

Oracle® Fusion Middleware

Configuring and Using the Diagnostics Framework for Oracle WebLogic Server



12c (12.2.1.4.0)

E90839-05

September 2022

The Oracle logo, consisting of the word "ORACLE" in white, uppercase letters, centered within a solid red square.

ORACLE®

Oracle Fusion Middleware Configuring and Using the Diagnostics Framework for Oracle WebLogic Server, 12c (12.2.1.4.0)

E90839-05

Copyright © 2007, 2022, Oracle and/or its affiliates.

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, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

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

Intel and Intel Inside 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, Epyc, and the AMD 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 about 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 unless otherwise set forth in an applicable agreement between you and Oracle. 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, except as set forth in an applicable agreement between you and Oracle.

This document describes and tells how to configure and use the monitoring and diagnostic services provided by WebLogic Diagnostics Framework (WLDF).

Contents

Preface

Documentation Accessibility	xiv
Preface	xv

1 Introduction and Roadmap

What Is the WebLogic Diagnostics Framework?	1-1
Guide to This Document	1-2
Samples and Tutorials	1-4
Avitek Medical Records Application (MedRec) and Tutorials	1-4
WLDF Samples Available for Download	1-4
What's New in This Guide	1-4

2 Overview of the WLDF Architecture

Overview of the WebLogic Diagnostics Framework	2-2
Data Creation, Collection, and Instrumentation	2-3
Archive	2-3
Policies and Actions	2-4
Data Accessor	2-5
Monitoring Dashboard and Request Performance Pages	2-5
Monitoring Dashboard	2-5
Diagnostics Request Performance Page	2-6
Diagnostic Image Capture	2-6
How It All Fits Together	2-7
WLDF Support for Multitenancy	2-8

3 Using the Built-in Diagnostic System Modules

Overview	3-1
Types of Built-in Diagnostic System Modules	3-2
Data Collected by Built-in Diagnostic System Modules	3-2
Configuring a Built-in Diagnostic Module	3-3

Accessing Data Collected by a Built-in Diagnostic System Module	3-4
Using the Monitoring Dashboard	3-5
Using the Metrics Log Table in the Administration Console	3-5
Creating a Custom Diagnostic System Module Based on a Built-in	3-7

4 Using WLDF with Java Flight Recorder

About Java Flight Recorder	4-1
Using Java Flight Recorder with Oracle HotSpot	4-3
Key Features of WLDF Integration with Java Flight Recorder	4-3
Java Flight Recorder Use Cases	4-5
Diagnosing a Critical Failure — The "Black Box"	4-5
Profiling During Performance Testing or in Production	4-5
Real-Time Application Diagnostics and Reporting	4-6
Obtaining the Flight Recording File	4-7
Analyzing Java Flight Recorder Data	4-7
Java Flight Recorder Graphical User Interface	4-8
Analyzing Execution Flow — A Sample Walkthrough	4-9
Displaying Event Data for a Product Subcomponent	4-10
Viewing the Event Log to Display Details	4-10
Tracking Execution Flow by Analyzing an Operative Set	4-12
Expanding the Operative Set and Viewing Correlated Diagnostic Data	4-14
Changing the Location of Temporary JFR Files	4-16

5 Understanding WLDF Configuration

Configuration MBeans and XML	5-2
Tools for Configuring WLDF	5-2
How WLDF Configuration Is Partitioned	5-3
Server-Level Configuration	5-3
Application-Level Configuration	5-3
Configuring Diagnostic Image Capture and Diagnostic Archives	5-4
Configuring Diagnostic Image Capture for Java Flight Recorder	5-4
Configuring Diagnostic System Modules	5-5
About the Resource Descriptor	5-6
WLDF Runtime Control	5-8
Creating a Diagnostic System Module Based on a Configured Resource Descriptor	5-8
Creating a Diagnostic System Module Based on an External Resource Descriptor	5-10
Targeting a Diagnostic System Module to a Server or Cluster	5-11
Dynamically Activating or Deactivating Diagnostic System Modules	5-11
Using WLST to Activate and Deactivate Diagnostic System Modules	5-12

Example	5-12
More Information About Configuring Diagnostic System Modules	5-15
Configuring Diagnostic Modules for Applications	5-16
WLDF Configuration MBeans and Their Mappings to XML Elements	5-16

6 Configuring and Capturing Diagnostic Images

How to Initiate Image Captures	6-1
Configuring Diagnostic Image Captures	6-2
Configuring WLDF Diagnostic Volume	6-2
Low Volume Setting	6-3
Medium Volume Setting	6-4
High Volume Setting	6-5
WLST Commands for Generating an Image Capture	6-5
How Diagnostic Image Capture Is Persisted in the Server's Configuration	6-5
Content of the Captured Image File	6-6
Data Included in the Diagnostics Image Capture File	6-7
WLST Online Commands for Downloading Diagnostics Image Captures	6-8

7 Configuring Diagnostic Archives

Configuring the Archive	7-1
Configuring a File-Based Store	7-2
Configuring a JDBC-Based Store	7-2
Creating WLDF Tables in the Database	7-3
Apache Derby	7-3
Oracle Database	7-4
MySQL	7-7
Configuring JDBC Resources for WLDF	7-8
Retiring Data from the Archives	7-9
Configuring Data Retirement at the Server Level	7-9
Configuring Age-Based Data Retirement Policies for Diagnostic Archives	7-10
Sample Configuration	7-10

8 Configuring the Harvester for Metric Collection

Harvesting, Harvestable Data, and Harvested Data	8-1
Harvesting Data from the Different Harvestable Entities	8-2
Configuring the Harvester	8-2
Configuring the Harvester Sampling Period	8-3
Configuring the Types of Data to Harvest	8-3
Specifying Type Names for WebLogic Server MBeans and Custom MBeans	8-4

Harvesting from the Domain Runtime MBean Server	8-4
When Configuration Settings Are Validated	8-5
Sample Configurations for Different Harvestable Types	8-5
Harvester Performance Considerations	8-6

9 Configuring Policies and Actions

Policies and Actions	9-1
Overview of Policies and Actions Configuration	9-2
Sample Policies and Actions Configuration	9-4

10 Configuring Policies

How Policies Are Configured	10-1
Rule Type	10-3
Expression Language	10-4
Policy Expression	10-4
Actions	10-5
Policy Schedule	10-5
Alarm Options	10-7
Severity Option	10-8
Enablement Option	10-9
Configuring Scheduled Policies	10-9
Configuring Calendar Based Policies	10-9
Configuring Smart Rule Based Policies	10-10
Types of Diagnostic Data that Smart Rules Evaluate	10-11
Smart Rule Example	10-11
Chaining Policies	10-12
Configuring Log Policies	10-13
Configuring Instrumentation Policies	10-13
Creating Complex Policy Expressions Using WLDF Java EL Extensions	10-14
Writing Collected Metrics Policy Expressions Using Beans	10-15
Accessing MBean Data in Collected Metrics	10-16
Working with Complex MBean Attributes	10-17
Performing Bulk Queries on Collected Metrics from MBeans	10-18
Writing Collected Metrics Policy Expressions Using Functions	10-20
Examining Trends in Metric Values over Time	10-21
Extracting and Examining Collected Metrics in Policy Expressions	10-23
Lifecycle of Data Collection	10-24

11 Configuring Actions

Actions Overview	11-2
Types of Actions	11-2
Variables for Customizable Actions	11-3
Action Timeout	11-4
Configuring JMX Actions	11-4
Configuring JMS Actions	11-5
Configuring SNMP Actions	11-6
Configuring Log Actions	11-6
Configuring REST Actions	11-7
Configuring SMTP Actions	11-7
Configuring Image Actions	11-9
Configuring Elastic Actions	11-9
Elastic Scaling Operations Cannot Be Cancelled After Starting	11-11
Limiting Server Shutdown Time During Scale Down Operations	11-11
Configuring Script Actions	11-11
Configuring Heap Dump Actions	11-12
Configuring Thread Dump Actions	11-14

12 Configuring Instrumentation

Concepts and Terminology	12-2
Instrumentation Scope	12-2
Configuration and Deployment	12-2
Joinpoints, Pointcuts, and Diagnostic Locations	12-2
Diagnostic Monitor Types	12-3
Diagnostic Actions	12-4
Instrumentation Configuration Files	12-5
XML Elements Used for Instrumentation	12-6
<Instrumentation> XML Elements	12-6
<wldf-instrumentation-monitor> XML Elements	12-8
Mapping <wldf-instrumentation-monitor> XML Elements to Monitor Types	12-11
Configuring Server-Scoped Instrumentation	12-11
Configuring Application-Scoped Instrumentation	12-12
Comparing System-Scoped to Application-Scoped Instrumentation	12-13
Overview of the Steps Required to Instrument an Application	12-14
Creating a Descriptor File for a Delegating Monitor	12-15
Creating a Descriptor File for a Custom Monitor	12-16
Defining Pointcuts for Custom Monitors	12-17
Annotation-based Pointcuts	12-18

13 Configuring the DyeInjection Monitor to Manage Diagnostic Contexts

Contents, Life Cycle, and Configuration of a Diagnostic Context	13-2
Context Life Cycle and the Context ID	13-2
Dyes, Dye Flags, and Dye Vectors	13-2
Where Diagnostic Context Is Configured	13-3
Overview of the Process	13-3
Configuring the Dye Vector by Using the DyeInjection Monitor	13-4
Dyes Supported by the DyeInjection Monitor	13-6
PROTOCOL Dye Flags	13-7
THROTTLE Dye Flag	13-7
When Diagnostic Contexts Are Created	13-7
Configuring Delegating Monitors to Use Dye Filtering	13-8
How Dye Masks Filter Requests to Pass to Monitors	13-10
Dye Filtering Example	13-10
Using Throttling to Control the Volume of Instrumentation Events	13-11
Configuring the THROTTLE Dye	13-12
How Throttling is Handled by Delegating and Custom Monitors	13-13
Using <code>weblogic.diagnostics.context</code>	13-14

14 Accessing Diagnostic Data With the Data Accessor

Data Stores Accessed by the Data Accessor	14-1
Accessing Diagnostic Data Online	14-2
Accessing Data Using the Administration Console	14-2
Accessing Data Programmatically Using Runtime MBeans	14-3
Using WLST to Access Diagnostic Data Online	14-3
Using the WLDF Query Language with the Data Accessor	14-3
Accessing Diagnostic Data Offline	14-3
Accessing Diagnostic Data Programmatically	14-4
Resetting the System Clock Can Affect How Data Is Archived and Retrieved	14-9

15 Deploying WLDF Application Modules

Deploying a Diagnostic Module as an Application-Scoped Resource	15-2
Using Deployment Plans to Dynamically Control Instrumentation Configuration	15-3
Using a Deployment Plan: Overview	15-4
Creating a Deployment Plan Using <code>weblogic.PlanGenerator</code>	15-4
Sample Deployment Plan for Diagnostics	15-5
Enabling Java HotSwap	15-6

Deploying an Application with a Deployment Plan	15-6
Updating an Application with a Modified Plan	15-7

16 Using the Monitoring Dashboard

Running the Monitoring Dashboard	16-1
Scope of the Diagnostic Information Displayed	16-1
About the Monitoring Dashboard Interface	16-2
View List	16-3
Metric Browser	16-3
View Display Panel	16-6
Understanding How Metrics Are Collected and Presented	16-7
About Metrics and Chart Types	16-7
Current Time Range Charts	16-8
Custom Time Range Charts	16-8
Sequence in which Metrics Data is Displayed	16-8
Notes about Metric Data Retention	16-9
The Parts of a Chart	16-9

17 Configuring and Using WLDF Programmatically

How WLDF Generates and Retrieves Data	17-1
Mapping WLDF Components to Beans and Packages	17-2
Programming Tools	17-4
Configuration and Runtime APIs	17-5
Configuration APIs	17-5
Runtime APIs	17-6
WLDF Packages	17-6
Programming WLDF: Examples	17-7
Example: DiagnosticContextExample.java	17-7
Example: HarvesterMonitor.java	17-8
Notification Listeners	17-8
HarvesterMonitor.java	17-9
Example: JMXAccessorExample.java	17-13

18 Using Debug Patches

Dynamic Application of Debug Patches	18-1
Specifying the Debug Patch Directory	18-1
Configuring the WLDF Debug Patch Agent	18-2
WLST Commands for Debug Patches	18-2
Dynamically Activating a Debug Patch	18-3

A Smart Rule Reference

About the Parameters You Specify for Smart Rules	A-1
Cluster Scope Smart Rules	A-3
ClusterLowThroughput	A-5
ClusterHighProcessCpuLoadAverage	A-6
ClusterHighThroughput	A-8
ClusterLowPendingUserRequests	A-10
ClusterHighStuckThreads	A-11
ClusterLowQueueLength	A-13
ClusterHighPendingUserRequests	A-14
ClusterLowProcessCpuLoadAverage	A-16
ClusterHighIdleThreads	A-18
ClusterLowSystemLoadAverage	A-19
ClusterHighQueueLength	A-21
ClusterLowHeapFreePercent	A-22
ClusterHighSystemLoadAverage	A-24
ClusterHighHeapFreePercent	A-26
ClusterLowSystemCpuLoadAverage	A-27
ClusterLowIdleThreads	A-29
ClusterGenericMetricRule	A-31
ClusterHighSystemCpuLoadAverage	A-33
Server Scope Smart Rules	A-34
ServerLowIdleThreads	A-37
ServerHighThroughput	A-39
ServerGenericMetricRule	A-40
ServerLowPendingUserRequests	A-42
ServerLowProcessCpuLoadAverage	A-43
ServerHighSystemLoadAverage	A-44
ServerLowQueueLength	A-46
ServerLowThroughput	A-47
ServerHighQueueLength	A-48
ServerHighSystemCpuLoadAverage	A-50
ServerHighPendingUserRequests	A-51
ServerLowSystemCpuLoadAverage	A-52
ServerHighHeapFreePercent	A-54
ServerHighStuckThreads	A-55
ServerHighProcessCpuLoadAverage	A-56
ServerLowSystemLoadAverage	A-58

ServerLowHeapFreePercent	A-59
ServerHighIdleThreads	A-61

B WLDF Beans and Functions Reference

WLDF Beans Reference	B-1
clusterRuntime	B-2
domainRuntime	B-3
instrumentationEvent	B-4
log	B-6
partition	B-7
platform	B-8
resource	B-9
runtime	B-9
Functions Reference	B-10
wls:tableChanges	B-11
wls:tableAverages	B-11
wls:extract	B-11
wls:average	B-12
wls:changes	B-13
wls:aliveServersCount	B-13

C WLDF Query Language

Components of a Query Expression	C-1
Supported Operators	C-1
Operator Precedence	C-3
Numeric Relational Operations Supported on String Column Types	C-3
Supported Numeric Constants and String Literals	C-3
About Variables in Expressions	C-4
Creating Policy Expressions	C-4
Creating Log Event Policy Expressions	C-5
Creating Instrumentation Event Policy Expressions	C-5
Creating Harvester Policy Expressions	C-6
Creating Data Accessor Queries	C-7
Data Store Logical Names	C-8
Data Store Column Names	C-9
Creating Log Filter Expressions	C-10
Building Complex Expressions	C-11

D WLDF Instrumentation Library

Diagnostic Monitor Library	D-1
Diagnostic Action Library	D-9
TraceAction	D-10
DisplayArgumentsAction	D-11
TraceElapsedTimeAction	D-11
TraceMemoryAllocationAction	D-12
StackDumpAction	D-12
ThreadDumpAction	D-13
MethodInvocationStatisticsAction	D-14
Instrumenting an Application with MethodInvocationStatisticsAction and Querying the Results	D-15
Configuring the Harvester to Collect MethodInvocationStatisticsAction Data	D-20
Configuring Policies Based on MethodInvocationStatistics Metrics	D-22
Using JMX to Collect Data	D-22
MemoryAllocationStatisticsAction	D-22

E Using Wildcards in Expressions

Using Wildcards in Harvester Instance Names	E-1
Examples	E-1
Specifying Complex and Nested Harvester Attributes	E-2
Examples	E-3
Using the Accessor with Harvested Complex or Nested Attributes	E-4
Using Wildcards in Policy Instance Names	E-5
Specifying Complex Attributes in Harvester Policies	E-6

F WebLogic Scripting Tool Examples

WLST Commands for Diagnostics	F-1
Example: Dynamically Creating DyeInjection Monitors	F-2
Example: Configuring a Policy and a JMX Action	F-4
Example: Writing a JMXWatchNotificationListener Class	F-7
Example: Registering MBeans and Attributes For Harvesting	F-9
Example: Setting the WLDF Diagnostic Volume	F-12
Example: Capturing a Diagnostic Image	F-13
Example: Retrieving a JFR File from a Diagnostic Image Capture	F-14

G WLDF Query Language-Based Policies

Types of Policies	G-1
-------------------	-----

Policy Configuration Options	G-2
Configuring Harvester Policies Using the WLDF Query Language	G-2
Configuring Log Policies Using the WLDF Query Language	G-4
Configuring Instrumentation Policies Using the WLDF Query Language	G-4

Glossary

Preface

This document describes and tells how to configure and use the monitoring and diagnostic services provided by WebLogic Diagnostics Framework (WLDF).

- [Audience](#)
- [Documentation Accessibility](#)
- [Diversity and Inclusion](#)
- [Related Documentation](#)
- [Conventions](#)

Audience

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.

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 that have purchased support 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.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of

these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

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 Server Administration Console to configure WLDF.
- Monitoring Partitions in *Using WebLogic Server Multitenant* describes how to monitor partitions in WebLogic Server Multitenant.
- The WLDF system resource descriptor conforms to the `weblogic-diagnostics.xsd` schema, available at <http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/weblogic-diagnostics.xsd>.

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.

Preface

This document describes and tells how to configure and use the monitoring and diagnostic services provided by WebLogic Diagnostics Framework (WLDF).

- [Audience](#)
- [Documentation Accessibility](#)
- [Diversity and Inclusion](#)
- [Related Documentation](#)
- [Conventions](#)

Audience

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.

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 that have purchased support 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.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

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 Server Administration Console to configure WLDF.
- *Monitoring Partitions in Using WebLogic Server Multitenant* describes how to monitor partitions in WebLogic Server Multitenant.
- The WLDF system resource descriptor conforms to the `weblogic-diagnostics.xsd` schema, available at <http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/weblogic-diagnostics.xsd>.

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.

Convention	Meaning
<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.

1

Introduction and Roadmap

The WebLogic Diagnostics Framework (WLDF) is 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.

- [What Is the WebLogic Diagnostics Framework?](#)
- [Guide to This Document](#)
- [Samples and Tutorials](#)
- [What's New in This Guide](#)

What Is the WebLogic Diagnostics Framework?

The WebLogic Diagnostics Framework (WLDF) is a suite of services and APIs that provide the ability to collect and surface metrics that provide visibility into server and application performance. Independent Software Vendors (ISVs) can use these APIs, using standard interfaces such as WLST, REST, and JMX, to develop custom monitoring and diagnostic tools for integration with WLDF.

The suite of services, components, and APIs provided by WLDF for collecting and analyzing data includes the following:

- **Integration with Oracle HotSpot**—If WebLogic Server is configured with Oracle HotSpot, WLDF can generate diagnostic information about WebLogic Server that is captured in the Java Flight Recorder file.
- **Built-in diagnostic system modules**—A set of diagnostic modules available out-of-the-box that you can enable dynamically to capture basic performance data about the JVM, the WebLogic Server run time, and primary WebLogic Server subsystems, including JDBC data sources, messaging, and Java EE containers, such as servlets, EJBs, and resource adapters. The built-in diagnostic modules can also be cloned and modified, providing a simple way to create custom diagnostic system modules.
- **Monitoring Dashboard**—Graphically presents the current and historical operating state of WebLogic Server and hosted applications, including information gathered by the built-in diagnostic system modules. 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.
- **Diagnostic Image Capture**—Creates a diagnostic snapshot from the server that can be used for post-failure analysis. The diagnostic image capture includes Java Flight Recorder data, if it is available, that can be viewed in Java 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.
- Policies and Actions—Provides the means for monitoring server and application states and sending notifications based on criteria set in the policies.
- 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 [Related Documentation](#).

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. These APIs can be accessed using the JMX and the WebLogic Scripting Tool (WLST), as described in [Configuring and Using WLDF Programmatically](#).

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.

