRec) sample application with a custom monitor that uses `MethodInvocationStatisticsAction`. This example then shows using WLST online to query the performance statistics that have been collected, which can be done by navigating the `WLDFInstrumentationRuntimeMBean` instance associated with the instrumented application.

WLST online provides simplified access to MBeans. While JMX APIs require you to use JMX object names to interrogate MBeans, WLST enables you to navigate a hierarchy of MBeans in a similar fashion to navigating a hierarchy of files in a file system. For more information, see "Navigating and Interrogating MBeans" in *Oracle WebLogic Scripting Tool*.

The following subsections are included in this example:

- [Configuring the Custom Monitor to Use `MethodInvocationStatisticsAction`](#)
- [Using WLST to Query Method Performance Statistics](#)

Note: Code examples demonstrating Java EE APIs and other WebLogic Server features are provided with your WebLogic Server installation. To work with these examples, select the custom installation option when installing WebLogic Server, and select to install the Server Examples. For more information, see "Code Examples and Sample Applications" in *Introduction to Oracle WebLogic Server*.

B.2.7.1.1 Configuring the Custom Monitor to Use `MethodInvocationStatisticsAction` As of WebLogic Server 10.3, it is no longer necessary to create a `weblogic-diagnostics.xml` file in the application's META-INF directory to configure a custom monitor. Instead, you can complete all the required steps from the Administration Console, as described in the following steps for instrumenting the MedRec sample application:

1. In the Domain Structure pane of the Administration Console, select **Deployments**.
2. On the Summary of Deployments page, select **Control**, and click **medrec** in the Deployments table.

The Settings for medrec page is displayed.

3. Select **Configuration > Instrumentation**.
4. In the Diagnostic Monitors in this Module table, click **Add Custom Monitor**.
5. In the Add Custom Monitors page, enter `MethodStatsMonitor` as the monitor name. Optionally, you can enter a brief description.

6. In the Location Type selection box, select **Around**.
7. In the Pointcut text box, enter the following pointcut expression:

```
execution(public * com.bea.medrec.* *(...)) AND NOT
execution(public * com.bea.medrec.* get*(...)) OR
execution(public * com.bea.medrec.* set*(...)) OR
execution(public * com.bea.medrec.* __WL_*(...));
```

This pointcut expression specifies joinpoints for all public methods in classes within packages whose name starts with `com.bea.medrec`, but excludes the following methods:

- All accessor methods
- Methods that begin with the string `__WL_`

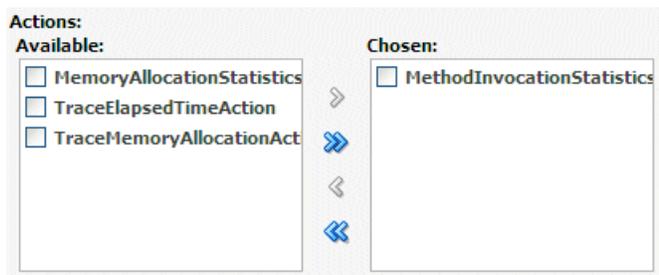
This pointcut expression encompasses a wide variety of public methods and classes in MedRec, but ignores all getter and setter methods, as well as code generated by WebLogic Server.

8. Below the pointcut expression text box, click **OK**.
9. On the Save Deployment Plan page, enter a new path for the deployment plan, or accept the default location, and click **OK**.
10. Select **Configuration > Instrumentation**, and click the name of the new custom monitor, `MethodStatsMonitor`, which is listed in the Diagnostic Monitors in this Module table.

The Settings for `MethodStatsMonitor` page is displayed.

11. In the Actions table, assign **MethodInvocationStatisticsAction** to the custom monitor, as shown in [Figure B-2](#):

Figure B-2 Choosing `MethodInvocationStatisticsAction` for Custom Monitor



12. Click **Save**, at the bottom of the Settings for `MethodStatsMonitor` page.
13. Apply the updated deployment plan to the MedRec application:
 - a. In the Domain Structure pane, select **Deployments**.
 - b. On the Summary of Deployments page, select **Control**, and click the selection box adjacent to `medrec` in the Deployments table, as shown in [Figure B-3](#):

Figure B–3 Selecting the MedRec Deployment

Deployments

Install Update Delete Start Stop Showing 1 to 5 of 5 Previous | Next

<input type="checkbox"/>	Name	State	Health	Type	Deployment Order
<input type="checkbox"/>	browser-starter	Active	OK	Web Application	100
<input type="checkbox"/>	jsf(1.2, 1.2.9.0)	Active		Library	100
<input type="checkbox"/>	jstl(1.2, 1.2.0.1)	Active		Library	100
<input checked="" type="checkbox"/>	medrec	Active	OK	Enterprise Application	1
<input type="checkbox"/>	physician	Active	OK	Enterprise Application	100

Install Update Delete Start Stop Showing 1 to 5 of 5 Previous | Next

- c. Click **Update**.
- d. In the Update Application Assistant page, select **Redeploy this application using the following deployment files**.
- e. Click **Next**, then click **Finish**.

The MedRec application is now redeployed, and the custom monitor MethodStatsMonitor is active.

Note: If Java HotSwap is not enabled, to add a new pointcut to the application's configuration, you need to redeploy the application to enable a custom monitor to be woven into the application code. (However, you can modify most of an application's monitor configuration without requiring a redeploy. This includes changes to the custom monitor's Actions, Properties, EnableDyeFiltering, and Description attributes — that is, anything that does not require bytecode weaving.