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.
- [Overview of the WLDF Architecture](#) provides a high-level view of the WLDF architecture.
- [Using the Built-in Diagnostic System Modules](#) describes the built-in system diagnostic modules, which are provided by the WebLogic Diagnostics Framework (WLDF) as a simple and easy-to-use mechanism for performing basic health and performance monitoring of a WebLogic Server instance
- [Using WLDF with Java Flight Recorder](#) describes the WLDF integration features with Java Flight Recorder, describes basic usage scenarios, and provides a sample walkthrough of using Java Mission Control to examine WebLogic Server events captured in a Java Flight Recorder file.
- [Understanding WLDF Configuration](#) provides an overview of how WLDF features are configured for servers and applications.
- [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.

- [Configuring Diagnostic Archives](#) describes how to configure and use the WLDF Diagnostic Archive component to persist diagnostic data to a file store or database.
- [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.
- [Configuring Policies and Actions](#) provides an overview of WLDF policies and actions.
- [Configuring Policies](#) describes how to configure policies to monitor server instances and applications for specific conditions and execute actions when those conditions are met.
- [Configuring Actions](#) describes how to configure actions that can be executed by policies.
- [Configuring Instrumentation](#) describes how to add diagnostic instrumentation code to WebLogic Server classes and to the classes of applications running on the server.
- [Configuring the DyeInjection Monitor to Manage Diagnostic Contexts](#) describes how to use the DyeInjection monitor and how to use dye filtering with diagnostic monitors.
- [Accessing Diagnostic Data With the Data Accessor](#) tells how to use the WLDF Data Accessor component to retrieve diagnostic data.
- [Deploying WLDF Application Modules](#) explains how to configure and manage instrumentation for an application as a diagnostics application module.
- [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.
- [Configuring and Using WLDF Programmatically](#) provides an overview of how you can use the JMX API and the WebLogic Scripting Tool (WLST) to configure and use WLDF components.
- [Using Debug Patches](#) describes how to apply debug patches dynamically, allowing you to capture diagnostic information using a patch that is activated and deactivated without requiring a server restart.
- [Smart Rule Reference](#) contains a comprehensive reference of all smart rules provided in WLDF are used as policy predicates, typically for elastic scaling operations in dynamic clusters.
- [WLDF Beans and Functions Reference](#) provides a reference for the beans provided by WLDF, and Java EL functions, that can be used in collected metrics policy expressions to obtain access to common WebLogic Server JMX data sources.
- [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.
- [WLDF Instrumentation Library](#) describes the predefined diagnostic monitors and diagnostic actions that are included in the WLDF Instrumentation Library.
- [Using Wildcards in Expressions](#) discusses how to use wildcards in WLDF expressions.
- [WebLogic Scripting Tool Examples](#) provides examples of how to perform WLDF monitoring and diagnostic activities using the WebLogic Scripting Tool.
- [WLDF Query Language-Based Policies](#) explains how to configure policies with expressions that use the deprecated WLDF query language.
- [Glossary](#) is a glossary of terms used in WLDF.

Samples and Tutorials

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

- [Avitek Medical Records Application \(MedRec\) and Tutorials](#)
- [WLDF Samples Available for Download](#)

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 optionally installed in the WebLogic Server distribution and is available by selecting the Complete Installation type. By default, MedRec is configured post-installation in the `ORACLE_HOME/user_projects/domains/medrec` directory, where `ORACLE_HOME` represents the Oracle home directory on your machine. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

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.

What's New in This Guide

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

2

Overview of the WLDF Architecture

WebLogic Diagnostics Framework (WLDF) consists of various components that work together to collect, archive, and access diagnostic information about a WebLogic Server instance and the application it hosts. This chapter provides an overview of the WLDF architecture, describes its components, and illustrates how all components work together to collect and access diagnostic information about a WebLogic Server and the application 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 [Using the Built-in Diagnostic System Modules](#).

The following topics summarize WLDF and its architectural components:

- [Overview of the WebLogic Diagnostics Framework](#)
- [Data Creation, Collection, and Instrumentation](#)
- [Archive](#)

The Archive component of WLDF captures the state of the system and archives it for future access in diagnosing critical faults in the system. It creates a historical archive using several persistent components.
- [Policies and Actions](#)

The Policies and Actions component of WLDF is used to create automated monitors that observe specific diagnostic states and send notifications based on configured rules.
- [Data Accessor](#)

The Data Accessor component of WLDF provides access to all the data collected by WLDF, including log, event, and metric data. It interacts with the Archive component to get historical data including logged event data and persisted metrics.
- [Monitoring Dashboard and Request Performance Pages](#)

The WebLogic Server Administration Console displays the Monitoring Dashboard and Diagnostics Request Performance pages. The diagnostics data collected is visually represented in these pages. The Monitoring Dashboard displays the current and historical operating state of WebLogic Server and hosted applications. The Diagnostics Request Performance page shows real-time and historical views of method performance information.
- [Diagnostic Image Capture](#)

The Diagnostic Image Capture component captures the key server state as a diagnostic image. The diagnostic image is a diagnostic snapshot of the server state used in diagnosing problems.

- [How It All Fits Together](#)
The components of the WLDF work together to collect data and diagnose faults in running server.
- [WLDF Support for Multitenancy](#)
Oracle WebLogic Server supports multitenancy, which provides a sharable infrastructure for use by multiple organizations. These organizations are a conceptual grouping of your own choosing, which you can think of as tenants. Tenants have access to domain partitions, which provide an isolated slice of the domain's configuration and runtime infrastructure.

Overview of the WebLogic Diagnostics Framework

The WLDF components interact with each other to process data at the server level. WLDF consists of the following components:

- 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
- Policies and Actions 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 Policies and Actions 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 Major WLDF components are shown in [Figure 2-1](#).

Figure 2-1 Major WLDF Components



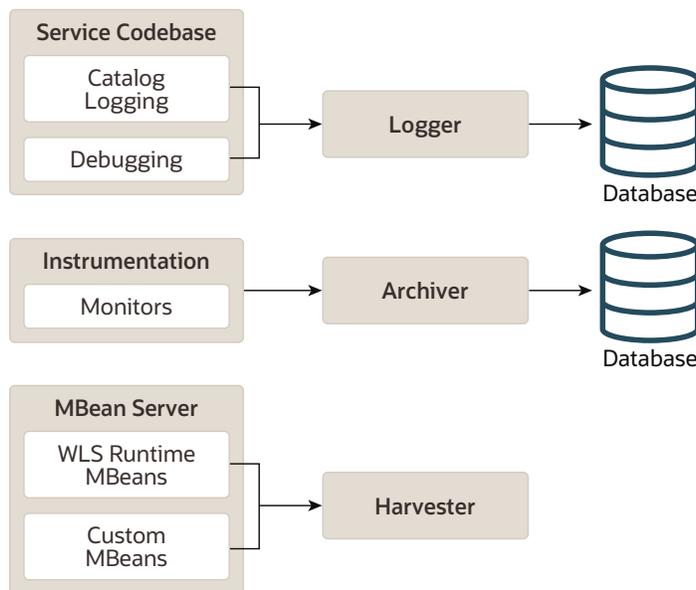
All of the framework components operate at the server level and are only aware of server scope. All the components exist entirely within the server process and participate in the standard server lifecycle. All artifacts of the framework are configured and stored on a per server basis.

Data Creation, Collection, and Instrumentation

Diagnostic data is collected from a number of logically classified sources. The sources are 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 or the Harvester, as shown in [Figure 2-2](#).

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



[Figure 2-2](#) shows that invocations of the server logging infrastructure serve as inline data publishers, and that 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 Policies and Actions system for automated monitoring and to the Archive for persistence.

Archive

The Archive component of WLDF captures the state of the system and archives it for future access in diagnosing critical faults in the system. It creates a historical archive using several persistent components.

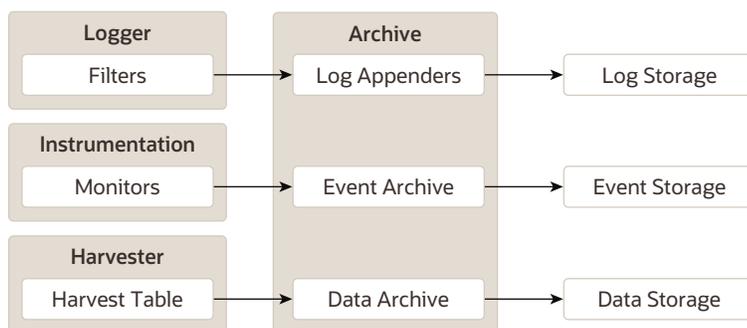
The past state is often critical in diagnosing faults in a system. This requires that the state be captured and archived for future access, creating a historical archive. In WLDF, the Archive

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

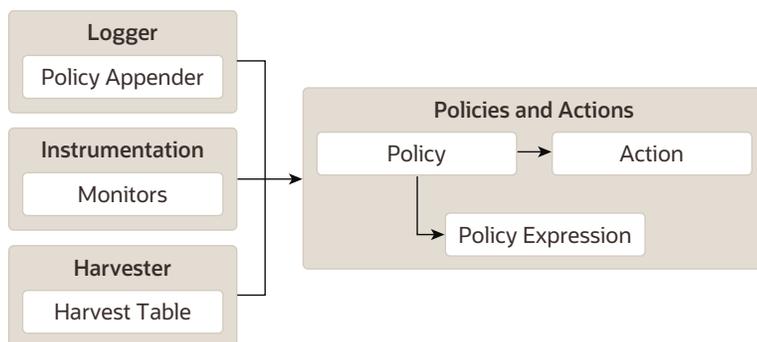


Policies and Actions

The Policies and Actions component of WLDf is used to create automated monitors that observe specific diagnostic states and send notifications based on configured rules.

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

Figure 2-4 Relationship of the Logger and the Harvester to the Policies and Actions System



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

Data Accessor

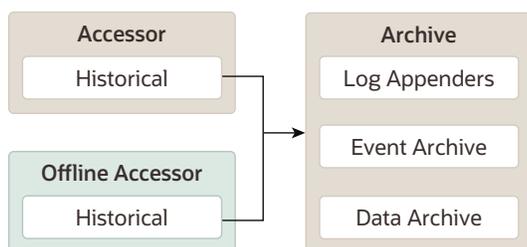
The Data Accessor component of WLDf provides access to all the data collected by WLDf, including log, event, and metric data. It interacts with the Archive component to get historical data including logged event data and persisted metrics.

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



Monitoring Dashboard and Request Performance Pages

The WebLogic Server Administration Console displays the Monitoring Dashboard and Diagnostics Request Performance pages. The diagnostics data collected is visually represented in these pages. The Monitoring Dashboard displays the current and historical operating state of WebLogic Server and hosted applications. The Diagnostics Request Performance page shows real-time and historical views of method performance information.

The following sections provide more information about the web pages that visually display the diagnostic data:

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

Monitoring Dashboard

The Monitoring Dashboard displays the current and historical operating state of WebLogic Server and hosted applications by providing visualizations of metric runtime MBean attributes, which surface some of the more critical runtime 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 runtime metrics for all running WebLogic Server instances in the domain. Built-in views surface some of the more critical runtime 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. See [Using the Monitoring Dashboard](#).

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. See [Creating Request Performance Data](#).

Diagnostic Image Capture

The Diagnostic Image Capture component captures the key server state as a diagnostic image. The diagnostic image is a diagnostic snapshot of the server state used in diagnosing problems.

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 HotSpot, and Java Flight Recorder is enabled, the diagnostic image capture includes all available Java Flight Recorder data from all producers. Furthermore, if WLDF is configured to generate WebLogic Server diagnostic information captured by Java Flight Recorder, the JFR file includes that information as well. The JFR file can be extracted from the diagnostic image capture and viewed in Java Mission Control. See [Using WLDF with Java Flight Recorder](#).

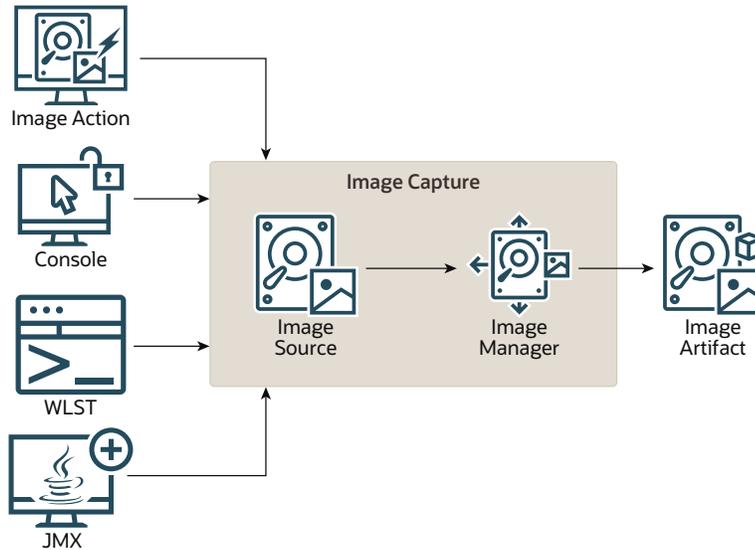
Image Capture support includes:

- 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 action, which is automatically creating a diagnostic image capture in response to the triggering of an associated Harvester policy, Log policy, or Instrumentation policy expression. For example, a Harvester policy that monitors runtime MBean attributes in a running server can execute an image action if the metrics harvested from specific runtime 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 about diagnostic image capture, see:

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

Figure 2-6 Diagnostic Image Capture

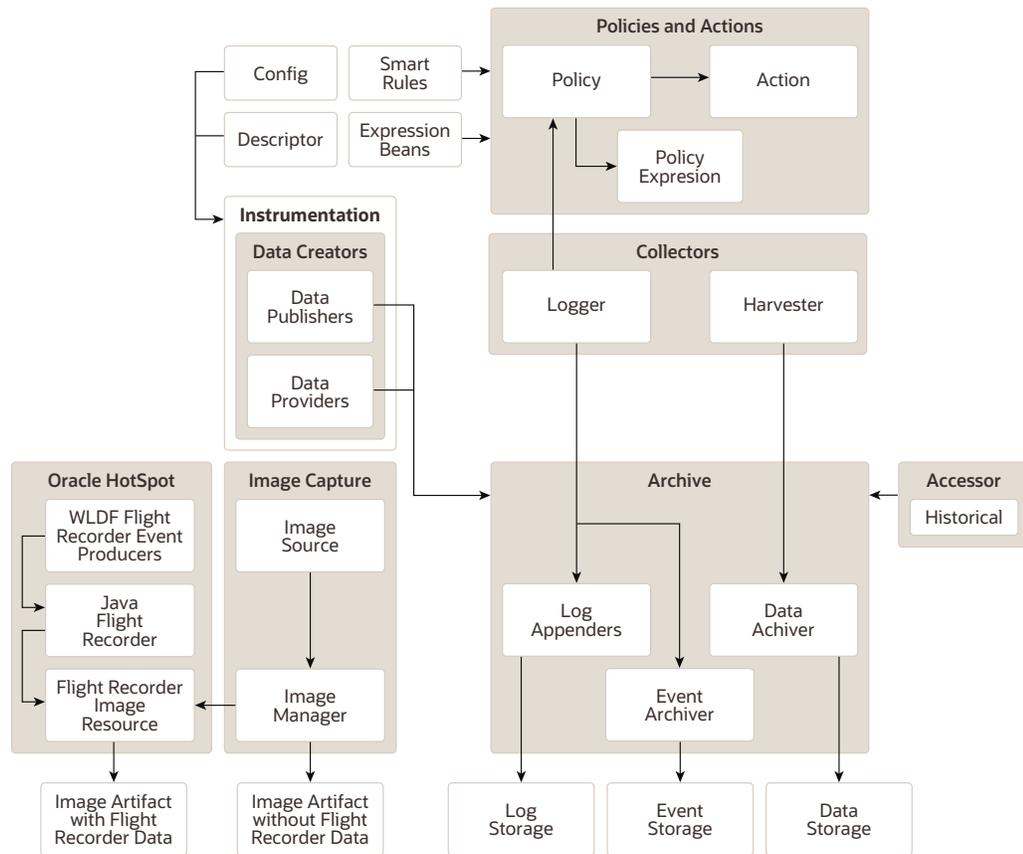


How It All Fits Together

The components of the WLDF work together to collect data and diagnose faults in running server.

[Figure 2-7](#) shows how all the parts of WLDF fit together.

Figure 2-7 Overall View of the WebLogic Diagnostics Framework



WLDF Support for Multitenancy

Oracle WebLogic Server supports multitenancy, which provides a sharable infrastructure for use by multiple organizations. These organizations are a conceptual grouping of your own choosing, which you can think of as tenants. Tenants have access to domain partitions, which provide an isolated slice of the domain's configuration and runtime infrastructure.

 **Note:**

WebLogic Server Multitenant domain partitions, resource groups, resource group templates, virtual targets, and Resource Consumption Management (RCM) are deprecated in WebLogic Server 12.2.1.4.0 and will be removed in the next release.

WebLogic Server MT support includes several diagnostic and monitoring capabilities for monitoring and debugging applications and resources deployed in domain partitions, including the following:

- WebLogic Server MT allows configuration of levels for `java.util.logging` loggers that are used by applications deployed to a partition. In a MT environment,

partition users can configure logger levels within the scope of the partition without affecting other partitions.

- Partition scoped log events to Server log and from partition deployed resources like JMS, SAF, and servlet resources, are made available to partition administrators from the WLDF Data Accessor.
- A partition user can set debug flags on WebLogic Server components so that debug messages can be broadcast on behalf of work performed for that partition, without affecting other partitions.
- The Harvester, and Policies and Actions components, can be defined within a resource group or resource group template deployed to a partition.
- Applications deployed within a partition may be instrumented using WLDF instrumentation.
- Image capture can be initiated either manually by a partition scope user, or by a policy configured in a partition scope diagnostic system module. Only the content specific to the partition is included in the generated diagnostic image.
- WebLogic Server supports the ability to monitor RCM metrics within a partition scope.
- WLDF provides integration with the Java Flight Recorder which enables WebLogic Server events to be included in the JFR flight recording. The JFR flight recording, which includes events from the JVM and from any other event producer, such as WebLogic Server and Oracle Dynamic Monitoring System (DMS), can include `partition-id` and `partition-name` attributes to distinguish data among partitions.

 **Note:**

The data from JFR is not visible or accessible to the MT Administrator.

See Monitoring and Debugging Partitions in *Using WebLogic Server Multitenant*.

3

Using the Built-in Diagnostic System Modules

The built-in diagnostic system modules are provided by the WebLogic Diagnostics Framework (WLDF) as a simple and easy-to-use mechanism for performing basic health and performance monitoring of a WebLogic Server instance.

- [Overview](#)
- [Configuring a Built-in Diagnostic Module](#)
You can configure a built-in diagnostic module to collect data from key WebLogic Server runtime MBeans. Use the WebLogic Server Administration Console or WLST command to configure the diagnostic module.
- [Accessing Data Collected by a Built-in Diagnostic System Module](#)
The built-in diagnostic modules collect data from the WebLogic Server. The collected data is visually represented as graph in the Monitoring Dashboard and as Metric Log tables in the Administration Console.
- [Creating a Custom Diagnostic System Module Based on a Built-in](#)
You can use one of the built-in diagnostic modules as the starting point and create customized diagnostic modules to suit your requirements. Using the built-in diagnostic modules simplifies the process of creating new diagnostic modules.

Overview

The WLDF built-in diagnostic modules collect data from key WebLogic Server run-time MBeans that monitor the main components of a server instance. Those main components are:

- JVM
- WebLogic Server run time
- JDBC, JMS, transaction, and logging services
- Java EE containers hosting servlets, EJBs, and Connector Architecture resource adapters

When configured in a WebLogic Server instance, the built-in diagnostic modules are particularly useful for providing a low-overhead, historical record of server performance. As server workload changes over time, or the performance characteristics change as a result of updates made to the server's configuration, you can examine the data collected by the built-ins to obtain details about performance changes. For example, if you notice a slowdown in the response time of one or more deployed applications, you can use the Monitoring Dashboard or the Metrics Log table in the WebLogic Server Administration Console to examine the data collected by the built-ins for performance bottlenecks associated with one or more WebLogic Server subsystems. Then using other diagnostic tools, such as custom diagnostic modules, policies and actions, or Java Flight Recorder, you can drill down further into details about those bottlenecks to pinpoint specific causes and test the effectiveness of solutions.

In WebLogic domains configured to run in production mode, a built-in diagnostic module is enabled by default in each server instance. (In domains configured to run in development mode, built-ins are disabled by default.) However, a built-in diagnostic module can be

enabled or disabled for a server instance easily and dynamically, using either the WebLogic Server Administration Console or WLST.

Data collected by the built-in diagnostic modules can be accessed easily, using tools such as the Metrics Log table in the WebLogic Server Administration Console or the Monitoring Dashboard. The data can also be accessed programmatically using JMX, WLST, or REST.

- [Types of Built-in Diagnostic System Modules](#)
- [Data Collected by Built-in Diagnostic System Modules](#)

Types of Built-in Diagnostic System Modules

WLDF provides three built-in diagnostic system module types:

- **Low** — Captures the most important data from key WebLogic Server runtime MBeans (enabled by default in production mode).
- **Medium** — Captures additional attributes from the WebLogic Server runtime MBeans captured by **Low**, and also includes data from additional runtime MBeans.
- **High** — Captures the most verbose data from attributes on the WebLogic Server runtime MBeans captured by **Medium**, and also includes data from a larger number of runtime MBeans.

The built-in diagnostic system module type configured for a server instance is specified in the

`WLDFServerDiagnosticMBean.WLDFBuiltinSystemResourceType=string` MBean attribute, where *string* can be set to one of **Low**, **Medium**, **High**, or **None**.

Data Collected by Built-in Diagnostic System Modules

When you enable a built-in diagnostic module in a WebLogic Server instance, WLDF begins collecting data from key WebLogic Server run-time MBeans to obtain information, such as the following:

Data Category	Example of Information Collected
JVM statistics	Amount of available free memory and JVM processor load on host machine.
Thread statistics	Threads being held by a request and the number of pending user requests.
JDBC subsystem statistics	Examples of information collected may include: <ul style="list-style-type: none"> • Number of connections currently in use by applications. • Average amount of time taken to create a physical connection to the database. • Number of leaked connections (that is, connections reserved from the data source but not returned to the data source). • Number of available and idle database connections. • Cumulative, running count of requests for a connection from a data source.

Data Category	Example of Information Collected
JMS subsystem statistics	Examples of information collected may include statistics about: <ul style="list-style-type: none"> • WebLogic JMS consumers and producers, such as number of messages pending by a consumer or producer. • JMS destinations, such as current number of messages in the destination, and number of pending messages in the destination. • The current number of connections to WebLogic Server.
Logging subsystem statistics	The number of log messages that the WebLogic Server instance has generated.
JTA subsystem	Examples of information collected may include: <ul style="list-style-type: none"> • Number of active transactions on the server. • Total number of seconds that transactions were active for all committed transactions.
Java EE container statistics	Examples of information collected may include statistics about: <ul style="list-style-type: none"> • EJBs, such as the EJB cache, EJB pool, and EJB transaction statistics. • Servlets, such as the average amount of time all invocations of a servlet have executed since the servlet was created.

**Note:**

The specific configuration of each built-in diagnostic module is internal to WebLogic Server and subject to change in a future release.

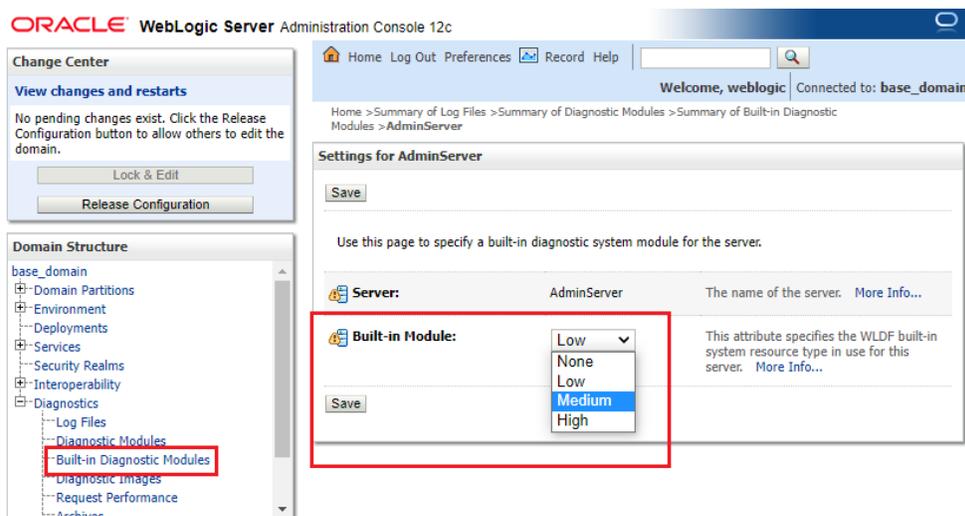
Configuring a Built-in Diagnostic Module

You can configure a built-in diagnostic module to collect data from key WebLogic Server runtime MBeans. Use the WebLogic Server Administration Console or WLST command to configure the diagnostic module.

Using the WebLogic Server Administration Console, you can perform the following steps to configure the built-in diagnostic module:

1. Select **Built-in Diagnostic Modules** in the Diagnostics area of the WebLogic Server Administration Console home page.

Figure 3-1 WebLogic Server Administration Console home page



2. In the Summary of Built-in Diagnostic Modules page, select the server instance for which you want to configure a built-in diagnostics module.
3. In the Settings for *server-name* page, select the built-in diagnostics module type you want to configure: *Low*, *Medium*, or *High*.

By default, once you select a built-in diagnostics module for a server instance, it is automatically activated and begins collecting the data in the Archive. From the Summary of Built-in Diagnostic Modules page, you can later deactivate the built-in module if desired by setting it to *None*.

Note:

Although WebLogic Server allows you to target multiple diagnostic system modules to a server instance, only one built-in diagnostic module type may be activated at any time.

For more information about configuring built-in system diagnostic modules in the WebLogic Server Administration Console, see the following topics in *Oracle WebLogic Server Administration Console Online Help*.

- [Activate a built-in diagnostic system module](#)
- [Select a built-in diagnostics system module](#)
- [Disable a built-in diagnostic system module](#)

Accessing Data Collected by a Built-in Diagnostic System Module

The built-in diagnostic modules collect data from the WebLogic Server. The collected data is visually represented as graph in the Monitoring Dashboard and as Metric Log tables in the Administration Console.

The following sections describe the different ways you can access the data collected by a built-in diagnostic system module:

- [Using the Monitoring Dashboard](#)
- [Using the Metrics Log Table in the Administration Console](#)

Using the Monitoring Dashboard

The Monitoring Dashboard is a good choice for viewing the data collected by the built-in diagnostic system modules. The Metric Browser simplifies selecting the specific MBean attributes you want to graph, and the tools available for customizing views and drilling down on data of interest are easy to use.

The Monitoring Dashboard does not provide a means to select the data collected by a particular diagnostic system module, including any of the built-ins. However, for a given server instance, you can easily select the runtime MBean instance and corresponding metrics you want to display. See [Using the Monitoring Dashboard](#), for complete details about the Monitoring Dashboard.

To view data collected by a built-in module:

1. Launch the Monitoring Dashboard, which you can do from the WebLogic Server Administration Console or separately in a Web browser. See [Running the Monitoring Dashboard](#).
2. In the Monitoring Dashboard, create a custom view, as described in [Create custom views](#) in the *Oracle WebLogic Server Administration Console Online Help*.
3. Navigate to the Metric Browser and select the following:
 - The server instance for which you want to display data collected by the built-in diagnostic system module.
 - The **Collected Metrics Only** button.
 - The MBean type and instance corresponding to the runtime MBean for which the data was collected.
4. Create a chart.
5. Open the Chart Properties dialog box, select **Custom (only applies to collected metrics)**, and specify the time range during which the data you wish to view was collected.
6. In the Metric Browser, select the metrics you want to display.

Using the Metrics Log Table in the Administration Console

You can access data collected by the built-in diagnostic system modules in the Metrics Log table, which is displayed by selecting the log file name **HarvestedDataArchive** in the Summary of Log Files console page.

Figure 3-2 Summary of Log Files

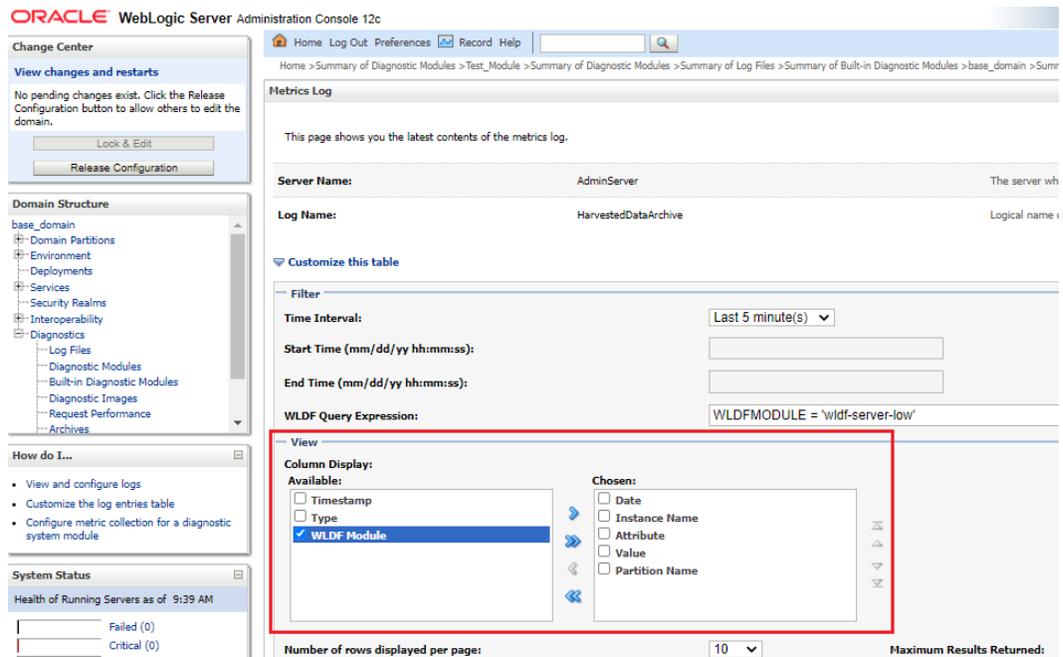
The screenshot shows the Oracle WebLogic Server Administration Console interface. The main content area is titled "Summary of Log Files" and contains a table of log files. The table has three columns: Name, Type, and Server. The "HarvestedDataArchive" log file is selected, indicated by a blue radio button. The table lists the following log files:

Name	Type	Server
<input type="radio"/> DataSourceLog	Data Source Profile Log	AdminServer
<input type="radio"/> DomainLog	Domain Log	AdminServer
<input type="radio"/> EventsDataArchive	Instrumentation	AdminServer
<input checked="" type="radio"/> HarvestedDataArchive	Metric Data	AdminServer
<input type="radio"/> HTTPAccessLog	HTTP Access	AdminServer
<input type="radio"/> ServerLog	Server Log	AdminServer

To display the metrics collected by a built-in diagnostic module in the Metrics Log table of the WebLogic Server Administration Console, complete the following steps:

1. Select **HarvestedDataArchive** in the Summary of Log Files console page, and click **View**.
2. In the Metrics Log console page, click **Customize this table**.
3. To constrain the table to display only metrics collected by a built-in diagnostic module, enter a string in the **WLDF Query Expression** field that specifies that built-in, such as the following:
 - `WLDFMODULE = 'wldf-server-low'` — Specifies metrics collected by the Low built-in diagnostic module.
 - `WLDFMODULE LIKE 'wldf-server-%'` — Specifies metrics collected by any of the built-in diagnostic modules.
4. In the Available column display box, select **WLDF Module**, and click the right arrow to move it to the Chosen box.

Figure 3-3 Metrics Log



5. Click Apply.

If the Archive contains a large amount of data, you can filter the Metrics Log table further by adding expressions to the WLDf query string. For example:

- `(WLDfMODULE LIKE 'wldf-server-%') AND (NAME LIKE '%Name=examples-demo%')` restricts the number of metrics displayed to harvested attributes with an instance name that includes the string `examples-demo`.
- `(WLDfMODULE LIKE 'wldf-server-%') AND (TYPE LIKE '%ServletRuntime%')` restricts the number of metrics displayed to harvested attributes of the `ServletRuntimeMBean`.
- `(WLDfMODULE LIKE 'wldf-server-%') AND (TYPE LIKE '%JMSDestination%' AND ATTRNAME = 'MessagesCurrentCount')` restricts the number of metrics displayed to harvested instances of the `JMSDestinationRuntimeMBean.MessagesCurrentCount` attribute.

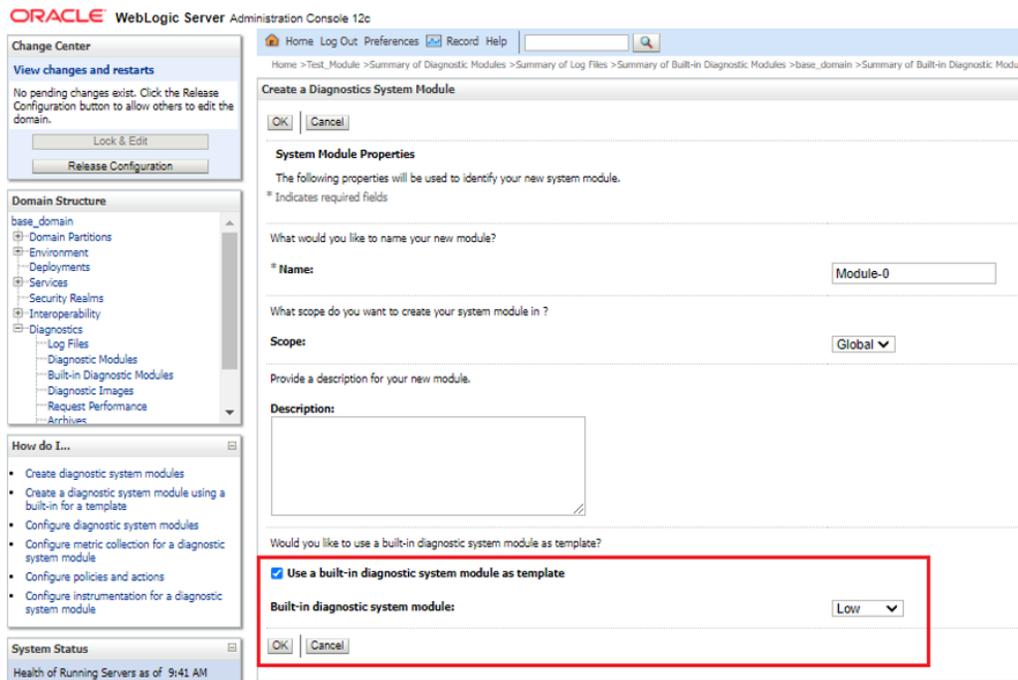
For more information about WLDf query expressions, see [WLDf Query Language](#).

Creating a Custom Diagnostic System Module Based on a Built-in

You can use one of the built-in diagnostic modules as the starting point and create customized diagnostic modules to suit your requirements. Using the built-in diagnostic modules simplifies the process of creating new diagnostic modules.

From the Create a Diagnostics System Module page of the WebLogic Server Administration Console, you can select **Use a built-in diagnostic system module as a template**, and then select the particular built-in module upon which you want to base your new diagnostic module.

Figure 3-4 Create a Diagnostics System Module



After you select the particular built-in module you want to use as a template, and click **OK**, you can navigate to the Settings for *module-name* page and make the following customizations as appropriate:

- The **Collected Metrics** tab displays the set of metrics configured for the particular built-in you are using as a template. By default, all the metrics configured in the built-in are enabled in your custom diagnostic module:
 - To delete a configured metric, select it and click **Delete**.
 - To add a metric not configured with the built-in used as a template, click **New**, and use the **Create a Metric** assistant to specify the metric.

For more information about customizing the metrics configured for your diagnostic system module, see [Configure metric collection for a diagnostic system module](#) in *Oracle WebLogic Server Administration Console Online Help*.

- The **Policies and Actions** tab displays a set of policies and actions that are configured but not actually enabled in the built-in module you are using for a template. The set of policies and actions available represent those that cover typical server-level situations for which actions are generally desirable when certain state criteria thresholds are met. You can delete, or add to the set of policies and actions as appropriate. You may also update threshold values to suit your situations.

 **Note:**

If you use one or more policies and actions that are configured in the built-in module, you must make sure that they are enabled in your diagnostic system module. In the Policies and Actions tab of the WebLogic Server Administration Console, select **Enabled** then click **Save**.

For more information about targeting and activating diagnostic system modules, see [Configuring Diagnostic System Modules](#).

4

Using WLDF with Java Flight Recorder

The integration of the WebLogic Diagnostics Framework (WLDF) with Java Flight Recorder enables WebLogic Server events to be propagated to the Java Flight Recorder for inclusion in a common data set for runtime or post-incident analysis. The Flight Recording data is also included in WLDF diagnostic image captures, which enables you to capture flight recording snapshots based on WLDF policies. You can use this capability to capture and analyze, in a single view, the runtime system information for both the JVM and the Fusion Middleware components running on it.

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.

- [About Java Flight Recorder](#)
Java Flight Recorder is a performance monitoring and profiling tool that records diagnostic information on a continuous basis. The Java Flight Recorder is available even when there is a catastrophic failure such as a system crash.
- [Using Java Flight Recorder with Oracle HotSpot](#)
Java Flight Recorder is available with Oracle Hotspot. If WebLogic Server is configured with Oracle HotSpot, Java Flight Recorder is disabled by default. Enable the Java Flight Recorder to capture the WLDF diagnostic data.
- [Key Features of WLDF Integration with Java Flight Recorder](#)
WLDF integration with Java Flight Recorder provides several useful features, including having WebLogic Server events captured in the flight recording, the ability to throttle the volume of data captured, tools for downloading diagnostic image captures, and more.
- [Java Flight Recorder Use Cases](#)
Java Flight Recorder helps to resolve important diagnostic issues such as diagnosing critical failure, and examining and reporting runtime data. When a critical failure occurs, the data captured by Java Flight Recorder is useful for failure analysis. Likewise, capturing data at specific time and at runtime help to diagnose data after and before a particular event.
- [Obtaining the Flight Recording File](#)
The diagnostic image capture is a single Java Flight Recorder (JFR) file that contains individual images produced by different server subsystems. The JFR file is included in the diagnostic image as `FlightRecording.jfr`.
- [Analyzing Java Flight Recorder Data](#)
You can extract the JFR file from the diagnostic image capture and use Java Mission Control to examine the contents of the JFR file. JFR provides graphical user interface which gives view of all the event information recorded in the JFR file.

About Java Flight Recorder

Java Flight Recorder is a performance monitoring and profiling tool that records diagnostic information on a continuous basis. The Java Flight Recorder is available even when there is a catastrophic failure such as a system crash.

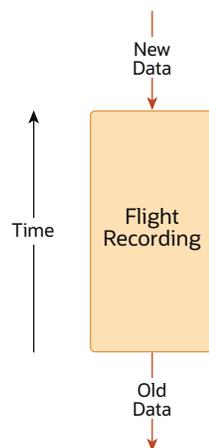
Java Flight Recorder is available in Oracle HotSpot. When WebLogic Server is configured with HotSpot, Java Flight Recorder is not enabled by default. See [Using Java Flight Recorder with Oracle HotSpot](#) for information about how to enable Java Flight Recorder with WebLogic Server.

 **Note:**

For the most current information about configurations supported in this release of WebLogic Server, see Oracle Fusion Middleware Supported System Configurations on the Oracle Technology Network.

Java Flight Recorder 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 4-1](#).

Figure 4-1 Circular Flight Recording Buffer



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 JFR file can be analyzed at any time, using Java Mission Control, to examine the details of system execution flow that occurred leading up to an event.

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

Java Flight Recorder provides the following key benefits:

- Designed to run continuously — When Java Flight Recorder 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 — Java Flight Recorder combines data generated by tools such as the Runtime Analyzer and the Latency Analysis Tool and presents it in one place.
- Integration with event providers — HotSpot includes a set of APIs that allow Java Flight Recorder to monitor additional system components, including WebLogic Server, Oracle Dynamic Monitoring System (DMS), and other Oracle products.

For more information about Java Flight Recorder, see Java Flight Recorder Runtime Guide at the following location:

<http://docs.oracle.com/javacomponents/index.html>

Using Java Flight Recorder with Oracle HotSpot

Java Flight Recorder is available with Oracle Hotspot. If WebLogic Server is configured with Oracle HotSpot, Java Flight Recorder is disabled by default. Enable the Java Flight Recorder to capture the WLDF diagnostic data.

To enable Java Flight Recorder, you must specify the following JVM options in the WebLogic Server instance in which the JVM runs:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

Note:

The sequence in which you specify JVM options to Hotspot is very important. The options are processed from left to right, and option values are overwritten if there are duplicates. Therefore, note the following:

- HotSpot does not recognize the `FlightRecorder` option unless it is preceded by the `UnlockCommercialFeatures` option.
- If you specify only the `FlightRecorder` option, or you specify `FlightRecorder` before specifying `UnlockCommercialFeatures`, the HotSpot JVM does not start.