However, with HotSwap enabled, you can change any monitor attribute and update the application without the need to redeploy it. For more information, see [Section 14.2, "Using Deployment Plans to Dynamically Control Instrumentation Configuration."](#)

B.2.7.1.2 Using WLST to Query Method Performance Statistics Once MedRec is redeployed, the MethodInvocationStatisticsAction begins capturing method performance statistics as the instrumented code is executed. This section shows how to generate statistics quickly and simply by navigating the MedRec patient application with the custom monitor enabled. This section then shows how to examine those statistics using WLST online.

To capture method performance statistics using the custom monitor configured for MedRec and query the results using WLST, complete the following steps:

1. Start the MedRec application, as described in "WebLogic Server API Examples and Sample Application" in *Introduction to Oracle WebLogic Server*.

Log in as a patient, administrator, or physician, and perform a small number of operations.

2. Invoke WLST online and navigate to the `WLDFInstrumentationRuntimeMBean` instance, as shown in the following example steps:

- a. Connect to the MedRec server:

```
wls:/offline> connect('weblogic','welcome1','localhost:7011')
Connecting to t3://localhost:7011 with userid weblogic ...
Successfully connected to Admin Server 'MedRecServer' that belongs to
domain 'medrec'.
```

- b. Use the `cd` command to navigate to the `WLDFInstrumentationRuntimeMBean` instance associated with the MedRec application:

```
cd('serverRuntime:/WLDFRuntime/WLDFRuntime/WLDFInstrumentationRuntimes/medr
ec')
Location changed to serverRuntime tree. This is a read-only tree with
ServerRuntimeMBean as the root.
For more help, use help(serverRuntime)
```

3. Access specific values collected by `MethodInvocationStatisticsAction` by invoking the following method on the `WLDFInstrumentationRuntimeMBean`:

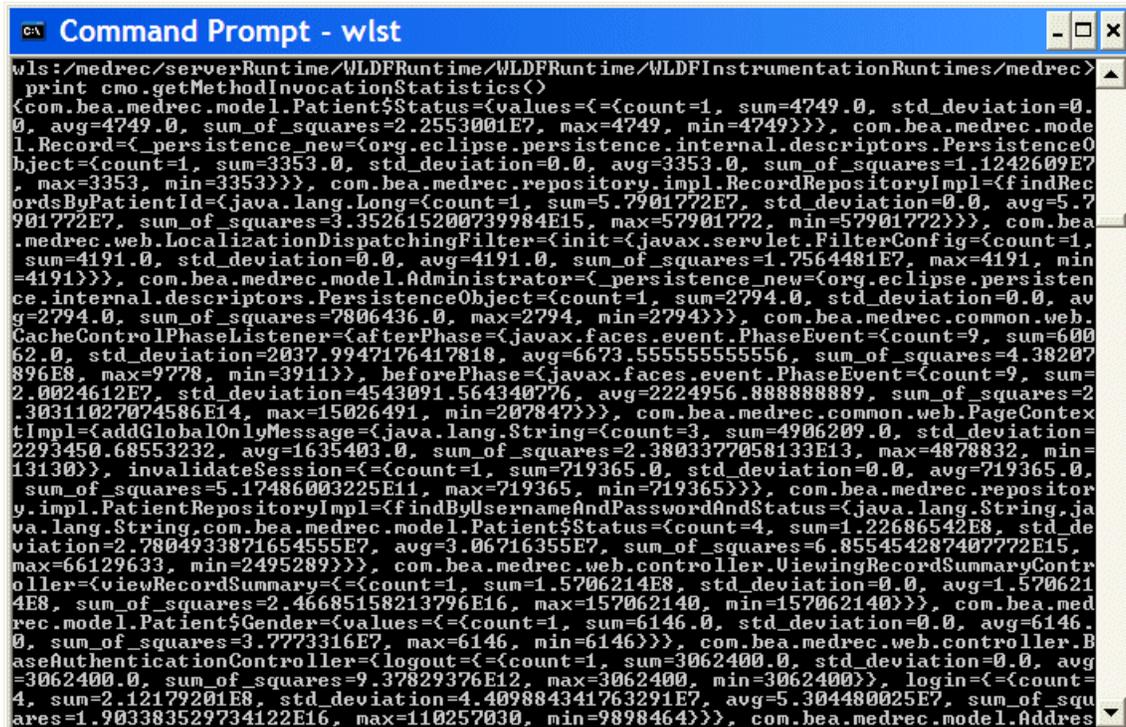
```
public Object getMethodInvocationStatisticsData(String expr) throws
ManagementException;
```

Using WLST interactively, you can pass a lookup expression to this method. The lookup expression specifies the particular subset of values that you are interested in viewing. These values are obtained from the map structure created by `MethodInvocationStatisticsAction`. For example, the following WLST command returns the average execution time (in nanoseconds) of all methods instrumented by `MethodInvocationStatisticsAction`:

```
cmo.getMethodInvocationStatisticsData("(com.bea%)(*)(?)(avg)")
array(java.lang.Object, [3352.0, 3632.0, 145270.0, 4050.5, 8450.916666666666,
1798645.75,
583538.0, 3610515.0, 1.9541031E7, 1.2796319E7, 3.07897E8, 4470.0, 3073.0,
3073.0,
2.4644752E7, 3492.5, 1051530.0, 2794.0, 390552.3333333333, 3632.0, 2095.5,
189409.33333333334,
2607.6666666666665, 2793.6666666666665, 4749.333333333333, 5308.0, 65930.0,
3.3950405E7,
3353.0, 3911.5])
```

Note that if you display the entire set of data values that have been collected, a large amount of information could be displayed in the WLST console, as shown in [Figure B-4](#):