Key Features of WLDF Integration with Java Flight Recorder

WLDF integration with Java Flight Recorder provides several useful features, including having WebLogic Server events captured in the flight recording, the ability to throttle the volume of data captured, tools for downloading diagnostic image captures, and more.

The key features provided by WLDF to leverage integration with Java Flight Recorder include the following:

- WLDF diagnostic data captured in a flight recording

WLDF can be configured to generate diagnostic data about WebLogic Server events that is captured in the 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 Java Flight Recorder, and can be

adjusted to include more, or less, data for each WebLogic Server event that is generated. See [Configuring WLDF Diagnostic Volume](#).

 **Note:**

- By default, the WLDF diagnostic volume is set to `Low`.
- The WLDF diagnostic volume setting does not affect explicitly configured diagnostic modules or the built-in 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 Java Flight Recorder remains minimized, which is especially important when systems are under load.
- The time interval encompassed in the 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 Java Flight Recorder. The JFR file includes data generated by all active event producers, including WebLogic Server. An image captured using the Policies and Actions 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 [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 Java Mission Control.

Java Flight Recorder Use Cases

Java Flight Recorder helps to resolve important diagnostic issues such as diagnosing critical failure, and examining and reporting runtime data. When a critical failure occurs, the data captured by Java Flight Recorder is useful for failure analysis. Likewise, capturing data at specific time and at runtime help to diagnose data after and before a particular event.

This section summarizes the three common business cases of using the Java Flight Recorder to resolve diagnostic issues:

For more information about scenarios using Java Flight Recorder, see also About Java Flight Recorder in *Java Flight Recorder Runtime Guide*, available at the following URL:

<http://docs.oracle.com/javacomponents/index.html>

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

Diagnosing a Critical Failure — The "Black Box"

When a "catastrophic" failure occurs, the content of the Java 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 Java Flight Recorder configuration at the time of the crash. Furthermore, depending on the disk storage parameters that are set, the Java Flight Recorder data buffer might contain a certain amount of data.
- WebLogic Server events, captured by WLDF, that preceded the failure.

Java Flight Recorder 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 way, the on-disk data can 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 Java Flight Recorder configuration at the time of the crash, including the path to the data buffer file when applicable.

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 real-time diagnostics reporting, 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 Java Flight Recorder optimizes the ability to perform deep analysis of lock contention and causes of latency.

Real-Time Application Diagnostics and Reporting

It is particularly useful to examine 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 Java 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 Java 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 that is particularly useful in conjunction with Java Flight Recorder is the image action. An image action generates a diagnostic image capture in response to the triggering of a policy that is configured in a diagnostic system module. The policy monitors the server environment for one or more specific conditions, and when those conditions occur, the policy can automatically execute an image action. When Flight Recorder is enabled, the diagnostic image capture automatically includes the JFR file. The JFR file can then be extracted from the diagnostic image capture and examined immediately in Java Mission Control or stored for later analysis. An image action, used when WLDF data is captured by Java Flight Recorder, is particularly well suited for real-time diagnosis of intermittent problems.

Image action is part of the Policies and Actions system in WLDF. To set up an image action, you create one or more individual policies. A policy includes a Java EL expression to specify the event for the policy to detect. For example, the following log policy expression detects the server log message with severity level `Critical` and ID `BEA-149618`:

```
log.severityString == 'Critical' && log.messageId == 'BEA-149618'
```

Policies can monitor any of the following:

- Runtime MBean instances in the local runtime MBean server
A scheduled policy can execute an image action if runtime 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 policy can execute an image action if a specific message, severity level, or string is issued.
- Event generated by the WLDF Instrumentation component
An event policy can execute an image action if an instrumentation service generates a particular event.

See the following topics:

- [Configuring Policies and Actions](#)
- [Configuring Image Actions](#)

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

- [Obtaining the Flight Recording File](#)

- [Analyzing Java Flight Recorder Data](#)

Obtaining the Flight Recording File

The diagnostic image capture is a single Java Flight Recorder (JFR) file that contains individual images produced by different server subsystems. The JFR file is included in the diagnostic image as `FlightRecording.jfr`.

A diagnostic image capture can be generated on-demand — for example, from the WebLogic Server Administration Console, Fusion Middleware Control, WLST, or a JMX application — or it can be generated as the result of an image action. 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 [Configuring and Capturing Diagnostic Images](#). Once you have extracted the JFR file, you can view its contents in Java 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 [Example: Retrieving a JFR File from a Diagnostic Image Capture](#).

Analyzing Java Flight Recorder Data

You can extract the JFR file from the diagnostic image capture and use Java Mission Control to examine the contents of the JFR file. JFR provides graphical user interface which gives view of all the event information recorded in the JFR file.

The following sections highlight some of the capabilities of Java 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 HotSpot:

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

For complete details about the Java Mission Control interface, see *Java Mission Control User's Guide* at the following location:

<http://docs.oracle.com/javacomponents/index.html>



Note:

Flight Recorder data may include `partition-id` and `partition-name` in captured JFR events, but only the partition user may have access to the JFR data containing the information corresponding to the partition for that user. See [Monitoring and Debugging Partitions](#) in *Using Oracle WebLogic Server Multitenant*.

 **Note:**

WebLogic Server Multitenant domain partitions, resource groups, resource group templates, and virtual targets are deprecated in WebLogic Server 12.2.1.4.0 and will be removed in the next release.

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

Java Flight Recorder Graphical User Interface

Java Mission Control includes the Java Flight Recorder graphical user interface, which allows users who are running a Java Flight Recorder-compliant version of Oracle HotSpot to view JVM recordings, current recording settings, and runtime 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 Java Flight Recorder 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 4-2](#) shows an example of the Overview tab in the Events Type View.

Note the following regarding the information shown in [Figure 4-2](#):

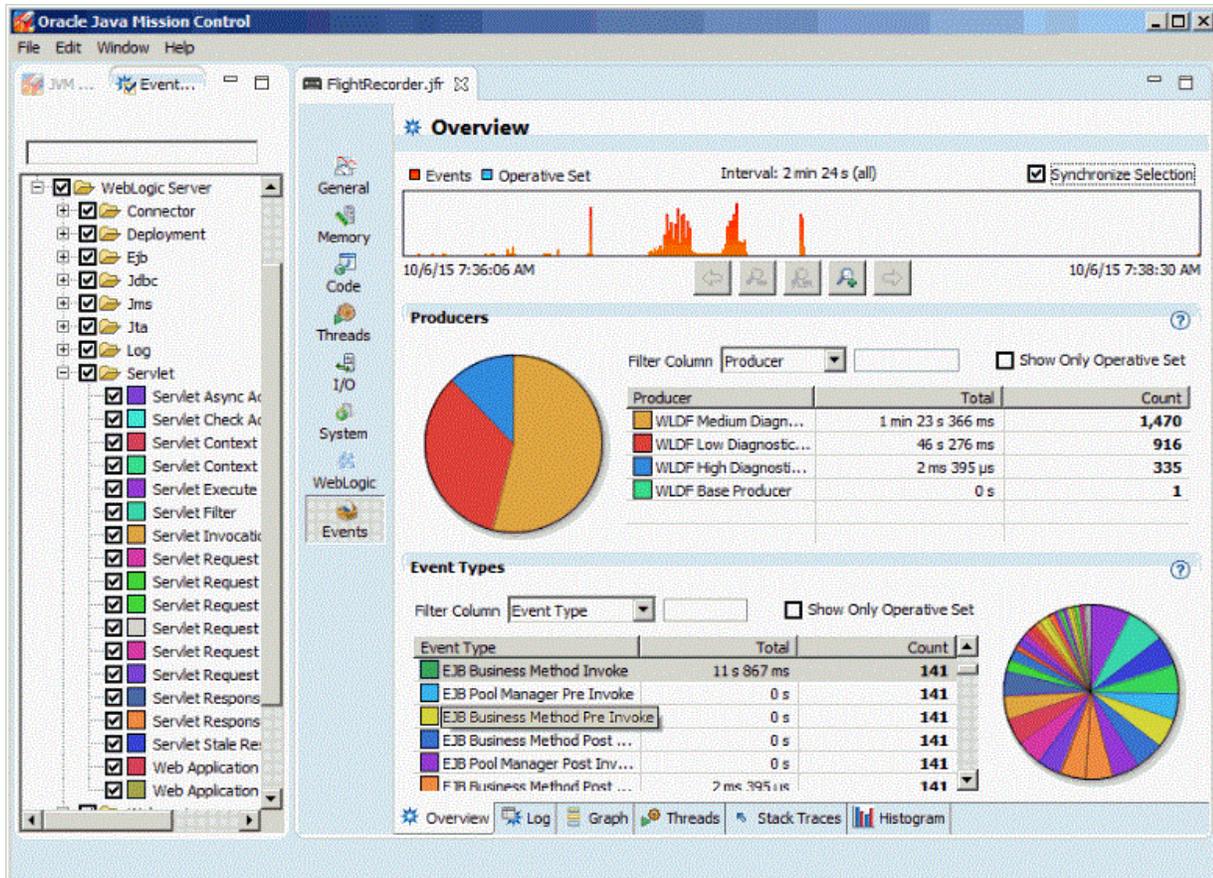
- The Events Type View is available by selecting the Events tab group icon.
- The name of the Java Flight Recorder file appears at the top of the Overview tab. Note that the Java Flight Recorder is always named `FlightRecording.jfr`, it is useful to rename it descriptively after downloading it from the 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 time line 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 Java 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 4-2 Example Overview Page of Java Flight Recorder File in Java Mission Control



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 Java Flight Recorder 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:

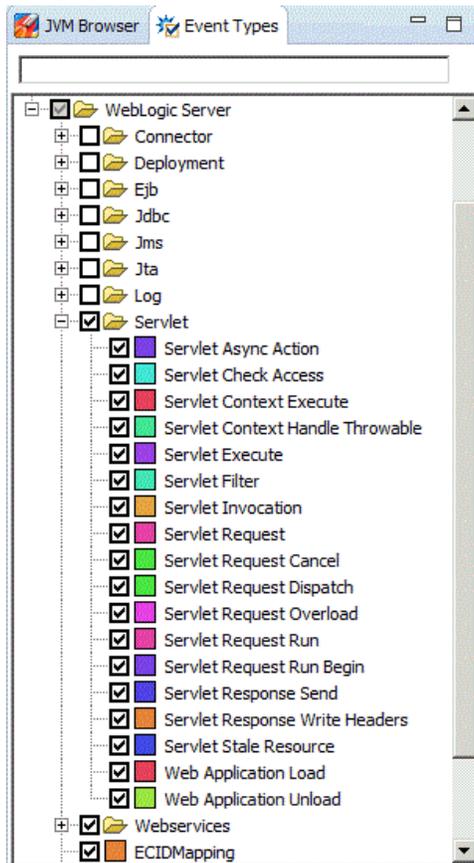
- [Displaying Event Data for a Product Subcomponent](#)
- [Viewing the Event Log to Display Details](#)
- [Tracking Execution Flow by Analyzing an Operative Set](#)
- [Expanding the Operative Set and Viewing Correlated Diagnostic Data](#)

Displaying Event Data for a Product Subcomponent

When you start Java 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 Java Flight Recorder graphical user interface is updated instantly to show information about only the selected event types.

Figure 4-3 shows the Event Types Browser with only servlet event types selected.

Figure 4-3 Event Types Browser



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 Java Flight Recorder graphical user interface. An example of the Log tab for servlet event types is shown in Figure 4-4.

Figure 4-4 Servlet Event Log

Event Log

Filter Column: Event Type Show Only Operative Set

Event Type	Start Time	End Time	Duration	Thread	Java T...
Servlet Request Run Begin	10/6/15 7:36:53.82...	10/6/15 7:36:53.82...	0 s	[ACTIVE] ExecuteThread: '1...	62
Servlet Request Run	10/6/15 7:36:53.82...	10/6/15 7:36:54.21...	385 ms 629 µs	[ACTIVE] ExecuteThread: '1...	62
Servlet Context Execute	10/6/15 7:36:53.82...	10/6/15 7:36:54.21...	385 ms 87 µs	[ACTIVE] ExecuteThread: '1...	62
Servlet Invocation	10/6/15 7:36:53.82...	10/6/15 7:36:54.21...	384 ms 671 µs	[ACTIVE] ExecuteThread: '1...	62
Servlet Filter	10/6/15 7:36:53.82...	10/6/15 7:36:54.21...	384 ms 279 µs	[ACTIVE] ExecuteThread: '1...	62
Servlet Filter	10/6/15 7:36:53.82...	10/6/15 7:36:54.21...	383 ms 430 µs	[ACTIVE] ExecuteThread: '1...	62
Servlet Execute	10/6/15 7:36:53.82...	10/6/15 7:36:54.21...	383 ms 404 µs	[ACTIVE] ExecuteThread: '1...	62
Servlet Request Dispatch	10/6/15 7:36:54.16...	10/6/15 7:36:54.20...	39 ms 975 µs	[ACTIVE] ExecuteThread: '1...	62

Event Attributes

Name	Value
Start Time	10/6/15 7:36:53.825 AM
End Time	10/6/15 7:36:54.211 AM
Duration	385 ms 629 µs
User ID	<WLS Kernel>
Return Value	
Method Name	runInternal
Class Name	weblogic.servlet.internal.ServletRequestImpl
Transaction ID	N/A
Partition Id	0
Partition Name	DOMAIN
RID	0
ECID	afcb3bce-fb65-488d-91ce-92b5a2f6f142-00000178
Servlet Name	Faces Servlet
URI	/medrec/admin/viewNewlyRegisteredPatient.action

Overview Log Graph Threads Stack 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 4-4 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
 - Partition ID and name — Note that events generated on behalf of a server or domain scope resource are tagged with a `partition-id` of 0, and the `partition-name` of DOMAIN.
 - Relationship ID (RID), which distinguishes the work done in one thread on one process, from work done by any other threads on this and other processes on behalf of the same request. See Understanding ECIDs and RIDs in Correlating Messages in *Administering Oracle Fusion Middleware*.
 - 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 [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.

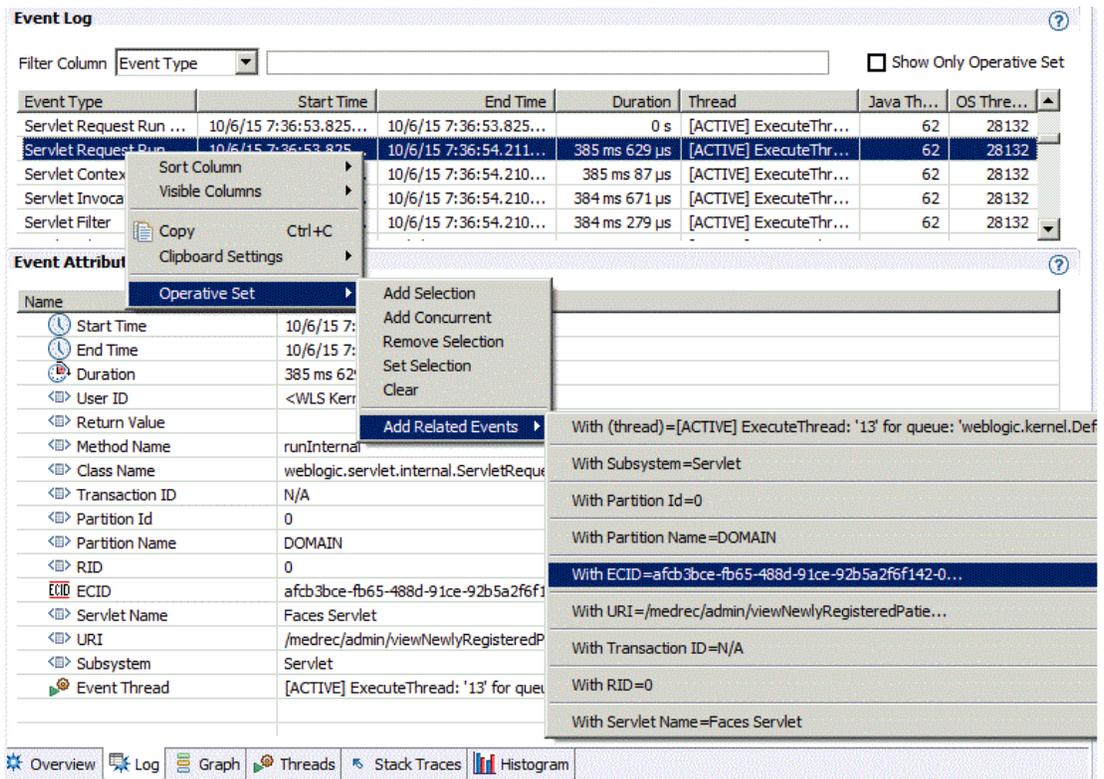
Tracking Execution Flow by Analyzing an Operative Set

The Java Flight Recorder graphical user interface in Java 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 Java 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 4-4](#). 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.)

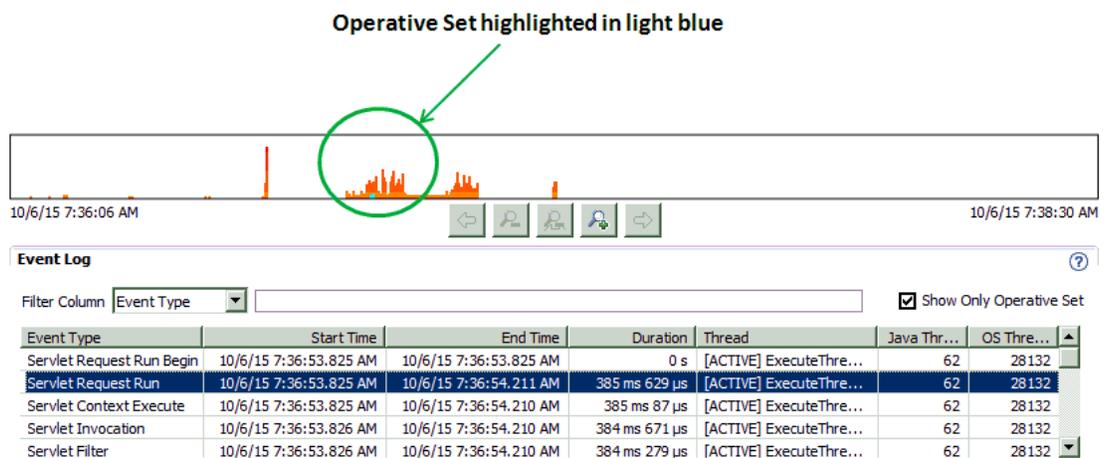
This operative set is defined by right-clicking the desired event in the Event Log, and then selecting **Operative Set > Add Related Events > With ECID=<ecid>**. See [Figure 4-5](#).

Figure 4-5 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 4-6. Note how the operative set is indicated in the range navigator.