Figure B-4 Displaying All Data Values Collected by MethodInvocationStatisticsAction



```

C:\> Command Prompt - wlst
wls:/medrec/serverRuntime/WLDFRuntime/WLDFRuntime/WLDFInstrumentationRuntimes/medrec>
print cmo.getMethodInvocationStatistics()
{com.bea.medrec.model.Patient$Status={values={count=1, sum=4749.0, std_deviation=0.0, avg=4749.0, sum_of_squares=2.2553001E7, max=4749, min=4749}}, com.bea.medrec.model.Record={_persistence_new={org.eclipse.persistence.internal.descriptors.PersistenceObject={count=1, sum=3353.0, std_deviation=0.0, avg=3353.0, sum_of_squares=1.1242609E7, max=3353, min=3353}}, com.bea.medrec.repository.impl.RecordRepositoryImpl={findRecordsByPatientId={java.lang.Long={count=1, sum=5.7901772E7, std_deviation=0.0, avg=5.7901772E7, sum_of_squares=3.352615200739984E15, max=57901772, min=57901772}}, com.bea.medrec.web.LocalizationDispatchingFilter={init={javax.servlet.FilterConfig={count=1, sum=4191.0, std_deviation=0.0, avg=4191.0, sum_of_squares=1.7564481E7, max=4191, min=4191}}, com.bea.medrec.model.Administrator={_persistence_new={org.eclipse.persistence.internal.descriptors.PersistenceObject={count=1, sum=2794.0, std_deviation=0.0, avg=2794.0, sum_of_squares=7806436.0, max=2794, min=2794}}, com.bea.medrec.common.web.CacheControlPhaseListener={afterPhase={javax.faces.event.PhaseEvent={count=9, sum=60062.0, std_deviation=2037.9947176417818, avg=6673.555555555556, sum_of_squares=4.38207896E8, max=9778, min=3911}}, beforePhase={javax.faces.event.PhaseEvent={count=9, sum=2.0024612E7, std_deviation=4543091.564340776, avg=2224956.888888889, sum_of_squares=2.30311027074586E14, max=15026491, min=207847}}, com.bea.medrec.common.web.PageContextImpl={addGlobalOnlyMessage={java.lang.String={count=3, sum=4906209.0, std_deviation=2293450.68553232, avg=1635403.0, sum_of_squares=2.3803377058133E13, max=4878832, min=13130}}, invalidateSession={count=1, sum=719365.0, std_deviation=0.0, avg=719365.0, sum_of_squares=5.17486003225E11, max=719365, min=719365}}, com.bea.medrec.repository.impl.PatientRepositoryImpl={findByUsernameAndPasswordAndStatus={java.lang.String, java.lang.String, com.bea.medrec.model.Patient$Status={count=4, sum=1.22686542E8, std_deviation=2.7804933871654555E7, avg=3.06716355E7, sum_of_squares=6.855454287407772E15, max=66129633, min=2495289}}, com.bea.medrec.web.controller.ViewingRecordSummaryController={viewRecordSummary={count=1, sum=1.5706214E8, std_deviation=0.0, avg=1.5706214E8, sum_of_squares=2.46685158213796E16, max=157062140, min=157062140}}, com.bea.medrec.model.Patient$Gender={values={count=1, sum=6146.0, std_deviation=0.0, avg=6146.0, sum_of_squares=3.7773316E7, max=6146, min=6146}}, com.bea.medrec.web.controller.BaseAuthenticationController={logout={count=1, sum=3062400.0, std_deviation=0.0, avg=3062400.0, sum_of_squares=9.37829376E12, max=3062400, min=3062400}}, login={count=4, sum=2.12179201E8, std_deviation=4.409884341763291E7, avg=5.304480025E7, sum_of_squares=1.903383529734122E16, max=110257030, min=9898464}}, com.bea.medrec.model.Address

```

As an alternative, you can create a WLST script to invoke MethodInvocationStatistics and to format the collected data so that it is more easily read, as in [Example B-1](#):

Example B-1 Using WLST to Invoke MethodInvocationStatistics and Display Results

```

import sys

def getPositionalArgument(pos, default):
    value=None
    try:
        value=sys.argv[pos]
    except:
        value=default
    return value

url = getPositionalArgument(1, "t3://localhost:7001")
user = getPositionalArgument(2, "weblogic")
password = getPositionalArgument(3, "welcome1")
appName = getPositionalArgument(4, "myapp")

connect(user,password,url)
serverRuntime()
cd('/WLDFRuntime/WLDFRuntime/WLDFInstrumentationRuntimes/' + appName)

print "# Class Method | Count | Min | Max | Average | Std-dev |"
stats=cmo.getMethodInvocationStatistics()
for className in stats.keySet():
    classMap=stats.get(className)
    for methodName in classMap.keySet():
        methodMap=classMap.get(methodName)
        for sig in methodMap.keySet():
            str= className + " | " + methodName + "(" + sig + ")"

```

```

sigMap=methodMap.get(sig)
count=sigMap.get('count')
min=sigMap.get('min')
max=sigMap.get('max')
avg=sigMap.get('avg')
std_deviation=sigMap.get('std_deviation')
print str, "|", count, "|", min, "|", max, "|", avg, "|", std_deviation, "|"

```

The following shows the output produced by the WLST script shown in [Example B-1](#):

```

# Class Method | Count | Min | Max | Average | Std-dev |
jsp_servlet.__index __isStale() | 1 | 1378000 | 1378000 | 1378000.0 | 0.0 |
jsp_servlet.__index __getBytes(java.lang.String) | 3 | 1000 | 754000 | 252666.66666666666 | 354497.1399351795 |
jsp_servlet.__index __staticIsStale(weblogic.servlet.jsp.StaleChecker) | 1 | 861000 | 861000 | 861000.0 | 0.0 |
jsp_servlet.__index __jspService(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) | 2 | 70000 | 2113000 | 1091500.0 | 1021500.0 |
jsp_servlet.__index$MyMap containsKey(java.lang.Object) | 2 | 2000 | 101000 | 51500.0 | 49500.0 |
jsp_servlet.__index$MyMap containsValue(java.lang.Object) | 2 | 1000 | 2000 | 1500.0 | 500.0 |

```

B.2.7.2 Configuring the Harvester to Collect MethodInvocationStatisticsAction Data

To configure the Harvester to collect data gathered by `MethodInvocationStatisticsAction` instances, you must configure an instance of `WLDFHarvesterBean` using the following attribute:

```
Name=weblogic.management.runtime.WLDFInstrumentationRuntimeMBean
```

The scope is selected by the instance configuration.

The attribute specification defines the data that is collected by the Harvester. You can access the successive elements of the map by using the following notation:

```
MethodInvocationStatistics(className) (methodName) (methodParamSignature)
(metricName)
```

In the preceding notation:

- *className* represents the fully qualified Java class name. You can use the asterisk (*) wildcard character in a class name.
- *methodName* selects a specific method from the given class. You can use the asterisk (*) wildcard character in a method name.
- *methodParamSignature* represents a string that is a comma-separated list of a method's input argument types. Only the Java type names, without the argument names, are included in the signature specification. As in the Java language, the order of the parameters in the signature is significant.

This element also supports the asterisk (*) wildcard character, which can be used to specify the entire list of input argument types for a given method. The asterisk (*) wildcard character matches zero or more argument types at the position following its occurrence in the *methodParamSignature* expression.

You can also use the question mark (?) wildcard character to match a single argument type at any given position in the ordered list of parameter types.

Both of these wildcard characters can appear anywhere in the expression. See ["MethodInvocationStatistics Examples"](#) on page B-20.

- *metricName* represents the statistics to be harvested. You can use the asterisk (*) wildcard character in this key to harvest all of the supported metrics.

MethodInvocationStatistics Examples

Consider a class with the following overloaded methods:

```

package.com.foo;
public interface Bar {
    public void doIt();
    public void doIt(int a);
    public void doIt(int a, String s);
    public void doIt(String a, int b);
    public void doIt(String a, String b);
    public void doIt(String[] a);
    public void doNothing();
    public void doNothing(com.foo.Baz);
}

```

Table B-3 provides examples that show to use `MethodInvocationStatisticsAction` to harvest various statistics.

Table B-3 *MethodInvocationStatisticsAction* Examples

The following <code>MethodInvocationStatisticsAction</code> instance configuration causes the following to be harvested
<code>MethodInvocationStatistics(com.foo.Bar) (*) (*) (*)</code>	All statistics for all methods on <code>com.Foo.Bar</code> .
<code>MethodInvocationStatistics(com.foo.Bar) (doIt) () (*)</code>	All statistics for the <code>doIt ()</code> method that has no input arguments.
<code>MethodInvocationStatistics(com.foo.Bar) (doIt) (*) (*)</code>	All statistics for all <code>doIt ()</code> methods.
<code>MethodInvocationStatistics(com.foo.Bar) (doIt) (int, *) (*)</code>	All statistics for the <code>doIt (int)</code> and <code>doIt (int, String)</code> methods.
<code>MethodInvocationStatistics(com.foo.Bar) (doIt) (String[]) (*)</code>	All statistics for the <code>doIt (String[])</code> method. Note that array parameters are specified by the use of a pair of square brackets (<code>[]</code>) following the type name. Space characters are insignificant for the Harvester.
<code>MethodInvocationStatistics(com.foo.Bar) (doIt) (String, ?) (*)</code>	All statistics for <code>doIt ()</code> methods that have two input parameters and <code>String</code> as the first argument type. In this example class, this instance configuration matches the following methods: <ul style="list-style-type: none"> ▪ <code>doIt (String, int)</code> ▪ <code>doIt (String, String)</code>
<code>MethodInvocationStatistics(com.foo.Bar) (doNothing) (com.foo.Baz) (min,max)</code>	The min and max execution time for the <code>doNothing ()</code> method that has the single input parameter of type <code>com.foo.Baz</code> .

Note: Using a wildcard character in the `className` specification can have a negative impact on performance.

B.2.7.3 Configuring Watch Rules Based on `MethodInvocationStatistics` Metrics

You can use the same syntax described in the previous sections to use `MethodInvocationStatistics` metrics in a watch rule. You can create meaningful watch rules that do not use a wildcard character in the `MetricName` element by specifying whether you want the `min`, `max`, `avg`, `count`, `sum`, `sum_of_squares`, or `std_deviation` variable for a given method.

B.2.7.4 Using JMX to Collect Data

When using straight JMX to collect data, you can potentially impact server performance negatively if you invoke the `getAttribute("MethodInvocationStatistics")` method on the `WLDFInstrumentationRuntimeMBean`. This occurs because, depending on the instrumented classes, the nested map structure can contain a lot of data that involves expensive serialization.

When you use JMX to collect data, Oracle recommends using the `getMethodInvocationStatisticsData(String)` method.

B.2.8 MethodMemoryAllocationStatisticsAction

The `MethodMemoryAllocationStatisticsAction` uses the JRockit API to track the number of bytes allocated by a thread during a method call. Statistics are kept in-memory on the memory allocations, and no instrumentation events are created by this action.

The `MethodMemoryAllocationStatisticsAction` is very similar to the existing `MethodInvocationStatisticsAction`. However, statistics tracked by `MethodMemoryAllocationStatisticsAction` are related to the memory allocated within a method call.

The `MethodInvocationStatisticsAction` does not create an instrumentation event. When JRockit is available, the statistics are available through the `WLDFInstrumentationRuntimeMBean`.