Figure 4-6 Displaying an Operative Set



The runtime 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 4-7 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 4-7 Adding all WebLogic Server Events to Operative Set

Event Type	Start Time	End Time	Duration	Thread	Java Th...	Op: ▲
Servlet Context Execute	10/6/15 7:36:53.825...	10/6/15 7:36:54.210...	385 ms 87 µs	[ACTIVE] ExecuteThr...	62	
Servlet Invocation	10/6/15 7:36:53.826...	10/6/15 7:36:54.210...	384 ms 671 µs	[ACTIVE] ExecuteThr...	62	
Servlet Filter	10/6/15 7:36:53.826...	10/6/15 7:36:54.210...	384 ms 279 µs	[ACTIVE] ExecuteThr...	62	
Servlet Filter	10/6/15 7:36:53.827...	10/6/15 7:36:54.210...	383 ms 430 µs	[ACTIVE] ExecuteThr...	62	
Servlet Execute	10/6/15 7:36:53.827...	10/6/15 7:36:54.210...	383 ms 404 µs	[ACTIVE] ExecuteThread: '13' for queue: 'weblo	62	
EJB Business Method Invoke	10/6/15 7:36:53.828...	10/6/15 7:36:54.166...	337 ms 185 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.830...	10/6/15 7:36:53.831...	1 ms 481 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.834...	10/6/15 7:36:54.117...	283 ms 15 µs	[ACTIVE] ExecuteThr...	62	
EJB PoolManager Create	10/6/15 7:36:53.834...	10/6/15 7:36:53.839...	4 ms 395 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.840...	10/6/15 7:36:54.117...	277 ms 135 µs	[ACTIVE] ExecuteThr...	62	
EJB PoolManager Create	10/6/15 7:36:53.840...	10/6/15 7:36:53.895...	55 ms 352 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.856...	10/6/15 7:36:53.879...	22 ms 608 µs	[ACTIVE] ExecuteThr...	62	
EJB PoolManager Create	10/6/15 7:36:53.856...	10/6/15 7:36:53.861...	4 ms 495 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.862...	10/6/15 7:36:53.867...	5 ms 628 µs	[ACTIVE] ExecuteThr...	62	
EJB PoolManager Create	10/6/15 7:36:53.862...	10/6/15 7:36:53.865...	3 ms 591 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.897...	10/6/15 7:36:53.898...	1 ms 172 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.899...	10/6/15 7:36:53.906...	6 ms 545 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.899...	10/6/15 7:36:53.901...	1 ms 612 µs	[ACTIVE] ExecuteThr...	62	
EJB Business Method Invoke	10/6/15 7:36:53.913...	10/6/15 7:36:53.928...	15 ms 90 µs	[ACTIVE] ExecuteThr...	62	
EJB PoolManager Create	10/6/15 7:36:53.913...	10/6/15 7:36:53.918...	4 ms 725 µs	[ACTIVE] ExecuteThr...	62	
JDBC Transaction Is Same RM	10/6/15 7:36:54.035...	10/6/15 7:36:54.035...	0 s	[ACTIVE] ExecuteThr...	62	
JDBC Transaction Is Same RM	10/6/15 7:36:54.103...	10/6/15 7:36:54.103...	0 s	[ACTIVE] ExecuteThr...	62	
JTA Transaction Start	10/6/15 7:36:54.104...	10/6/15 7:36:54.104...	0 s	[ACTIVE] ExecuteThr...	62	
JTA Transaction Start	10/6/15 7:36:54.123...	10/6/15 7:36:54.123...	0 s	[ACTIVE] ExecuteThr...	62	
JDBC Transaction Start	10/6/15 7:36:54.124...	10/6/15 7:36:54.124...	0 s	[ACTIVE] ExecuteThr...	62	

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

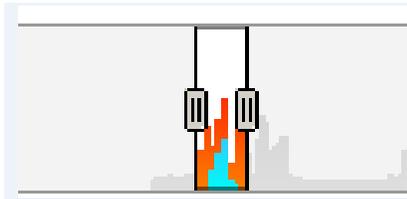
1. The servlet URI is invoked.
2. The servlet uses an EJB, which requires access to the database.
3. A JDBC connection is obtained and a transaction is started.

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 4-8](#).

Figure 4-8 Range Navigator Selection Bars



Events from additional producers, such as HotSpot, 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 4-9 shows drilling down into JDBC activity by selecting only JDBC events and JVM events. The Event Log is updated and listed in chronological order to show the JVM activity that occurred simultaneously to the flow of the JDBC events in the selected time interval.

Figure 4-9 Adding JVM Events to JDBC Event Log

Events Operative Set Interval: 2 s 847 ms (selected) Synchronize Selection

10/6/15 7:36:53 AM 10/6/15 7:36:55 AM

Show Only Operative Set

Event Type	Start Time	End Time	Duration	Thread	Java
Java Monitor Wait	10/6/15 7:36:52.998...	10/6/15 7:36:53.198...	199 ms 430 µs	weblogic.timers.TimerThread	
Java Monitor Wait	10/6/15 7:36:52.999...	10/6/15 7:36:53.298...	299 ms 820 µs	[STANDBY] ExecuteThread: '4' ...	
JDBC Transaction Start	10/6/15 7:36:53.155...	10/6/15 7:36:53.155...	0 s	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Connection Prepare	10/6/15 7:36:53.157...	10/6/15 7:36:53.166...	9 ms 238 µs	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Statement Creation	10/6/15 7:36:53.166...	10/6/15 7:36:53.166...	49 µs 370 ns	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Statement Execute Begin	10/6/15 7:36:53.167...	10/6/15 7:36:53.167...	0 s	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Statement Execute	10/6/15 7:36:53.167...	10/6/15 7:36:53.169...	1 ms 722 µs	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Connection Close	10/6/15 7:36:53.170...	10/6/15 7:36:53.170...	23 µs 257 ns	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Transaction End	10/6/15 7:36:53.177...	10/6/15 7:36:53.177...	0 s	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Transaction Commit	10/6/15 7:36:53.180...	10/6/15 7:36:53.180...	0 s	[ACTIVE] ExecuteThread: '8' fo...	
JDBC Connection Release	10/6/15 7:36:53.181...	10/6/15 7:36:53.181...	40 µs 903 ns	[ACTIVE] ExecuteThread: '8' fo...	
Java Monitor Wait	10/6/15 7:36:53.198...	10/6/15 7:36:53.645...	446 ms 810 µs	weblogic.timers.TimerThread	
Java Monitor Wait	10/6/15 7:36:53.198...	10/6/15 7:36:53.338...	139 ms 451 µs	[ACTIVE] ExecuteThread: '13' f...	
CPU Load	10/6/15 7:36:53.244...	10/6/15 7:36:53.244...	0 s	N/A	
Java Thread Statistics	10/6/15 7:36:53.244...	10/6/15 7:36:53.244...	0 s	N/A	
Class Loading Statistics	10/6/15 7:36:53.244...	10/6/15 7:36:53.244...	0 s	N/A	
Compiler Statistics	10/6/15 7:36:53.244...	10/6/15 7:36:53.244...	0 s	N/A	
Java Monitor Wait	10/6/15 7:36:53.280...	10/6/15 7:36:53.554...	273 ms 842 µs	[ACTIVE] ExecuteThread: '8' fo...	
Exception Statistics	10/6/15 7:36:53.287...	10/6/15 7:36:53.287...	0 s	JFR request timer	
Java Monitor Wait	10/6/15 7:36:53.287...	10/6/15 7:36:54.287...	1 s	JFR request timer	
Java Monitor Wait	10/6/15 7:36:53.300...	10/6/15 7:36:53.582...	281 ms 964 µs	[STANDBY] ExecuteThread: '4' ...	
Method Profiling Sample	10/6/15 7:36:53.342...	10/6/15 7:36:53.342...	0 s	[ACTIVE] ExecuteThread: '13' f...	
VM Operation	10/6/15 7:36:53.366...	10/6/15 7:36:53.414...	48 ms 92 µs	VM Thread	
Garbage Collection	10/6/15 7:36:53.366...	10/6/15 7:36:53.414...	48 ms 25 µs	N/A	
Young Garbage Collection	10/6/15 7:36:53.366...	10/6/15 7:36:53.414...	48 ms 25 µs	N/A	

Changing the Location of Temporary JFR Files

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

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

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

For more information about Java Flight Recorder configuration settings, see *Java Flight Recorder Runtime Guide* at the following location:

<http://docs.oracle.com/javacomponents/index.html>

5

Understanding WLDF Configuration

The WebLogic Diagnostics Framework (WLDF) provides several features 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.

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

- [Configuration MBeans and XML](#)
WLDF is configured using configuration MBeans (Managed Beans), and the configuration is persisted in the 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 persisted in the XML files.
- [Tools for Configuring WLDF](#)
You can configure the WLDF in several ways such as using the built-in diagnostic modules, WebLogic Administration Console, WebLogic Scripting Tool (WLST), JMX and WLDF configuration beans, and editing the XML configuration files.
- [How WLDF Configuration Is Partitioned](#)
You can use WLDF to perform diagnostics tasks for server instances, clusters, and for applications.
- [Configuring Diagnostic Image Capture and Diagnostic Archives](#)
Configure the Diagnostic Image Capture and Diagnostic Archive components in the `config.xml` file for a domain. The server configuration details are defined in the `<server-diagnostic-config >` element of the XML configuration file.
- [Configuring Diagnostic Image Capture for Java Flight Recorder](#)
The JFR file contains data for all events procedures that are enabled. When WebLogic Server is configured with a supported version of Oracle HotSpot and Java Flight Recorder is enabled, the JFR file is automatically included in the diagnostic image capture.
- [Configuring Diagnostic System Modules](#)
To configure and use the Instrumentation, Harvester, and Policies and Actions 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. The configuration of diagnostic system module is defined in a resource descriptor.
- [Configuring Diagnostic Modules for Applications](#)
- [WLDF Configuration MBeans and Their Mappings to XML Elements](#)
The set of WLDF configuration MBeans, along with the diagnostic system module beans for WLDF objects, are organized into a specific hierarchy in a WebLogic domain.

Configuration MBeans and XML

WLDF is configured using configuration MBeans (Managed Beans), and the configuration is persisted in the 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 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 [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 Using JMX for Oracle WebLogic Server*](#).

Tools for Configuring WLDF

You can configure the WLDF in several ways such as using the built-in diagnostic modules, WebLogic Administration Console, WebLogic Scripting Tool (WLST), JMX and WLDF configuration beans, and editing the XML configuration files.

Refer to the following sections for more information about the tools:

- Use the built-in diagnostic system modules, which provide a simple and easy-to-use mechanism for performing basic health and performance monitoring of a WebLogic Server instance. See [Using the Built-in Diagnostic System Modules](#).
- Use the WebLogic Server Administration Console to configure WLDF for server instances and clusters. See [Configure the WebLogic Diagnostics Framework](#) in the *Oracle WebLogic Server Administration Console Online Help*.
- Write scripts to be run in the WebLogic Scripting Tool (WLST). For specific information about using WLST with WLDF, see [WebLogic Scripting Tool Examples](#). Also see Introduction in *Understanding the WebLogic Scripting Tool* for general information about using WLST.
- Configure WLDF programmatically using JMX and the WLDF configuration MBeans. See [Configuring and Using WLDF Programmatically](#) for specific information about programming WLDF. See *MBean Reference for Oracle WebLogic Server* 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 WebLogic Server 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.

How WLDF Configuration Is Partitioned

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

- [Server-Level Configuration](#)
- [Application-Level Configuration](#)

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 [Configuring Diagnostic Image Capture and Diagnostic Archives](#).

You configure the following WLDF components as the parts of one or more diagnostic system modules 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)
- Policies and Actions
- Instrumentation

See [Configuring Diagnostic System Modules](#).

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 [Configuring Diagnostic Modules for Applications](#).

Configuring Diagnostic Image Capture and Diagnostic Archives

Configure the Diagnostic Image Capture and Diagnostic Archive components in the `config.xml` file for a domain. The server configuration details are defined in the `<server-diagnostic-config>` element of the XML configuration file.

The `<server-diagnostic-config>` element is a child of the `<server>` element in a domain, as shown in [Example 5-1](#).

Example 5-1 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. -->
</domain>
```

Note:

If WebLogic Server is configured with Oracle HotSpot, and Java Flight Recorder is enabled, the diagnostic image capture can optionally include a Java Flight Recorder file, also called a JFR file, that includes WebLogic Server events. The JFR file can then be viewed in Java Mission Control. See [Using WLDF with Java Flight Recorder](#).

See the following topics:

- [Configuring and Capturing Diagnostic Images](#)
- [Configuring Diagnostic Archives](#)

Configuring Diagnostic Image Capture for Java Flight Recorder

The JFR file contains data for all events procedures that are enabled. When WebLogic Server is configured with a supported version of Oracle HotSpot and Java Flight

Recorder is enabled, the JFR file is automatically included in the diagnostic image capture.

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:**

Note the following:

- If WebLogic Server is configured with Oracle HotSpot, Java Flight Recorder is disabled by default unless HotSpot is started using the JVM parameters described in [Using Java Flight Recorder with Oracle HotSpot](#).
- By default, the WLDF diagnostic volume is set to `Low`.
- For the most current information about configurations supported in this release of WebLogic Server, including HotSpot support, see Oracle Fusion Middleware Supported System Configurations on the Oracle Technology Network.

To include WebLogic Server event data in the JFR file:

1. Ensure that WebLogic Server is configured with Oracle HotSpot, which installed separately from WebLogic Server.

See Planning the Oracle WebLogic Server Installation in *Installing and Configuring Oracle WebLogic Server and Coherence*.

2. Ensure that Java Flight Recorder is enabled.

In a default installation of Oracle HotSpot with WebLogic Server, Java Flight Recorder is disabled. For information about enabling Java Flight Recorder with HotSpot and WebLogic Server, see [Using Java Flight Recorder with Oracle HotSpot](#).

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.

See [Configure WLDF diagnostic volume](#) in *Oracle WebLogic Server Administration Console Online Help*.

 **Note:**

If the WLDF diagnostic volume is set to `Off`, and Java 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.

Configuring Diagnostic System Modules

To configure and use the Instrumentation, Harvester, and Policies and Actions 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. The configuration of diagnostic system module is defined in a resource descriptor.

The diagnostic system module created at the server level contains the configurations for the components. When creating a diagnostic system module, note the following:

- Diagnostic 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.
- You can target multiple diagnostic system modules to any given server or cluster.
- [WLDF Runtime Control](#) allows you to dynamically enable or disable a diagnostic system module without changing the domain configuration.
- Runtime control also allows you to deploy, activate, deactivate, and undeploy a diagnostic system module on-the-fly that is not defined in the domain configuration.

The following sections described the configuration of diagnostic system modules:

- [About the Resource Descriptor](#)
- [WLDF Runtime Control](#)
- [Creating a Diagnostic System Module Based on a Configured Resource Descriptor](#)
- [Creating a Diagnostic System Module Based on an External Resource Descriptor](#)
- [Targeting a Diagnostic System Module to a Server or Cluster](#)
- [Dynamically Activating or Deactivating Diagnostic System Modules](#)
- [Using WLST to Activate and Deactivate Diagnostic System Modules](#)
- [More Information About Configuring Diagnostic System Modules](#)

About the Resource Descriptor

A diagnostic system module has a corresponding resource descriptor that defines the diagnostic module's configuration. A resource descriptor can be either *configured* or *external*:

- A *configured* resource descriptor is one that is defined as part of the domain configuration, and exists as a file in the `DOMAIN_HOME/config/diagnostics` directory. A configured resource descriptor is referenced by the domain `config.xml` file, and the corresponding diagnostic system module:
 - Is persisted in the domain configuration.
 - Is available to all servers and clusters in the domain.
 - Can be targeted to a server or cluster through the domain configuration.
 - Can be activated or deactivated dynamically using Runtime Control, regardless of whether it is explicitly targeted to a server or cluster.

Any dynamic changes made to the activation state of the diagnostic system module are not persisted across server restarts.

- An *external* resource descriptor is one that is *not* referenced by the domain `config.xml` file; that is, it is defined outside the domain configuration. The

diagnostic system module that is configured by an external resource descriptor may be deployed and activated on a server using Runtime Control. However, this diagnostic system module:

- Is not persisted in the domain configuration (that is, it is not referenced by the domain `config.xml` file).
- Can be deployed, activated, and deactivated only dynamically.
- Cannot have its deployment and activation state persisted in the domain configuration.
- Remains in memory only until the server or cluster on which it is activated is shut down.
- Cannot be automatically available on server restart.

An external resource descriptor may exist in a file located outside the `DOMAIN_HOME/config/diagnostics` directory, or may be passed as a String object using the WLDF Runtime Control API (see [Creating a Diagnostic System Module Based on an External Resource Descriptor](#)).

 **Note:**

The configuration of a diagnostic module conforms to the `diagnostics.xsd` schema, available at <http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/weblogic-diagnostics.xsd>.

Except for the name and list of targets for the diagnostic system module, all configuration information for a diagnostic system module is contained in its resource descriptor file.

[Example 5-2](#) shows portions of the descriptor file for a diagnostic system module named `myDiagnosticModule`.

Example 5-2 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/2.0/
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 policies and one or more
      actions-->
  </watch-notification>
</wldf-resource>
```

WLDF Runtime Control

WLDF Runtime Control allows you to control the activation or deactivation of diagnostics system modules dynamically at run time without making a change to the domain configuration. This allows you to perform specific, targeted diagnostic analysis tasks, and optionally of limited duration, without interfering with the operation of the server instances themselves.

You can use Runtime Control to do the following:

- Dynamically activate and deactivate diagnostic system modules that are persisted in the domain configuration without restarting the servers or clusters to which they are targeted.
- Dynamically deploy, activate, deactivate, and undeploy diagnostic system modules that are configured by an external resource descriptor.

Note:

Note the following:

- Changes applied to diagnostic system modules using Runtime Control, whether defined by configured or external resource descriptors, are not persisted. When a server instance is restarted, that server returns to its configured state, and any changes prior to that restart that were made using Runtime Control are lost.
- If you use the Runtime Control to activate a diagnostic system module that is based on an external resource descriptor (see [Creating a Diagnostic System Module Based on an External Resource Descriptor](#)), the diagnostic resource name that you specify in the `createSystemResourceControl()` command to create that diagnostic system module is used as the WLDF Module name in Harvester records in the archive.

Creating a Diagnostic System Module Based on a Configured Resource Descriptor

You create a diagnostic system module based on a configured resource descriptor using either the WebLogic Server Administration Console or the WebLogic Scripting Tool (WLST). It is created as a `WLDFResourceBean`, and the configuration is persisted in a resource descriptor file named `DIAG_MODULE.xml`, where `DIAG_MODULE` is the name of the diagnostic system 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_HOME\config\diagnostics` directory, 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*.

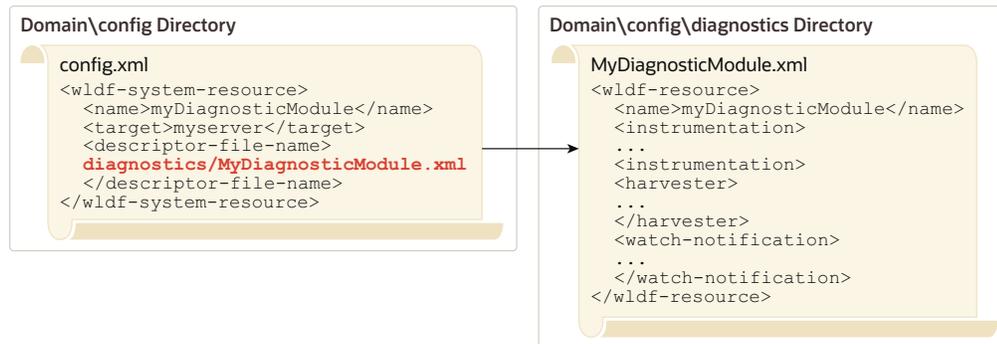
The `config.xml` file can contain references to multiple diagnostic system modules, in one or more `<wldf-system-resource>` elements. The `<wldf-system-resource>` element includes the name of the diagnostic system module file and the list of servers and clusters to which the module is targeted.

For example, [Example 5-3](#) shows a `config.xml` file with a module named `myDiagnosticModule` targeted to the server `myserver` and another module named `newDiagnosticMod` targeted to servers `myserver` and `ManagedServer2`. Note that `myDiagnosticModule` and `newDiagnosticMod` are both targeted to `myserver`.

Example 5-3 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>myserver,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 5-1](#).

Figure 5-1 Relationship of config.xml to System Descriptor File

For instructions on creating a diagnostic system module that is persisted in the domain, see [Create diagnostic system modules](#) in the *Oracle WebLogic Server Administration Console Online Help*.

Creating a Diagnostic System Module Based on an External Resource Descriptor

WLDF provides the following API that you can use to pass an external resource descriptor and create a diagnostic system module on-the-fly. You can use this API to dynamically create and activate a diagnostic system module for a server, but neither its deployment nor activation state is persisted when the servers or clusters on which it was activated are rebooted. This API is provided by the following MBeans:

- `weblogic.management.runtime.WLDFControlRuntimeMBean`
- `weblogic.management.runtime.WLDFSystemResourceControlRuntimeMBean`

Using this API, you can pass the resource descriptor as a String object on-the-fly. For ease-of-use, WLDF also provides the following WLST commands, which you can use with a resource descriptor file that exists externally to the domain configuration:

- `createSystemResourceControl()` — Creates (deploys) a diagnostics system module on-the-fly using a specified descriptor file.
- `destroySystemResourceControl()` — Destroys (undeploys) a diagnostics system module previously created on-the-fly.

Externally configured diagnostic system modules that are deployed and activated in a server or cluster are automatically destroyed when that server or cluster is shut down.

If you activate a diagnostic system module that is based on an external resource descriptor, the diagnostic resource name that you specify in the `createSystemResourceControl` command is used as the module name. For example, this is the name that appears in the **WLDF Module** column when displaying the contents of the Harvester archive in the WebLogic Server Administration Console. For more information about the `createSystemResourceControl` command, see [Diagnostics Commands in WLST Command Reference for WebLogic Server](#).

For an example of using WLST to create, activate, and destroy a diagnostic system module that is based on an external resource descriptor, see [Using WLST to Activate and Deactivate Diagnostic System Modules](#).

Targeting a Diagnostic System Module to a Server or Cluster

A diagnostic system module can be targeted by the domain `config.xml` file to zero, one, or more servers or clusters. In addition, a given server can have multiple modules targeted to it simultaneously. Typically you create multiple modules that monitor different aspects of your system. You can then choose which modules 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 instances 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.

For information about how to use the WebLogic Server Administration Console to target a diagnostic system module that is persisted in the domain configuration, see [Target and untarget diagnostic system modules](#) in *Oracle WebLogic Server Administration Console Online Help*.



Note:

You cannot use the WebLogic Server Administration Console to target diagnostic system modules that are configured by an external descriptor. However, you can use WLST as described in [Using WLST to Activate and Deactivate Diagnostic System Modules](#).

Dynamically Activating or Deactivating Diagnostic System Modules

After you configure a diagnostic system module, you can activate or deactivate it without making a configuration change or rebooting the server instance to which it is targeted. This capability gives you control over the operative state of diagnostic system modules without restarting the targeted server or cluster instance or making a change to the domain configuration.

Because the domain configuration and all resource files are replicated to all servers in the domain, all configured WLDF resources are available for dynamic activation and deactivation on all servers in the domain. Note that if you dynamically activate or deactivate a diagnostics system module, and restart the targeted server, the module's activation state is reverted to whatever is configured in the domain.

For information about using this capability in the WebLogic Server Administration Console for diagnostic system modules that are persisted in the domain configuration, see [Dynamically activate or deactivate a diagnostic system module](#) in *Oracle WebLogic Server Administration Console Online Help*. (Note that you cannot use the WebLogic Server Administration Console to dynamically activate or deactivate diagnostic system modules that are configured by an external descriptor.)

You can also use WLST to dynamically activate or deactivate diagnostic system modules, including those configured by an external descriptor, as described in [Using WLST to Activate and Deactivate Diagnostic System Modules](#).

Using WLST to Activate and Deactivate Diagnostic System Modules

You can also use WLST to dynamically activate or deactivate a diagnostic system module. This capability is provided by the WLST commands listed and described in [Table 5-1](#):

Table 5-1 WLST Commands to Dynamically Activate and Deactivate Diagnostic Modules

Command	Summary
<code>enableSystemResource</code>	Enables a diagnostic system module on a WebLogic Server instance.
<code>disableSystemResource</code>	Disables a diagnostic system module on a WebLogic Server instance.
<code>createSystemResourceControl</code>	Creates a diagnostics system module from an external diagnostic descriptor file. Note that the diagnostics system module remains in memory only until the server is shut down and is not deployed the next time the server is restarted.
<code>destroySystemResourceControl</code>	Destroys, or undeploys, a diagnostics system module configured in an external diagnostic descriptor without changing the domain configuration.
<code>listSystemResourceControls</code>	Lists the diagnostic system modules currently configured on a WebLogic Server instance.

For complete details about these WLST commands, see *Diagnostics Commands in WLST Command Reference for WebLogic Server*.

- [Example](#)

Example

This section gives an example of the steps for using WLST to dynamically activate and deactivate the following diagnostic system modules:

- `Module-0`, configured in the domain and defined by the resource descriptor file `Module-0-3905.xml` located in the `DOMAIN_HOME/config/diagnostics` directory
- `Module-1`, configured in the domain and defined by the resource descriptor file `Module-0-3905.xml` located in the `DOMAIN_HOME/config/diagnostics` directory
- `External-1`, not a part of the domain configuration, but defined by the external resource descriptor `external-wldf`. This external resource descriptor is configured in the file `external-wldf.xml`, which is external to the domain configuration.

These examples assume the following has been set up:

- The domain `config.xml` file references two diagnostic system modules that are part of the domain configuration, as follows:

```
<wldf-system-resource>
  <name>Module-0</name>
  <descriptor-file-name>diagnostics/Module-0-3905.xml</descriptor-file-name>
```

```

    <description></description>
  </wldf-system-resource>
<wldf-system-resource>
  <name>Module-1</name>
  <descriptor-file-name>diagnostics/Module-1-3904.xml</descriptor-file-name>
  <description></description>
</wldf-system-resource>

```

- The server name shown in these examples is `myserver`.
- The external descriptor file `external-wldf.xml` is located in the domain's root directory, `wl_domain`. It contains the following lines for configuring the diagnostic system module named `External-1`:

```

<?xml version='1.0' encoding='UTF-8'?>
<wldf-resource xmlns="http://xmlns.oracle.com/weblogic/weblogic-diagnostics"
xmlns:sec="http://xmlns.oracle.com/weblogic/security"
xmlns:wls="http://xmlns.oracle.com/weblogic/security/wls"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-diagnostics
http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/weblogic-diagnostics.xsd">
  <name>External-1</name>
  <harvester>
    <enabled>true</enabled>
    <sample-period>10000</sample-period>
    <harvested-type>
      <name>weblogic.management.runtime.ServerRuntimeMBean</name>
      <harvested-attribute>OverallHealthState.ReasonCodeSummary</harvested-attribute>
      <harvested-attribute>OverallHealthState.State</harvested-attribute>
      <namespace>ServerRuntime</namespace>
    </harvested-type>
  </harvester>
</wldf-resource>

```

Step 1: List Diagnostic System Modules

The following WLST command, shown in **bold**, lists the diagnostic system modules that are currently configured:

```

wls:/wl_domain/Server1> listSystemResourceControls()
External      Enabled      Name
-----      -
false         false       Module-0
false         false       Module-1

```

The preceding command shows that `Module-0` and `Module-1` are configured in the domain (that is, they are referenced from `config.xml` and are not configured by external resource descriptors), but that they have not been activated.

Step 2: Activate Module-0

The following WLST command activates `Module-0`:

```

wls:/mydomain/serverConfig> enableSystemResource('Module-0')

```

You can also supply a server name to all of the WLDF system resource runtime control functions. If you do not specify a server name, the `enableSystemResource()` command defaults to the server instance to which WLST is currently connected. (However, by default, all configured WLDF system resources are available for runtime control operations on all servers in the domain.)

```
wls:/mydomain/serverConfig> enableSystemResource('Module-0', Server='myserver')
```

Step 3: Verify that Module-0 is Activated

The following WLST command shows that Module-0 is now activated:

```
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         true        Module-0
false         false       Module-1
```

Step 4: Activate Module-1

The following WLST commands activate Module-1 and verify the activation state of all diagnostic system modules:

```
wls:/mydomain/serverConfig> enableSystemResource('Module-1', Server='myserver')
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         true        Module-0
false         true        Module-1
```

Step 5: Deactivate Configured Diagnostic Modules

The following WLST commands deactivate all diagnostic system modules that are configured in the domain and verify their state:

```
wls:/mydomain/serverConfig> disableSystemResource('Module-0')
wls:/mydomain/serverConfig> disableSystemResource("Module-1")
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         false       Module-0
false         false       Module-1
```

Step 6: Create a Diagnostic System Module from an External Resource Descriptor File

The external resource descriptor needs to be accessible by the WLST client. The following WLST command creates and deploys the diagnostic system module External-1 from the external resource descriptor in the file `external-wldf.xml`, and verifies its activation state. ()

```
wls:/mydomain/serverConfig> createSystemResourceControl('external-wldf',
'external-wldf.xml')
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         false       Module-0
true          false       external-wldf
false         false       Module-1
```

Note that the External column identifies External-1 as being configured by an external resource descriptor.

Step 7: Activate External-1

Because the `createSystemResourceControl()` command only deploys the diagnostic system module, the following WLST command activates it. The subsequent command verifies the diagnostic system module's activation state.

```
wls:/mydomain/serverConfig> enableSystemResource("external-wldf")
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         false       Module-0
true          true        external-wldf
false         false       Module-1
```

Step 8: Deactivate External-1

The following WLST commands deactivate `External-1` and verify its deactivation status:

```
wls:/mydomain/serverConfig> disableSystemResource("external-wldf")
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         false       Module-0
true          false      external-wldf
false         false       Module-1
```

Step 9: Destroy External-1

The following WLST command destroys the diagnostic system module that is configured by an external resource descriptor:

```
wls:/mydomain/serverConfig> destroySystemResourceControl("external-wldf")
```

Step 10: Verify Original State of Configured Diagnostic Modules

The following WLST command verifies that the domain's configuration is reverted to its original state, showing only the two diagnostic system modules whose configuration is persisted in the `config.xml` file:

```
wls:/mydomain/serverConfig> listSystemResourceControls()
External      Enabled      Name
-----
false         false       Module-0
false         false       Module-1
```

More Information About Configuring Diagnostic System Modules

See the following sections for detailed instructions about configuring WLDf system modules:

- [Configuring the Harvester for Metric Collection](#)
- [Configuring Policies and Actions](#)
- [Configuring Instrumentation](#)
- [Configuring the DyelInjection Monitor to Manage Diagnostic Contexts](#)

Configuring Diagnostic Modules for Applications

WLDF supports the ability to configure instrumentation of an application by means of a diagnostic application module. A diagnostic application module is similar to a diagnostic system module, with the exception that you configure it in an XML descriptor file that you package with the application archive file. A diagnostic application module deployed this way is available only to the application in which that module is enclosed. This ensures that the application can be reliably deployed into new environments with access to all required resources in the diagnostic module. 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, `C:\Oracle\Middleware\Oracle_Home\user_projects\applications\medrec\dist\standalone\exploded\medrec\META-INF\weblogic-diagnostics.xml`.



Note:

The DyelInjection 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 [Configuring the DyelInjection Monitor to Manage Diagnostic Contexts](#).

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

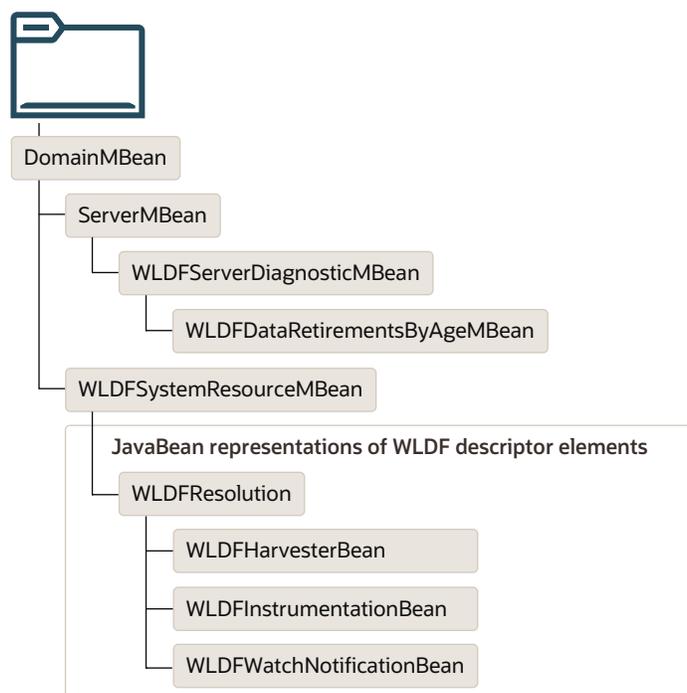
- [Configuring Application-Scoped Instrumentation](#)
- [Deploying WLDF Application Modules](#)

WLDF Configuration MBeans and Their Mappings to XML Elements

The set of WLDF configuration MBeans, along with the diagnostic system module beans for WLDF objects, are organized into a specific hierarchy in a WebLogic domain.

[Figure 5-2](#) shows the hierarchy of the WLDF configuration MBeans and the diagnostic system module beans for WLDF objects in a WebLogic Server domain.

Figure 5-2 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 [Configuring the DyelInjection 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.
- `WLDFDataRetirementByAgeMBean` specifies how data retirement for a WLDF archive is performed based on the age of records in that archive.
- `WLDFSystemResourceMBean` contains the name of a descriptor file for a diagnostic module in the `DOMAIN_HOME/config/diagnostics` directory and the names of one or more the target servers on 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.

- 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 5-1](#) and [Example 5-2](#).) 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.
 - Policies and Actions: 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.

6

Configuring and Capturing Diagnostic Images

You can use the Diagnostic Image Capture component of the WebLogic Diagnostics Framework (WLDF) to create a diagnostic snapshot or dump of a server's internal runtime state at the time of the capture. The captured information is useful for analyzing the cause of a server failure. If WebLogic Server is configured with Oracle HotSpot, and Java Flight Recorder is enabled, the diagnostic image capture includes WebLogic Server diagnostic data that can be viewed in Java Mission Control. See [Using Java Flight Recorder with Oracle HotSpot](#).

- [How to Initiate Image Captures](#)
The Diagnostic Image Capture component gathers the most common sources of key server state used in diagnosing problems, and packages that state into a single file or diagnostic image. You can initiate the image capture of a server using the WebLogic Administration Console or WLST command.
- [Configuring Diagnostic Image Captures](#)
When you initiate the capture of diagnostic image, you can configure the capture details such as the destination location of the image capture, WLDF diagnostic volume, and how often an image must be captured during failures.
- [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.
- [Content of the Captured Image File](#)
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.

How to Initiate Image Captures

The Diagnostic Image Capture component gathers the most common sources of key server state used in diagnosing problems, and packages that state into a single file or diagnostic image. You can initiate the image capture of a server using the WebLogic Administration Console or WLST command.

A diagnostic image capture can be initiated by:

- A configured policy. See [Configuring Actions](#).
- A request initiated by a user in the WebLogic Server 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 6-1](#).
- WLST command

Configuring Diagnostic Image Captures

When you initiate the capture of diagnostic image, you can configure the capture details such as the destination location of the image capture, WLDF diagnostic volume, and how often an image must be captured during failures.

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

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

The following example shows a simple policy expression that returns `true` when the value of `HeapFreePercent` attribute of `JVMRuntimeMBean` is less than 20:

```
wls.runtime.serverRuntime.JVMRuntime.heapFreePercent < 20
```

See [Configuring Scheduled Policies](#) and [Configuring Image Actions](#). Also, see [Configure Policies and Actions](#) in the *Oracle WebLogic Server Administration Console Online Help*.

- [Configuring WLDF Diagnostic Volume](#)
- [WLST Commands for Generating an Image Capture](#)

Configuring WLDF Diagnostic Volume

If WebLogic Server is configured with Oracle HotSpot, and the Java Flight Recorder is enabled, the Java 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 Java Mission Control. If Java Flight Recorder is not enabled, or if WebLogic Server is configured with a different JVM, the Java Flight Recorder data is not captured in the diagnostics image capture.



Note:

When WebLogic Server is configured with HotSpot, by default Java Flight Recorder is disabled. For information about how to enable it, see [Using Java Flight Recorder with Oracle HotSpot](#).

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

Volume Setting	Description
Off	Disables the collection of data in the Java Flight Recorder diagnostic image.
Low	Enabled by default. For information about data that is collected, see Low Volume Setting .
Medium	Captures a moderate amount of data. See Medium Volume Setting .
High	Captures in-depth data. See High Volume Setting .

 **Note:**

The specific set of events for which diagnostic data is collected using the diagnostic volume settings is subject to change in future releases of WebLogic Server.

For information about how to set the diagnostic volume, see [Configure WLDF diagnostics volume](#) in the *Oracle WebLogic Server Administration Console Online Help*. For an example using WLST, see [Example: Setting the WLDF Diagnostic Volume](#).

- [Low Volume Setting](#)
- [Medium Volume Setting](#)
- [High Volume Setting](#)

Low Volume Setting

The `Low` diagnostic volume setting is enabled by default. With this setting, basic information is generated and captured, and log messages with the "emergency", "alert", or "critical" levels are recorded.

In the current release of WebLogic Server, the following events are captured at the `Low` setting:

```
ThrottleInformation
WLDF Logging Snapshot
WLDF LogRecord Snapshot
WLDF WLLogRecord Snapshot
Connector Activate Endpoint
Connector Deactivate Endpoint
Connector Inbound Transaction Rollback
Connector Outbound Connection Closed
Connector Outbound Connection Error
Connector Outbound Destroy Connection
Connector Outbound Register Resource
Connector Outbound Release Connection
Connector Outbound Reserve Connection
Connector Outbound Transaction Rollback
Connector Outbound Unregister Resource
Deployment Complete
Deployment Do Cancel
Deployment Do Prepare
Deployment Operation
EJB Business Method Invoke
```

EJB Business Method Post Invoke
EJB Business Method Pre Invoke
JDBC Connection Rollback
JDBC Statement Execute
JDBC Statement Execute Begin
JDBC Transaction Rollback
Servlet Invocation
Servlet Request Run
Servlet Request Run Begin
Web Application Load
Web Application Unload
Webservices JAXRPC Client Request
Webservices JAXRPC Client Response
Webservices JAXRPC Dispatch
Webservices JAXRPC Request
Webservices JAXRPC Response
Webservices JAXWS Endpoint
Webservices JAXWS Request
Webservices JAXWS Resource

Medium Volume Setting

With the `Medium` diagnostic volume setting, additional information is captured, and messages with the "error" level and above are recorded. For example, User IDs are captured by the `Medium` and `High` volume settings (capturing them imposes a performance overhead not appropriate for the `Low` setting).

In the current release of WebLogic Server, the following events are captured at the `Medium` setting, in addition to those captured at the `Low` setting:

Connector Inbound Transaction Commit
Connector Inbound Transaction Start
Connector Outbound Transaction Commit
Connector Outbound Transaction Start
EJB Home Create
EJB Home Remove
EJB PoolManager Create
EJB Pool Manager Post Invoke
EJB Pool Manager Pre Invoke
JDBC Connection Close
JDBC Connection Commit
JDBC Connection Create Statement
JDBC Connection Get Vendor Connection
JDBC Connection Prepare
JDBC Connection Release
JDBC Connection Reserve
JDBC Data Source Get Connection
JDBC Driver Connect
JDBC Statement Creation
Servlet Execute
Servlet Request Dispatch
Servlet Request
Servlet Filter
Servlet Async Action
Servlet Context Execute
Servlet Response Write Headers
Servlet Response Send
Servlet Stale Resource
Servlet Check Access
JMS BE Consumer Log

High Volume Setting

With the `High` diagnostic volume setting, in-depth information is captured, and messages with the "error" level and above are recorded. Stack traces are also captured with the `High` setting, but only for events for which a stack trace add value (for example, stack traces where application code would normally be visible are generated, but stack traces that only show internal code and that do not vary at all are not generated).

In the current release of WebLogic Server, the following events are captured at the `High` setting in addition to those captured at the `Medium` setting:

```
EJB Database Access
EJB Business Method Post Invoke Cleanup
EJB Pool Manager Remove
EJB Replicated Session Manager
EJB Timer Manager
JDBC Transaction Commit
JDBC Transaction End
JDBC Transaction Get XA Resource
JDBC Transaction Is Same RM
JDBC Transaction Prepare
JDBC Transaction Start
JTA Transaction Commit
JTA Transaction End
JTA Transaction Prepared
JTA Transaction Prepare
JTA Transaction Start
Servlet Request Overload
Servlet Request Cancel
Servlet Context Handle Throwable
```

WLST Commands for Generating an Image Capture

[Example 6-1](#) shows an example of WLST commands for generating an image capture.

Example 6-1 Sample WLST Commands for Generating a Diagnostic Image

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

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.

In the `config.xml` file, the image capture is described under the `<server-diagnostic-config>` subelement of the `<server>` element for the server, as shown in [Example 6-2](#):

Example 6-2 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.

Content of the Captured Image File

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.

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

If WebLogic Server is configured with Oracle HotSpot, and Java Flight Recorder is enabled, the diagnostic image capture includes a Java Flight Recorder image, `FlightRecording.jfr`, that can be viewed in Java Mission Control. The contents of the Java Flight Recorder image contains all available data from the Java Flight Recorder, and the volume of data produced by WLDF depends on the diagnostics volume setting. When Java Flight Recorder is enabled, data is always provided by the JVM, 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 Java Flight Recorder image as well.

 **Note:**

Note the following:

- 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.
- If a non-WebLogic event producer in the WebLogic Server environment, such as DMS, has configured Java Flight Recorder to record data, the WLDF diagnostic image capture includes a Java Flight Recorder image file with the recorded data even if the WLDF diagnostics volume is set to `Off`.
- When WebLogic Server is configured with HotSpot, Java Flight Recorder is not enabled by default. For information about how to enable it, see [Using Java Flight Recorder with Oracle HotSpot](#).