The following statistics for each method are kept:

- `count`
- `min`
- `max`
- `avg`
- `sum`
- `sum_of_squares`
- `std_deviation`

Using Wildcards in Expressions

WLDF allows for the use of wildcards in instance names within the <harvested-instance> element, and also provides drill-down and wildcard capabilities in the attribute specification of the <harvested-attribute> element.

WLDF also allows the same wildcard capabilities for instance names in Harvester watch rules, as well as specifying complex attributes in Harvester watch rules.

These capabilities are discussed in the following sections:

- [Using Wildcards in Harvester Instance Names](#)
- [Specifying Complex and Nested Harvester Attributes](#)
- [Using the Accessor with Harvested Complex or Nested Attributes](#)
- [Using Wildcards in Watch Rule Instance Names](#)
- [Specifying Complex Attributes in Harvester Watch Rules](#)

C.1 Using Wildcards in Harvester Instance Names

When specifying instance names within the <harvested-instance> element, you can:

- Express the instance name in non-canonical form, allowing you to specify the property list of the ObjectName out of order
- Express the ObjectName as a JMX ObjectName query pattern without concern as to the order of the property list.
- Use zero or more asterisk (*) wildcard characters in any of the values in the property list of an ObjectName, such as Name=*
- Use zero or more asterisk (*) wildcard characters to replace any character sequence in a canonical ObjectName string. In this case, you must ensure that any properties of the ObjectName not substituted by a wildcard character are in canonical form.

C.1.1 Examples

The instance specification in [Example C-1](#) indicates that all instances of the WorkManagerRuntimeMBean are to be harvested. This is equivalent to not providing any instance-name qualification in the <harvested-type> declaration.

Example C-1 Harvesting All Instances of an MBean

```
<harvested-type>  
  <name>weblogic.management.runtime.WorkManagerRuntimeMBean</name>  
  <harvested-instance>*<harvested-instance>
```

```

    <known-type>true</known-type>
    <harvested-attribute>PendingRequests</harvested-attribute>
</harvested-type>

```

[Example C-2](#) shows a JMX ObjectName pattern as the <harvested-instance> value:

Example C-2 Using a JMX ObjectName Pattern

```

<harvested-type>
  <name>com.acme.CustomMBean</name>
  <harvested-instance>adomain:Type=MyType,*</harvested-instance>
  <known-type>>false</known-type>
</harvested-type>

```

In [Example C-3](#), some of the values in the ObjectName property list contain wildcard characters:

Example C-3 Using Wildcards in the Property List

```

<harvested-type>
  <name>com.acme.CustomMBean</name>
  <harvested-instance>adomain:Type=My*,Name=*,*</harvested-instance>
  <known-type>>false</known-type>
</harvested-type>

```

In [Example C-4](#), all harvestable attributes of all instances of com.acme.CustomMBean are to be harvested, but only those in which the instance name contains the string Name=mybean.

Example C-4 Harvesting All Attributes of Multiple Instances

```

<harvested-type>
  <name>com.acme.CustomMBean</name>
  <harvested-instance>*Name=mybean*</harvested-instance>
  <known-type>true</known-type>
</harvested-type>

```

C.2 Specifying Complex and Nested Harvester Attributes

The Harvester provides the ability to access metric values nested within complex attributes of an MBean. A complex attribute can be a map or list object, a simple POJO, or different nestings of these types of objects. For example:

- anObject.anAttribute
- arrayAttribute[1]
- mapAttribute(aKey)
- aList[1] (aKey)

In addition, wildcard characters can be used for list indexes and map keys to specify multiple elements within a collection of those types. The following wildcard characters are available:

- You can use the asterisk (*) wildcard character to specify all key values for Map attributes.
- You can use the percent (%) wildcard character to replace parts of a Map key string and identify a group of keys that match a particular pattern.

You can also specify a discrete set of key values by using a comma-separated list.

For example:

- `aList[1] (partial%Key%)`
- `aList[*] (key1, key3, keyN)`
- `aList[*] (*)`

In the last example, where the asterisk (*) wildcard character is used for the index to a list and as the key value to a nested map object, nested arrays of values are returned.

Embedding the asterisk (*) wildcard character in a comma-separated list of map keys is equivalent to specifying all map keys. For example, the following two specifications are equivalent:

- `aList[*] (key1, *, keyN)`
- `aList[*] (*)`

Note: Leading or trailing spaces will be stripped from a map key unless the map key is enclosed within quotation marks.

Using a map key pattern can result in a large number of elements being scanned, returned, or both. The larger the number of elements in a map, the bigger the impact is on performance.

The more complex the matching pattern is, the more processing time is required.

C.2.1 Examples

To use drill-down syntax to harvest the nested State property of the HealthState attribute on the ServerRuntime MBean, use the diagnostic descriptor shown in [Example C-5](#).

Example C-5 Using Drill-Down Syntax

```
<harvester>
  <sample-period>10000</sample-period>
  <harvested-type>
    <name>weblogic.management.runtime.ServerRuntimeMBean</name>
    <harvested-attribute>HealthState.State</harvested-attribute>
  </harvested-type>
</harvester>
```

To harvest the elements of an array or list, the Harvester supports a subscript notation in which a value is referred to by its index position in the array or list. For example, to refer to the first element in the array attribute `URLPatterns` in the `ServletRuntimeMBean`, specify `URLPatterns[0]`. [Example C-6](#) shows referencing all elements of `URLPatterns` using a wildcard character.