- [Data Included in the Diagnostics Image Capture File](#)
- [WLST Online Commands for Downloading Diagnostics Image Captures](#)

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_NAME\logs\diagnostic_images`. Each image instance has a unique name, as follows:

```
diagnostic_image_DOMAIN_SERVER_YYYY_MM_DD_HH_MM_SS.zip
```

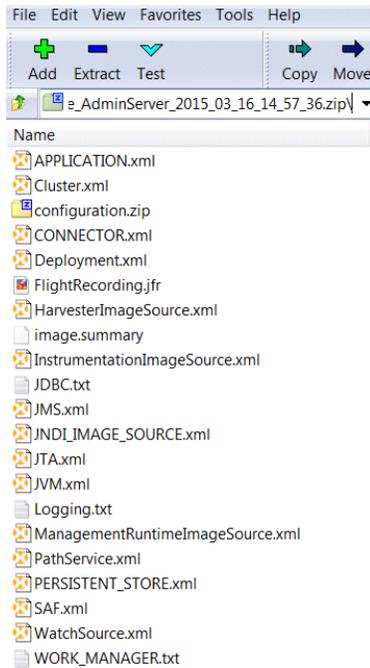
The contents of the file include at least the following information:

- 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
- WebLogic Server version including patch and build number information

If WLDF is configured with Oracle HotSpot, as described in [Configuring Diagnostic Image Capture for Java Flight Recorder](#), the image also contains the Java Flight Recorder file, `FlightRecording.jfr`. The JFR file can be extracted as described in [WLST Online Commands for Downloading Diagnostics Image Captures](#), and viewed in Java Mission Control. See [Analyzing Java Flight Recorder Data](#).

[Figure 6-1](#) 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 6-1 Contents of an Image File



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:

Table 6-1 WLST Commands for Downloading Image Captures

Command	Summary
captureAndSaveDiagnosticImage	Captures a diagnostic image and downloads it locally.
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 Java Flight Recorder diagnostics data for viewing in Java Mission Control.

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

7

Configuring Diagnostic Archives

The Archive component captures and persists all data events, log records, and metrics collected by the WebLogic Diagnostics Framework (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), or in off-line mode using the WebLogic Scripting Tool (WLST).

This chapter explains how to configure the Archive, and also how to configure WLDF to archive diagnostic data to a file store or a Java Database Connectivity (JDBC) data source:

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

- [Configuring the Archive](#)
You can 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.
- [Configuring a File-Based Store](#)
WLDF supports the ability to use a file-based store for the Archive. If you choose the use of a file-based store, the only configuration option you must set is the location of the directory where the store is to be maintained. The default directory is `DOMAIN_HOME/servers/SERVER_NAME/data/store/diagnostics`.
- [Configuring a JDBC-Based Store](#)
- [Retiring Data from the Archives](#)
To maintain the archived data, you must delete the old archived data periodically. WLDF includes a configuration-based data retirement feature for doing this. The data can be deleted based on the size of the data and time period when it was created.

Configuring the Archive

You can 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 7-1](#) and [Example 7-7](#).

Note:

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

Configuring a File-Based Store

WLDF supports the ability to use a file-based store for the Archive. If you choose the use of a file-based store, the only configuration option you must set is the location of the directory where the store is to be maintained. The default directory is `DOMAIN_HOME/servers/SERVER_NAME/data/store/diagnostics`.

When you save to a file-based store, WLDF uses the WebLogic Server persistent store. See *Using the WebLogic Persistent Store* in *Administering the WebLogic Persistent Store*.

An example configuration for a file-based store is shown in [Example 7-1](#).

Example 7-1 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>
```

Configuring a JDBC-Based Store

WLDF supports the ability to create the Archive in 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 WebLogic Server Administration Console, see [Configure database connectivity](#) in *Oracle WebLogic Server Administration Console Online Help*. For additional information about JDBC configuration, see *Administering JDBC Data Sources for Oracle WebLogic Server*.



Note:

If you install multiple WLDF schemas in the same database, you need to provide a way to distinguish among them when accessing the diagnostic archives. You can do this when you configure the diagnostic archive for a server instance by specifying the schema name to use for accessing JDBC-based archive tables in that database. See [Configuring JDBC Resources for WLDF](#).

- [Creating WLDF Tables in the Database](#)
- [Configuring JDBC Resources for WLDF](#)

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.

- [Apache Derby](#)
- [Oracle Database](#)
- [MySQL](#)

Apache Derby

[Example 7-2](#) shows the DDL that you can use to create the `wls_events` and `wls_hvst` tables in Apache Derby.

Example 7-2 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(250) 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),
  WLDFFMODULE VARCHAR(250) default NULL
);

COMMIT;
```

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

Oracle Database

[Example 7-3](#) shows the DDL that you can use to create the `wls_events` table in Oracle database.

Example 7-3 DDL Definition of the `wls_events` Table for Oracle Database

```
SET SERVEROUTPUT ON;

DECLARE
  vCtr      Number;
  vSQL      VARCHAR2(2000);
  vcurr    VARCHAR2(256);
BEGIN

  SELECT sys_context('userenv', 'current_schema') into vcurrSchema from dual;
  dbms_output.put_line('Current Schema: '||vcurrSchema);

  SELECT COUNT(*)
  INTO vCtr
  FROM user_tables
  WHERE table_name = 'WLS_EVENTS';

  IF vCtr = 0 THEN
    dbms_output.put_line('Creating WLS_EVENTS table');
    vSQL := 'CREATE TABLE "WLS_EVENTS" (
      "RECORDID" NUMBER(20,0) DEFAULT NULL,
      "TIMESTAMP" NUMBER(20,0) DEFAULT NULL,
      "CONTEXTID" VARCHAR2(250 BYTE) DEFAULT NULL,
      "TXID" VARCHAR2(250 BYTE) DEFAULT NULL,
      "USERID" VARCHAR2(250 BYTE) DEFAULT NULL,
      "TYPE" VARCHAR2(250 BYTE) DEFAULT NULL,
      "DOMAIN" VARCHAR2(250 BYTE) DEFAULT NULL,
      "SERVER" VARCHAR2(250 BYTE) DEFAULT NULL,
      "SCOPE" VARCHAR2(250 BYTE) DEFAULT NULL,
      "MODULE" VARCHAR2(250 BYTE) DEFAULT NULL,
      "MONITOR" VARCHAR2(250 BYTE) DEFAULT NULL,
      "FILENAME" VARCHAR2(250 BYTE) DEFAULT NULL,
      "LINENUM" NUMBER(10,0) DEFAULT NULL,
      "CLASSNAME" VARCHAR2(250 BYTE) DEFAULT NULL,
      "METHODNAME" VARCHAR2(250 BYTE) DEFAULT NULL,
```



```
DECLARE
vCtr      Number;
vSQL      VARCHAR2(1000);
vcurrSchema VARCHAR2(256);
BEGIN

SELECT sys_context( 'userenv', 'current_schema' ) into vcurrSchema from dual;
dbms_output.put_line('Current Schema: '||vcurrSchema);

SELECT COUNT(*)
INTO vCtr
FROM user_tables
WHERE table_name = 'WLS_HVST';

IF vCtr = 0 THEN
dbms_output.put_line('Creating WLS_HVST table');
vSQL := 'CREATE TABLE "WLS_HVST"
(
"RECORDID" NUMBER(20,0) NOT NULL,
"TIMESTAMP" NUMBER(20,0) DEFAULT NULL,
"DOMAIN" VARCHAR2(250 BYTE) DEFAULT NULL,
"SERVER" VARCHAR2(250 BYTE) DEFAULT NULL,
"TYPE" VARCHAR2(250 BYTE) DEFAULT NULL,
"NAME" VARCHAR2(250 BYTE) DEFAULT NULL,
"ATTRNAME" VARCHAR2(250 BYTE) DEFAULT NULL,
"ATTRTYPE" NUMBER(10,0) DEFAULT NULL,
"ATTRVALUE" VARCHAR2(4000 BYTE) DEFAULT NULL,
"WLDFMODULE" VARCHAR2(250 BYTE) DEFAULT NULL
)';
EXECUTE IMMEDIATE vSQL;
vSQL := 'CREATE UNIQUE INDEX WLS_HVST_RECORD_IDX ON WLS_HVST(RECORDID)';
EXECUTE IMMEDIATE vSQL;
vSQL := 'CREATE INDEX WLS_HVST_TS_IDX ON WLS_HVST(TIMESTAMP)';
EXECUTE IMMEDIATE vSQL;
END IF;

SELECT COUNT(*)
INTO vCtr FROM user_tab_columns
WHERE table_name = 'WLS_HVST' AND column_name = 'WLDFMODULE';

IF vCtr = 0 THEN
dbms_output.put_line('Creating WLDFMODULE column in WLS_HVST table');
vSQL := 'ALTER TABLE WLS_HVST ADD("WLDFMODULE" VARCHAR2(250 BYTE) DEFAULT NULL)';
EXECUTE IMMEDIATE vSQL;
END IF;

SELECT COUNT(*) INTO vCtr FROM user_sequences
WHERE sequence_name = 'SEQ_WLS_HVST_RECORDID';

IF vCtr = 0 THEN
vSQL := 'CREATE SEQUENCE SEQ_WLS_HVST_RECORDID MINVALUE 1 MAXVALUE 999999999999999999
START WITH 1 INCREMENT BY 1 NOCACHE';
EXECUTE IMMEDIATE vSQL;
END IF;

SELECT COUNT(*) INTO vCtr FROM user_triggers
WHERE table_name = 'WLS_HVST';

IF vCtr = 0 THEN
vSQL := 'CREATE OR REPLACE TRIGGER TRG_WLS_HVST_INSERT
BEFORE INSERT ON WLS_HVST
```

```

REFERENCING NEW AS newRow
FOR EACH ROW
BEGIN
  IF :newRow.RECORDID IS NULL THEN
    SELECT SEQ_WLS_HVST_RECORDID.nextval INTO :newRow.RECORDID FROM DUAL;
  END IF;
END;';
EXECUTE IMMEDIATE vSQL;
END IF;

END;
/

```

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

MySQL

[Example 7-5](#) shows the DDL that you can use to create the `wls_events` table in MySQL database.

Example 7-5 DDL Definition of the `wls_events` Table in MySQL Database

```

DROP PROCEDURE if exists create_alter_wls_events
/

CREATE PROCEDURE create_alter_wls_events()
language sql
BEGIN
  CREATE TABLE IF NOT EXISTS WLS_EVENTS
  (
    RECORDID BIGINT AUTO_INCREMENT PRIMARY KEY,
    TIMESTAMP BIGINT NOT NULL,
    CONTEXTID VARCHAR(250) default NULL,
    TXID VARCHAR(250) default NULL,
    USERID VARCHAR(250) default NULL,
    TYPE VARCHAR(250) default NULL,
    DOMAIN VARCHAR(250) default NULL,
    SERVER VARCHAR(250) default NULL,
    SCOPE VARCHAR(250) default NULL,
    MODULE VARCHAR(250) default NULL,
    MONITOR VARCHAR(250) default NULL,
    FILENAME VARCHAR(250) default NULL,
    LINENUM INT UNSIGNED default NULL,
    CLASSNAME VARCHAR(250) default NULL,
    METHODNAME VARCHAR(250) default NULL,
    METHODDSC VARCHAR(4000) default NULL,
    ARGUMENTS TEXT(100000) default NULL,
    RETVAL VARCHAR(4000) default NULL,
    PAYLOAD BLOB(100000),
    CTXPAYLOAD VARCHAR(4000),
    DYES BIGINT UNSIGNED default NULL,
    THREADNAME VARCHAR(250) default NULL,
    INDEX(TIMESTAMP)
  );

  IF NOT EXISTS(
    SELECT * FROM `information_schema`.`COLUMNS`
      WHERE COLUMN_NAME='THREADNAME' AND TABLE_NAME='WLS_EVENTS') THEN
    ALTER TABLE `WLS_EVENTS` ADD `THREADNAME` varchar(250) default NULL;
  END IF;

```

```

END
/

CALL create_alter_wls_events()
/

DROP PROCEDURE if exists create_alter_wls_events
/

```

Example 7-6 shows the DDL that you can use to create the `wls_hvst` table in MySQL database.

Example 7-6 DDL Definition of `wls_hvst` Table in MySQL Database

```

DROP PROCEDURE if exists create_alter_wls_hvst
/

CREATE PROCEDURE create_alter_wls_hvst()
language sql
BEGIN
    CREATE TABLE IF NOT EXISTS WLS_HVST
    (
        RECORDID BIGINT AUTO_INCREMENT PRIMARY KEY,
        TIMESTAMP BIGINT NOT NULL,
        DOMAIN VARCHAR(250) default NULL,
        SERVER VARCHAR(250) default NULL,
        TYPE VARCHAR(250) default NULL,
        NAME VARCHAR(250) default NULL,
        SCOPE VARCHAR(250) default NULL,
        ATTRNAME VARCHAR(250) default NULL,
        ATTRTYPE INT default NULL,
        ATTRVALUE VARCHAR(4000) default NULL,
        WLDFMODULE VARCHAR(250) default NULL,
        INDEX(TIMESTAMP)
    );

    IF NOT EXISTS(
        SELECT * FROM `information_schema`.`COLUMNS`
        WHERE COLUMN_NAME='WLDFMODULE' AND TABLE_NAME='WLS_HVST') THEN
        ALTER TABLE `WLS_HVST` ADD `WLDFMODULE` varchar(250) default NULL;
    END IF;

END
/

CALL create_alter_wls_hvst()
/

DROP PROCEDURE if exists create_alter_wls_hvst
/

```

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

Configuring JDBC Resources for WLDF

After you create the tables in your database, you must configure JDBC to access the tables. (See *Administering 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.

If multiple WLDF JDBC archive schemas exist in the same database, you can specify the particular schema to use for accessing JDBC-based archive tables in that database. There is no default value for a schema name: If you do not specify one, no schema name is applied when WLDF validates the runtime table, and no schema name is used for the SQL statements. You specify the schema name in the `WLDFServerDiagnosticMBean.DiagnosticJDBCSchemaName` attribute, which you can access from the Diagnostic Archives: Configuration page in the WebLogic Server Administration Console. See [Configure diagnostic archives](#) in *Oracle WebLogic Server Administration Console Online Help*.

An example configuration for a JDBC-based store is shown in [Example 7-7](#).

Example 7-7 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.

Retiring Data from the Archives

To maintain the archived data, you must delete the old archived data periodically. WLDF includes a configuration-based data retirement feature for doing this. The data can be deleted based on the size of the data and time period when it was created.

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:

- [Configuring Data Retirement at the Server Level](#)
- [Configuring Age-Based Data Retirement Policies for Diagnostic Archives](#)
- [Sample Configuration](#)

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 [Configuring Age-Based Data Retirement Policies for Diagnostic Archives](#).

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.

Sample Configuration

Data retirement configuration settings are persisted in the `config.xml` configuration file for the server's domain, as shown in [Example 7-8](#).

Example 7-8 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>
```

8

Configuring the Harvester for Metric Collection

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

This chapter includes the following sections about the Harvester and how to configure it:

- [Harvesting, Harvestable Data, and Harvested Data](#)
- [Harvesting Data from the Different Harvestable Entities](#)
- [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. The Harvester configuration includes the sampling period, the type of data to harvest, and the type names for WebLogic Server MBeans and custom MBeans.
- [Harvester Performance Considerations](#)

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 Runtime 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 `WLDFHarvesterRuntimeMBean` in the *Oracle WebLogic Server MBean Reference*.

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

Harvesting Data from the Different Harvestable Entities

You can configure the Harvester to 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 8-1](#).

Table 8-1 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 runtime MBean types and attributes are known at startup. Therefore, when the Harvester configuration is loaded, the set of harvestable WebLogic Server entities is the same as the set of WebLogic Server runtime MBean types and attributes. As types are instantiated, those instances also become 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 WebLogic Server 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*.

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. The Harvester configuration includes the sampling period, the type of data to harvest, and the type names for WebLogic Server MBeans and custom MBeans.

[Example 8-1](#) 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 (that is, non-WebLogic Server) MBean. The text following the listing explains each element in the listing.

Example 8-1 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>

```

- [Configuring the Harvester Sampling Period](#)
- [Configuring the Types of Data to Harvest](#)
- [Specifying Type Names for WebLogic Server MBeans and Custom MBeans](#)
- [Harvesting from the Domain Runtime MBean Server](#)
- [When Configuration Settings Are Validated](#)
- [Sample Configurations for Different Harvestable Types](#)

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 .

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 [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 [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.

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 runtime 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 runtime 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. (See [Example 8-1](#).)
- 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 cannot be harvested.

Harvesting from the Domain Runtime MBean Server

The `<harvested-type>` element supports a `<namespace>` attribute that lets you harvest metrics from MBeans registered in the Domain Runtime MBean Server. However, Oracle recommends that you limit the usage to harvesting only Domain Runtime-specific MBeans, such as the `ServerLifecycleRuntimeMBean`. Harvesting of remote managed server MBeans through the Domain Runtime MBean Server 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 Domain Runtime MBean server is available only on the Administration Server. Attempts to harvest Domain Runtime MBeans on a Managed Server are ignored. See [Example 8-5](#).

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.

Sample Configurations for Different Harvestable Types

In [Example 8-2](#), the `<harvested-type>` element in the `DIAG_MODULE.xml` configuration file specifies that the `ServerRuntimeMBean` is to be harvested. Because no `<harvested-instance>` subelement is present, all instances of the type will be collected. However, because there is always only one instance of the server runtime MBean, there is no need to provide a specific list of instances. And because there are no `<harvested-attribute>` subelements present, all available attributes of the MBean are harvested for each of the two instances.

Example 8-2 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 8-3](#), 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 subelement `<harvested-attribute>` specifies that only two of the available attributes of the `WLDFHarvesterRuntimeMBean` will be harvested: `TotalSamplingTime` and `CurrentSnapshotElapsedTime`.

Example 8-3 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 8-4](#), 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 8-4 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 8-5](#), 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 [Harvesting from the DomainRuntime MBeanServer](#)). The subelement `<harvested-attribute>` specifies that only the `StateVal` attribute will be harvested.