Example C-6 Using a Wildcard Character to Reference All Elements of an Array

```
<harvester>
  <sample-period>10000</sample-period>
  <harvested-type>
    <name>weblogic.management.runtime.ServletRuntimeMBean</name>
    <harvested-attribute>URLPatterns[*]</harvested-attribute>
  </harvested-type>
</harvester>
```

To harvest the elements of a map, each individual value is referenced by the key enclosed in parentheses. Multiple keys can be specified as a comma-delimited list, in which case the values corresponding to specified keys in the map are harvested, as shown in the following examples.

The following example shows the following
<code><harvested-attribute>MyMap(Foo)</harvested-attribute></code>	Harvesting the value from the map with key <code>Foo</code> .
<code><harvested-attribute>MyMap(Foo,Bar)</harvested-attribute></code>	Harvesting the value from the map with keys <code>Foo</code> and <code>Bar</code> .
<code><harvested-attribute>MyMap(Foo%Bar)</harvested-attribute></code>	Using the percent (%) wildcard character with a key specification to harvest all values from the map if their keys start with <code>Foo</code> and end with <code>Bar</code> .
<code><harvested-attribute>MyMap(*)</harvested-attribute></code>	Harvesting all values from a map by using the asterisk (*) wildcard character.
<code><harvested-attribute>MyBeanMyMap(Foo)</harvested-attribute></code>	The MBean has a JavaBean attribute <code>MyBean</code> , which has a nested map type attribute <code>MyMap</code> . This example harvests this value from the map that has the key <code>Foo</code> .

C.3 Using the Accessor with Harvested Complex or Nested Attributes

While a large number of complex or nested attributes can be specified as a single expression in terms of the Harvester and Watch and Notifications configuration, the actual metrics are persisted in terms of each individually gathered metric.

For example, the attribute specification `mymap(*) . (a, b, c)` maps to the following actual nested attributes:

```
mymap(key1) . a
mymap(key1) . b
mymap(key1) . c
mymap(key2) . a
mymap(key2) . b
mymap(key2) . c
```

Each of the preceding six metrics are stored in a separate record in the `HarvestedDataArchive`, with the shown attribute names stored in the `ATTRNAME` column in each corresponding record. The values in the `ATTRNAME` column are the values you must use in Accessor queries when retrieving them from the archive.

The following are examples of query strings:

```
NAME="foo:Name=MyMBean" ATTRNAME="mymap(key1) . a"
NAME="foo:Name=MyBean" ATTRNAME LIKE "mymap(%).a"
NAME="foo:Name=MyMBean" ATTRNAME MATCHES "mymap\((.*?)\) . a"
```

C.4 Using Wildcards in Watch Rule Instance Names

Within Harvester watch rules, you can use the asterisk (*) wildcard character to specify portions of an `ObjectName`. This gives you the ability to watch for multiple instances of multiple types.

For example, to specify the `OpenSocketsCurrentCount` attribute for all instances of the `ServerRuntimeMBean` that begin with the name `managed`, use the following syntax:

```
#{com.bea:*Name=managed*Type=ServerRuntime*//OpenSocketCurrentCount}
```

As an alternative, you can use JMX `ObjectName` query patterns, as in the following example:

```
#{mydomain:Key1=MyMBean,*//simpleAttribute}
```

Note: The `ObjectName` query pattern syntax supported by the Harvester is determined by whatever is supported by the underlying JMX implementation. The preceding example demonstrates the syntax supported by JDK 5 and later. For information about the full syntax that is supported, see the description of the `javax.management.ObjectName` class corresponding to the version of the JDK with which your installation of WebLogic Server is configured.

C.5 Specifying Complex Attributes in Harvester Watch Rules

You can specify complex attributes (a collection, an array type or an Object with nested intrinsic attribute types) within Harvester watch rule expressions. There are several ways to do this.

The following example shows a drill-down into a nested attribute in a complex type, which is then checked to see if it is greater than 0:

```
#{somedomain:name=MyMbean//complexAttribute.nestedAttribute} > 0
```

You can also use wildcard characters to specify multiple Map keys. The following wildcard characters are available for specifying complex attributes:

- You can use an asterisk character (`*`) to specify all key values for Map attributes.
- You can use a percent character (`%`) to replace parts of a Map key string and to identify a group of keys that match a particular pattern.

In addition, you can use a comma-separated list to specify a discrete set of key values.

For example:

```
#{[com.bea.foo.BarClass]//aList[*].(some%partialKey%).(aValue,bValue)} > 0
```

The rule in the preceding example examines all elements of the `aList` attribute on all instances of `com.bea.foo.BarClass`, drilling down into a nested map for all keys starting with the text `some` and containing the text `partialKey` afterwards. The returned values are assumed to be Map instances, from which values for the keys `aValue` and `bValue` are compared to determine if they are greater than 0.

When using the `MethodInvocationStatistics` attribute on the `WLDFInstrumentationRuntime` type, the system needs to determine the type from the variable. If the system cannot determine the type when validating the attribute expression, the expression is not valid. For example, the following expression is not valid:

```
#{ com.bea:Name=myScope, * //MethodInvocationStatistics(...).(...
```

You must explicitly declare the type in this situation, as shown in the following example that shows drilling down into the nested map structure:

```
$(com.bea:Name=hello,Type=WLDFInstrumentationRuntime,*/MethodInvocationStatistics  
(*)(*)(*)(count)) >= 1
```

WebLogic Scripting Tool Examples

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

- [Example: Dynamically Creating DyeInjection Monitors](#)
- [Example: Configuring a Watch and a JMX Notification](#)
- [Example: Writing a JMXWatchNotificationListener Class](#)
- [Example: Registering MBeans and Attributes For Harvesting](#)
- [Example: Setting the WLDF Diagnostic Volume](#)
- [Example: Capturing a Diagnostic Image](#)
- [Example: Retrieving a JFR File from a Diagnostic Image Capture](#)

For information about running WebLogic Scripting Tool (weblogic.WLST) scripts, see "Running WLST from Ant" in *Oracle WebLogic Scripting Tool*. For information about developing JMX applications, see *Developing Manageable Applications With JMX for Oracle WebLogic Server*.

D.1 Example: Dynamically Creating DyeInjection Monitors

This demonstration script (see [Example D-1](#)) shows how to use the weblogic.WLST tool to create a DyeInjection monitor dynamically. This script:

- 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.

The demonstration script in [Example D-1](#) 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 shown in [Example D-2](#). This must be placed in a file named weblogic-diagnostics.xml and placed into the META-INF directory of an application archive. It is also possible to create a monitor using a JSR-88 deployment plan. For more information about deploying applications, see *Deploying Applications to Oracle WebLogic Server*.

Example D-1 Example: Using WLST to Dynamically Create DyInjection Monitors (demoDyeMonitorCreate.py)

```

# Script name: demoDyeMonitorCreate.py
#####
# Demo script showing how to create a DyInjectionMonitor 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 DyInjection 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.
#####
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 DyInjection 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 DyInjection 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()
```

Example D-2 Example: Downstream Monitor Artifact

```
<?xml version="1.0" encoding="UTF-8"?>
<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-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>
```

D.2 Example: Configuring a Watch and a JMX Notification

This demonstration script (see [Example D-3](#)) 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:

- 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 [Section D.3, "Example: Writing a JMXWatchNotificationListener Class"](#)).

- The demoHarvester.py script, which registers the OpenSocketsCurrentCount attribute with the harvester for collection (see [Section D.4, "Example: Registering MBeans and Attributes For Harvesting"](#)).

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.

Example D-3 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/mywldf")
watch=cmo.createWatch("mywatch")
watch.setEnabled(1)
jmxnot=cmo.createJMXNotification("myjmx")
watch.addNotification(jmxnot)
serverRuntime()
cd("/")
on=cmo.getObjectName().getCanonicalName()
watch.setRuleExpression("${"+on+"} > 1")
watch.getRuleExpression()
watch.setRuleExpression("${"+on+"//OpenSocketsCurrentCount} > 1")
watch.setAlarmResetPeriod(10000)
edit()
save()
activate()
disconnect()
exit()

```

D.3 Example: Writing a JMXWatchNotificationListener Class

[Example D-4](#) shows how to write a JMXWatchNotificationListener.

Example D-4 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=WLDWatchJMXNotificationRuntime," +
                "WLDWatchNotificationRuntime=WatchNotification," +
                "WLDRuntime=WLDRuntime"
            );
        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()
}

```

D.4 Example: Registering MBeans and Attributes For Harvesting

This demonstration script shows how to use the `weblogic.WLST` tool to register MBeans and attributes for collection by the WLDF Harvester. This script:

- Connects to a server and boots the server first if necessary.
- Looks up or creates a 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.

Example D-5 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.getObject().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()

```

D.5 Example: Setting the WLDF Diagnostic Volume

By default, neither Oracle JRockit nor WLDF gather data and record most events in a WebLogic Server instance unless specifically configured otherwise. Note that even when WLDF diagnostic volume is set to `Off`, JRockit and WLDF generate global events that have information about the recording settings; for example, JVM metadata events that list active recordings, and WLDF `GlobalInformationEvents` that list the volume level for the domain, server, and machine.

[Example D-6](#) shows changing the WLDF Diagnostic Volume to `Medium`:

Example D-6 Setting WLDF Diagnostic Volume

```

connect()
edit()
startEdit()
cd("Servers/myserver")
cd("ServerDiagnosticConfig")
cd("myserver")
cmo.setWLDFDiagnosticVolume("Medium")
save()
activate()

```

D.6 Example: Capturing a Diagnostic Image

The diagnostic image capture can be created for a WebLogic Server instance in any of the following ways:

- WebLogic Server Administration Console
- WLST script
- Image notification by means of the Watch and Notification component

Note: If WebLogic Server is running in production mode, the server's SSL port must be used when executing the commands included in this script.

[Example D-7](#) show a sample WLST script that captures a diagnostic image. This example does the following:

- Captures an diagnostic image after connecting, and then waits for the image task to complete.
- Uses the `getAvailableCapturedImages()` command to obtain a list of available diagnostic image files in the server's image directory.
- Loops through the list of available images in the diagnostic image capture and saves each image file locally using the `saveDiagnosticImageCaptureFile()` command.

Example D-7 Creating a Diagnostic Image Capture

```
#
# WLST script to capture a WLDF Diagnostic Image and
# retrieve the image files to a local dir.
#
# Usage:
#
# java weblogic.WLST captureImage.py [username] [passwd] [url] [output-dir]
#
# where
#
#   username      Username to use to connect
#   passwd        Password for connecting to server
#   url           URL to connect to the server
#   output-dir    Path to place saved entries
#
from java.io import File

# Retrieve a positional argument if it exists; if not,
# the provided default is returned.
#
# Params:
# pos   The integer location in sys.argv of the parameter
# default The default value to return if the parameter does not exist
#
# returns the value at sys.argv[pos], or the provided default if necessary
def getPositionalArgument(pos, default):
    value=None
    try:
        value=sys.argv[pos]
    except:
        value=default
    return value

# Credential arguments
uname=getPositionalArgument(1, "weblogic")
passwd=getPositionalArgument(2, "welcome1")
url=getPositionalArgument(3, "t3://localhost:7001")
outputDir=getPositionalArgument(4, ".")

connect(uname, passwd, url)
serverRuntime()
currentDrive=currentTree()

# Capture a new diagnostic image
try:
    cd("serverRuntime:/WLDFRuntime/WLDFRuntime/WLDFImageRuntime/Image")
    task=cmo.captureImage()
    Thread.sleep(1000)
    while task.isRunning():
```

```

        Thread.sleep(5000)
        cmo.resetImageLockout();
    finally:
        currentDrive()

# List the available diagnostic image files in the server's image capture dir
images=getAvailableCapturedImages()
if len(images) > 0:
    # For each diagnostic image found, retrieve image file, renaming it as
    # the user sees fit
    for image in images:
        saveName=outputDir+File.separator+serverName+'-'+image
        saveDiagnosticImageCaptureFile(image,saveName)

```

D.7 Example: Retrieving a JFR File from a Diagnostic Image Capture

The following example shows retrieving the JRockit Flight Recorder (JFR) file from each diagnostic image capture located in the image destination directory on the server and copying it to a local directory. This example script does the following:

- Connects to WebLogic Server, passing the required credentials.
- Creates a diagnostic image capture.
- Obtains a list of the available diagnostic image files in the server's configured image directory.
- For each diagnostic image file, attempts to retrieve the JFR file and save it to a local directory, ensuring that each file is renamed as necessary to avoid any from being overwritten.

Note: If WebLogic Server is running in production mode, the server's SSL port must be used when executing the commands included in this script.

Example D-8 Retrieving a Diagnostic Image Capture File

```

#
# WLST script to capture a WLDF Diagnostic Image and
# save the FlightRecording.jfr entry locally
#
# Usage:
#
# java weblogic.WLST captureImageEntry.py [username] [passwd] [url] [output-dir]
[image-suffix]
#
# where
#
# username      Username to use to connect
# passwd        Password for connecting to server
# url           URL to connect to the server
# output-dir    Path to place saved entries
# image-suffix  Suffix to use to rename JFR image entries locally
#
import os.path
from java.io import File

# Retrieve a positional argument if it exists; if not,
# the provided default is returned.

```

```

#
# Params:
# pos   The integer location in sys.argv of the parameter
# default The default value to return if the parameter does not exist
#
# returns the value at sys.argv[pos], or the provided default if necessary
def getPositionalArgument(pos, default):
    value=None
    try:
        value=sys.argv[pos]
    except:
        value=default
    return value

# Credential arguments
uname=getPositionalArgument(1, "weblogic")
passwd=getPositionalArgument(2, "welcome1")
url=getPositionalArgument(3, "t3://localhost:7001")
outputDir=getPositionalArgument(4, ".")
imageSuffix=getPositionalArgument(5, "_WLS")

connect(uname, passwd, url)
serverRuntime()
currentDrive=currentTree()

# Capture a new diagnostic image capture file
try:
    cd("serverRuntime:/WLDFRuntime/WLDFRuntime/WLDFImageRuntime/Image")
    task=cmo.captureImage()
    Thread.sleep(1000)
    while task.isRunning():
        Thread.sleep(5000)
    cmo.resetImageLockout();
finally:
    currentDrive()

# List the available diagnostic image captures in the server's image capture dir
images=getAvailableCapturedImages()
if len(images) > 0:
    # For each image capture found, retrieve the JFR entry and save it to a local
    # file, renaming it to avoid collisions in the event there are multiple
    # diagnostic image capture files with JFR entries.
    i=0
    for image in images:
        saveName=outputDir+File.separator+"FlightRecording_"
        "+imageSuffix+"+"-"+str(i)+".jfr"
        while os.path.exists(saveName):
            i+=1
            saveName=outputDir+File.separator+"FlightRecording_"
            "+imageSuffix+"+"-"+str(i)+".jfr"
        saveDiagnosticImageCaptureEntryFile(image, 'FlightRecording.jfr', saveName)
        i+=1

```

Glossary

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 Diagnostics 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 Diagnostics 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 Diagnostics 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 Diagnostics 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 Diagnostics 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.

diagnostic module

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

diagnostic monitor

A diagnostic monitor is a unit of diagnostic code that defines the following:

1. The locations in a program where the diagnostic code is added
2. The diagnostic actions that are executed at those locations

WebLogic Server provides a library of useful diagnostic monitors. You can integrate these monitors into server and application classes. Once integrated, the monitors take effect at server startup for server classes, and at 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 Diagnostics Framework supports these types of diagnostic notifications: Java Management Extensions (JMX), Java Message Service (JMS), Simple Mail Transfer Protocol (SMTP), Simple Network Management Protocol (SNMP), and WLDf Image Capture. See also [diagnostic image](#).

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 in diagnostic monitors to determine whether or not a data event should be generated. 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, metadata 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 a harvestable entity 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 *Oracle WebLogic Server MBean Reference*.

The WebLogic Diagnostics Framework only makes run-time MBeans available as harvestable. In order for an MBean to be harvestable, it must be registered in the local WebLogic Server run-time 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.

See also [harvestable entities](#), [harvestable data](#), [Harvester's configuration data set](#).

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.

See also [harvestable entities](#), [harvestable 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 method execution site. 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 Diagnostics Framework, MBean classes are used to configure the service and to monitor its run-time 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. However, for user-defined MBeans, the set of types can grow dynamically as more MBeans appear at run time. 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 user-defined MBeans.

MBean type metadata

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

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

metadata

Metadata is information that describes the information the WebLogic Diagnostics 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 metadata. The metadata made available by means of the Data Accessor includes:

1. A list of supported data store types. For example, SERVER_LOG, HTTP_LOG, and HARVESTED_DATA.
2. A list of available data stores.
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 Diagnostics 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.

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 Diagnostics 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 to inspect server and application classes and insert the diagnostic byte code at well-defined locations, if necessary at class load time. The diagnostic byte code enables the WebLogic Diagnostics 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.