Example 8-5 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>

```

Harvester Performance Considerations

Because the Harvester tracks all MBeans that are registered in the local WebLogic Server Runtime MBean server, applications that create a high volume of transient MBeans can create performance issues in WLDF. Here, a transient MBean is an MBean with a very short life span that can be registered and unregistered very quickly, typically within the space of a few milliseconds. Such MBeans can create a load stress in the Harvester and the Policies and Actions system, which tracks MBean registrations. This performance problem is particularly a risk when high-volume JMS applications are not coded according to recommended best practices.

When JMS connections are not cached properly, a scenario can develop in which hundreds of connections (and consequently, the corresponding connection, producer, and consumer runtime MBeans) are created and destroyed every second when the system is operating under heavy load. This situation can cause load stress on both the Harvester and the Policies and Actions system.

To avoid this problem, make sure your JMS applications conform to the best coding practices described in *Cache and Re-use Client Resources* in *Tuning Performance of*

Oracle WebLogic Server. As a result, you will not only obtain better WLDF performance, but you will also improve your JMS and overall server performance.

9

Configuring Policies and Actions

The Policies and Actions component of the WebLogic Diagnostics Framework (WLDF) provides the means for monitoring server and application states and then executing actions based on criteria set in the policies. Policies and actions are configured as part of a diagnostic module that is targeted to one or more server instances in a domain.



Note:

As of WebLogic Server 12.2.1, the terms **watch** and **notification** are replaced by **policy** and **action**, respectively. However, the definition of these terms has not changed.

The following sections give an overview of the Policies and Actions component, and also provide an example of a Policies and Actions configuration:

- [Policies and Actions](#)
You can configure policies to analyze log records, data events, and harvested metrics.
- [Overview of Policies and Actions Configuration](#)
A complete policy and action configuration includes settings for one or more policies, one or more actions, and any underlying configurations required for the action media; for example, the SNMP configuration required for an SNMP-based action.
- [Sample Policies and Actions Configuration](#)
A set of policies and actions is configured in a diagnostic module file named `DIAG_MODULE.xml`.

Policies and Actions

You can configure policies to analyze log records, data events, and harvested metrics.

A *policy* identifies a situation that you want to trap for monitoring or diagnostic purposes.

A policy includes:

- A policy expression (with the exception of calendar-based policies)
The default language for policy expressions is the WLDF query language; however, the WLDF query language is deprecated. You can also use Java Expression Language (EL) for policy expressions.
- An alarm setting
- One or more action handlers

You can also configure policies to enable elasticity in dynamic clusters; that is, to automatically scale a dynamic cluster up or down by a specific number of server instances. You can define policies to enable two broad categories of elasticity:

- Calendar-based scaling — Scaling operations on a dynamic cluster that are executed on a particular date and time.
- Policy-based scaling — Scaling operations on a dynamic cluster that are executed in response to changes in demand.

 **Note:**

To configure an elastic scaling policy for a dynamic cluster, you must create a domain-scope diagnostic system module in which you define the scaling policy, and then target that diagnostic module to the Administration Server.

For more information about enabling elasticity in WebLogic Server, including instructions for downloading and running a demonstration example, see Policy-Based Scaling in *Configuring Elasticity in Dynamic Clusters for Oracle WebLogic Server*.

An *action* is an operation that is executed when a policy expression evaluates to true. WLDF supports the following types of actions:

- Scaling a dynamic cluster
- Java Management Extensions (JMX)
- Java Message Service (JMS)
- Simple Mail Transfer Protocol (SMTP), for example, e-mail
- Simple Network Management Protocol (SNMP)
- Diagnostic image
- Log
- REST
- Script
- Heap dump
- Thread dump

You must associate a policy with an action for a useful diagnostic activity to occur; for example, to notify an administrator about specified states or activities in a running server.

Policies and actions are configured separately from each other. An action can be associated with multiple policies, and a policy can be associated with multiple actions. This provides the flexibility to recombine and re-use policies and actions, according to current needs.

Overview of Policies and Actions Configuration

A complete policy and action configuration includes settings for one or more policies, one or more actions, and any underlying configurations required for the action media; for example, the SNMP configuration required for an SNMP-based action.

The main elements required for configuring policies and actions in a WLDF system resource descriptor file, `DIAG_MODULE.xml`, are shown in [Example 9-1](#). As the listing shows, the base element for defining policies and actions is `<watch-notification>`.

Policies are defined in <watch> elements, and actions are defined in elements named for each of the types of action; for example, <jms-notification>, <jmx-notification>, <smtp-notification>, and <image-notification>.

Example 9-1 A Skeleton Policy and Action 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-notification>
</wldf-resource>
<!-- ----- Other system resource configuration elements ----- -->
  <watch-notification>
    <log-watch-severity>
      <!-- Threshold severity for a log policy to be evaluated further
           (This can be narrowed further at the policy level.) -->
    </log-watch-severity>
  </watch-notification>
<!-- ----- Policy configuration elements: ----- -->
  <watch>
    <!-- A policy expression -->
  </watch>
  <watch>
    <!-- A policy expression -->
  </watch>
<!-- Any other policy configurations -->

<!-- ----- Action configuration elements: ----- -->
<!-- The following action configuration elements show one of each
      type of supported actions. 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 action; 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 action -->
  </jmx-notification>
  <smtp-notification>
    <!-- Configuration for an SMTP-based action; requires a
           corresponding SMTP configuration via a mail-session element -->
  </smtp-notification>
  <snmp-notification>
    <!-- Configuration for an SNMP-based action; requires a
           corresponding SNMP agent configuration via an snmp-agent
           element -->
  </snmp-notification>
  <image-notification>
    <!-- Configuration for an image-based action -->
  </image-notification>
  <watch-notification>
<!-- ----- Other configuration elements ----- -->
</wldf-resource>
```

 **Note:**

While the notification media must be configured so they can be used by the actions 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 policy and action can be individually enabled and disabled by setting `<enabled>>true</enabled>` or `<enabled>>false</enabled>` for the individual policy or action. In addition, the entire policy and action facility can be enabled and disabled by setting `<enabled>>true</enabled>` or `<enabled>>false</enabled>` for all policies and actions. The default value is `<enabled>>true</enabled>`.

The `<watch-notification>` element contains a `<log-watch-severity>` sub-element, which affects how actions are executed by log policies.

If the maximum severity level of the log messages that triggered the policy do not at least equal the provided severity level, then the resulting actions are not executed. Note that this only applies to actions executed by log policies. Do not confuse this element with the `<severity>` element defined on policies. The `<severity>` element assigns a severity to the policy itself, whereas the `<log-watch-severity>` element controls which actions are executed by log-type policies.

For information about how to configure policies and actions using the WebLogic Server Administration Console, see [Configure policies and actions](#) in *Oracle WebLogic Server Administration Console Online Help*.

Sample Policies and Actions Configuration

A set of policies and actions is configured in a diagnostic module file named `DIAG_MODULE.xml`.

[Example 9-2](#) shows a complete configuration. The details of this example are explained in the following topics:

- [Configuring Policies](#)
- [Configuring Actions](#)

Example 9-2 Sample Policies and Actions 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/2.0/
weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <!-- Instrumentation must be configured and enabled for instrumentation
  policies -->
  <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
      policies. 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 policies and actions are defined under the
      watch-notification element -->
<watch-notification>
  <enabled>true</enabled>
  <log-watch-severity>Info</log-watch-severity>
  <!-- A harvester policy configuration -->
  <watch>
    <name>myWatch</name>
    <enabled>true</enabled>
    <rule-type>Harvester</rule-type>
    <rule-expression>${com.bea:Name=myserver,Type=ServerRuntime//
SocketsOpenedTotalCount} &gt;= 1</rule-expression>
    <alarm-type>AutomaticReset</alarm-type>
    <alarm-reset-period>60000</alarm-reset-period>
    <notification>myMailNotif,myJMXNotif,mySNMPNotif</notification>
  </watch>
  <!-- An instrumentation policy 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 policy 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 actions -->
  <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>
```

10

Configuring Policies

The WebLogic Diagnostics Framework (WLDF) provides three main types of policies, which are differentiated by the sorts of data each can monitor. The policy types are:

- **Scheduled** policies, which monitor diagnostic data that is generated by runtime MBeans according to a specific schedule. These policies can also be used to execute an action at a specific time or on a schedule.
- **Log** policies, which monitor messages generated into the server or domain logs.
- **Instrumentation** policies, also known as Event Data policies, which monitor events generated by the WLDF Instrumentation component.

This chapter explains how to configure each policy type and includes the following sections:

For information about how to create a policy using the WebLogic Server Administration Console, see [Create policies for a diagnostic system module](#) in *Oracle WebLogic Server Administration Console Online Help*.

- [How Policies Are Configured](#)
There are several components of a policy that you configure, such as the type, expression, corresponding actions to be executed when the policy is evaluated to true, and more.
- [Configuring Scheduled Policies](#)
- [Configuring Log Policies](#)
Use log policies to monitor the occurrence of specific messages or strings in the server or domain log. Policies of this type are triggered as a result of a log message containing the specified data being issued.
- [Configuring Instrumentation Policies](#)
You use instrumentation policies to monitor the events from the WLDF Instrumentation component. Policies of this type are evaluated as a result of an event being posted by the Instrumentation component, which occurs when code that matches a deployed Instrumentation monitor is exercised.
- [Creating Complex Policy Expressions Using WLDF Java EL Extensions](#)
Oracle expects that the library of smart rules packaged with WLDF are sufficient for meeting the needs of creating scheduled policies that evaluate runtime performance data in a server or cluster. However, if you have a specific scheduled policy need that cannot be satisfied by a smart rule, WLDF also provides a set of extensions to Java EL. These extensions are intended for use in policies that evaluate very specific characteristics or trends in metrics collected from runtime MBean servers in your WebLogic domain.

How Policies Are Configured

There are several components of a policy that you configure, such as the type, expression, corresponding actions to be executed when the policy is evaluated to true, and more.

You can use any of the following tools to configure policies for diagnostic system modules:

- WebLogic Server Administration Console

- Fusion Middleware Control
- WLST
- REST
- JMX application

This chapter refers primarily to using the WebLogic Server Administration Console or WLST.

The following table summarizes the attributes, elements, and options that you configure when creating a policy, and also identifies any requirements each configuration item has for specific policy types.

Item	Description	Policy Requirement
Rule Type	Attribute that determines the policy's type. The default is <code>Harvester</code> .	Must be specified for log and instrumentation policies. Optional for scheduled policies.
Expression Language	Attribute that establishes the language used in the policy expression. The two supported languages are Java Expression Language (EL), and WLDF query language (deprecated).	Use EL in all policy types. The WLDF query language is supported, but deprecated.
Policy Expression	Expression that identifies a situation or condition that you want to trap for monitoring or diagnostic purposes. The expression can analyze log records, data events, or MBean metrics, depending on the rule type setting.	Optional for scheduled policies, but required for all others. If a scheduled policy does not include an expression, the policy always executes the associated actions according to the Policy Schedule .
Actions	One or more operations that are executed when a policy expression is evaluated to <code>true</code> .	Optional.
Policy Schedule	A calendar-based schedule that determines when a scheduled policy is evaluated.	Required for all scheduled policies. Not available for log or instrumentation policies.
Alarm Options	Options that determine whether, or when, a policy can be evaluated again after it has been evaluated to <code>true</code> . The default is <code>None</code> , which enables the policy to always be evaluated again.	Optional for all policy types.
Severity Option	Log message severity value that, when the policy is evaluated to <code>true</code> , is: <ol style="list-style-type: none"> 1. Specified for the log message that is generated in the logging system. 2. Passed to the actions that are configured with the policy. The default is <code>Notice</code> .	Optional for all policy types.
Enablement Option	Flags that either enable or disable a policy from being evaluated. The default is <code>enabled</code> .	Optional for all policy types.

- [Rule Type](#)

- [Expression Language](#)
- [Policy Expression](#)
- [Actions](#)
- [Policy Schedule](#)
- [Alarm Options](#)
- [Severity Option](#)
- [Enablement Option](#)

Rule Type

When creating a policy, you must define its type in its **rule type** attribute. Policies with different rule types differ in two ways:

- The syntax for specifying the conditions being monitored are unique to the rule type.
- Log and instrumentation policies are triggered in real time, whereas scheduled policies are triggered by settings on the [WLDFScheduleBean](#) interface, described in [Policy Schedule](#).

The way to define the rule type depends on the tool you use to create the policy:

- If you are using the WebLogic Server Administration Console or Fusion Middleware Control, the rule type is determined by the policy type you are creating. For each of the policy types you can choose in either console, the following table identifies the corresponding rule type and [WLDWatchBean.RuleType](#) attribute value that is defined for that policy:

Policy Type . . .	Rule Type	WLDWatchBean.RuleType Value
Smart Rule	Harvester	Harvester
Calendar Based	Harvester	Harvester
Collected Metrics	Harvester	Harvester
Server Log	Log	Log
Domain Log	Log	DomainLog
Event Data	Instrumentation	EventData

For information about choosing a policy type using the WebLogic Server Administration Console, see [Create policies for a diagnostic system module](#) in *Oracle WebLogic Server Administration Console Online Help*.

- If you are using WLST, REST, or JMX to configure a policy, you set the [WLDWatchBean.RuleType](#) attribute as follows:

Policy Type	Rule Type Attribute
Scheduled policy	Harvester
Log policy	Log - for server log monitoring DomainLog - for domain log monitoring
Instrumentation	EventData - for instrumentation event monitoring

Expression Language

Policy expressions can be created using either of the following languages:

- Java Expression Language (EL) (recommended)
- WLDF query language (deprecated in WebLogic Server 12.2.1)

See Java EL tutorials at <https://docs.oracle.com/javasee/7/tutorial/jsf-el.htm#GJDDD>. For more information about Java (EL), see the JSR-000341 Expression Language 3.0 specification at <https://jcp.org/aboutJava/communityprocess/final/jsr341/index.html>.

If you have diagnostic system modules created with a previous release of WebLogic Server, WLDF supports expressions that use the WLDF query language. If you are creating new policies for either an existing or a new diagnostic system module, Oracle strongly recommends using Java EL as the policy expression language.

 **Note:**

The policies described in this chapter are Java EL based. For information about configuring policies that use the WLDF query language, see [WLDF Query Language-Based Policies](#).

Policy Expression

A policy expression encapsulates all information necessary for specifying a rule that, when evaluated to `true`, causes the associated actions to be executed. When you use Java EL as the expression language, you can construct a policy expression that uses the following out-of-the-box resources to set the conditions that determine whether to fire an associated action:

- Beans
A bean is a Java object that represents the data available for a policy expression to use, such as metrics from MBeans, log event information, or structured data surfaced by other beans. Beans are accessed in policy expressions using standard JavaBean conventions.
- Functions
Functions are a set of operations that are provided either by EL itself, or by WLDF, that can be utilized from policy expressions to transform or evaluate data.
- Smart rules
Smart Rules are special set of functions that encapsulate more complex logic and monitoring capabilities, and have specialized support in both the WebLogic Server Administration Console and Fusion Middleware Control. They can be used by themselves, or with other expression components as part of a larger, more complex expression.

Actions

Each policy can be associated with one or more actions that are executed whenever the policy evaluates to `true`. See [Configuring Actions](#).

Policy Schedule

All scheduled policies must be configured with a schedule. Scheduling allows policies to be evaluated according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals.

You configure a policy schedule by setting attributes on the `WLDFScheduleBean` interface, which is a property of the `WLDFWatchBean`. You can set these attributes using the WebLogic Server Administration Console, WLST, REST, or a JMX application. When you are configuring new policies, the WebLogic Server Administration Console and Fusion Middleware Control provide convenient assistants and workflows for configuring common scheduling scenarios.



Note:

The `WLDFScheduleBean` is used for policy evaluation only when:

- The configured policy rule type is "Harvester".
- The configured expression language for the policy is "EL".

Note also that although scheduled policies that use the `WLDFScheduleBean` for scheduling are configured as Harvester types, the WLDF Harvester component is not used for scheduling.

[Table 10-1](#) lists the attributes of the `WLDFScheduleBean` and their default values, which are the same as for the `javax.ejb.ScheduleExpression` interface. In addition, the syntax for specifying a value, range, list, or interval for a specific unit of time is also the same as that described for the `ScheduleExpression` interface. See <http://docs.oracle.com/javaee/7/api/javax/ejb/ScheduleExpression.html>.

Table 10-1 WLDFScheduleBean Attributes and Default Values

Attribute	Description	Default	Allowable Values and Examples
second	One or more seconds within a minute	0	<p>Allowable values: 0 to 59</p> <p>Can be a value, range, list, or interval. To specify every <i>n</i> seconds of the minute, use <code>"*/n"</code>.</p> <p>For example:</p> <ul style="list-style-type: none"> <code>second = "30"</code> - (value) run policy every 30 seconds within the minute <code>second = "10,20,30"</code> - (list) run policy on seconds 10, 20, and 30 within the minute <code>second = "1-10"</code> - (range) run policy on each of seconds 1 through 10 within the minute <code>second = "30/10"</code> - (interval) run policy every 10 seconds within the minute, starting at second 30 <code>second = "*/5"</code> - (interval) run policy every 5 seconds within the minute
minute	One or more minutes within an hour	<code>*/5</code>	<p>Allowable values: 0 to 59</p> <p>Can be a value, range, list, or interval. To specify every <i>n</i> minutes of the hour, specify <code>"*/n"</code>.</p> <p>For example:</p> <ul style="list-style-type: none"> <code>minute = "30"</code> - (value) run policy every 30 minutes <code>minute = "*/2"</code> - (interval) run policy every two minutes of the hour
hour	One or more hours within a day	*	<p>Allowable values: 0 to 23</p> <p>Can be a value, range, list, or interval.</p> <p>For example:</p> <ul style="list-style-type: none"> <code>hour="16"</code> - (value) run policy at 16:00. <code>hour = "*" - (range) run policy at every hour within a day.</code>
dayOfWeek	One or more days within a week	*	<p>Allowable values:</p> <ul style="list-style-type: none"> 0 to 7, where 0 and 7 represent Sunday. For example, <code>dayOfWeek="3"</code> Sun, Mon, Tue, Wed, Thu, Fri, Sat. For example, <code>dayOfWeek="Mon"</code> <p>Can be a value, range, or list. For example:</p> <ul style="list-style-type: none"> <code>dayOfWeek = "3"</code> - run policy on Wednesday <code>dayOfWeek = "Mon-Fri"</code> - run policy each day from Monday to Friday <code>dayOfWeek = "Mon, Wed, Fri"</code> - run policy on Monday, Wednesday, and Friday

Table 10-1 (Cont.) WLDFScheduleBean Attributes and Default Values

Attribute	Description	Default	Allowable Values and Examples
dayOfMonth	One or more days within a month	*	<p>Allowable values:</p> <ul style="list-style-type: none"> 1 to 31 Last -7 to -1 {1st, 2nd, 3rd, 4th, 5th, Last} {Sun, Mon, Tue, Wed, Thu, Fri, Sat} <p>Last represents the last day of the month.</p> <p>-x (where x is in the range 7 to 1) means x days before the last day of the month.</p> <p>1st, 2nd, and so on, specified with a day of the week identifies a single occurrence of that day within the month.</p> <p>Can be a value, range, or list. For example:</p> <ul style="list-style-type: none"> dayOfMonth = "1" - run policy on first day of the month dayOfMonth = "-3" - run policy on the third day before the end of the month dayOfMonth = "2nd Mon" - run policy on the second Monday of the month dayOfMonth = "1st Fri, 3rd Fri" - run policy on the first and third Friday of the month dayOfMonth = "1 to 10" - run policy on each of the first 10 days of the month
month	One or more months within a year	*	<p>Allowable values:</p> <ul style="list-style-type: none"> 1 to 12. Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec <p>Can be a value, range, or list. For example:</p> <ul style="list-style-type: none"> month = "7" - run policy on the 7th month of the year month = "Feb" - run policy in February month = "1 - 3" - run policy for the first three months of the year month = "Jan, Apr, Jul, Oct" - run policy in January, April, July, and October
year	A specific calendar year	*	<p>Allowable values: a four-digit calendar year.</p> <p>You can specify one value. For example:</p> <ul style="list-style-type: none"> year = "2015" - run policy in 2015
timezone	Time zone for the schedule	null	<p>Defaults to the local VM time zone. You may use this attribute to specify a non-default time zone ID in whose context the schedule specification is to be evaluated.</p>

Alarm Options

A policy that has been evaluated to `true` is referred to as having been *triggered*. For policies that are run repeatedly, an alarm determines when a policy can be evaluated again after it has been triggered. If a policy is configured with an alarm, a triggered policy is not evaluated again until the alarm is reset. For policies that are evaluated repeatedly, you can optionally define a minimum time that must transpire after a policy has been triggered before the policy can be evaluated again.

An alarm is important to configure for a policy that is run repeatedly to prevent the associated actions from being executed too frequently, such as generating a flood of emails or JMX

notifications. For example, if you have a scheduled policy that executes a scale up action on a dynamic cluster, you should set an alarm that delays evaluating the policy again until the dynamic cluster is fully scaled up and is processing incoming requests. This delay is referred to as the *alarm reset period*. Without a proper alarm reset period, the scale up action could be executed again prematurely and counterproductively.

To configure an alarm for a policy, you specify the following:

- The alarm type
- The alarm reset period

The following table lists and describes each of the available alarm types:

Alarm Type	Description
None	Allows the policy to be triggered whenever possible. This is the default.
AutomaticReset	Allows the policy to be triggered whenever possible, except that subsequent occurrences cannot occur any sooner than the interval specified in the alarm reset period.
ManualReset	Allows the policy to be triggered only once. After it is triggered, you must manually reset it to fire again. You can reset an alarm using a run-time MBean operation, either programmatically or with WLST. For example, you can use the <code>resetWatchAlarm</code> operation on the WLDWatchNotificationRuntimeMBean .

Note the following alarm state behaviors:

- When the alarm type is `AutomaticReset`, a policy enters the *alarm state* when triggered and stays in that state until the time interval specified by the alarm reset period has expired.
- If a policy is configured with a `ManualReset` alarm, the policy enters the alarm state when triggered, and remains in that state until you manually reset it.
- When a policy is in the alarm state, the policy is not evaluated again until the alarm is reset.
- If a policy's alarm type is `None`, the configured action can be executed every time that the policy is triggered. The alarm state is never set in these cases.

Severity Option

Whenever a policy is triggered, a message is automatically generated in the logging system. The severity option is an optional value you can configure that:

1. Gets assigned as the severity value of the log message generated in the logging system.
2. Is also passed to the actions that are configured with the policy.

The severity option must be one that is defined for the WebLogic logging service in the [weblogic.logging.Severities](#) class. The accepted values are `Info`, `Notice`, `Warning`, `Error`, `Critical`, `Alert`, and `Emergency`. The default is `Notice`.

Enablement Option

Each policy can be individually enabled and disabled by using the `Enabled` attribute on that policy. The value you specify for this attribute is `true` or `false`. When disabled, a policy is not evaluated and its configured actions are not executed.

However, note that the `WLDFWatchNotificationBean`, which is the parent of all policy and action configurations in a diagnostic system module, also has an `Enabled` attribute. If the value of the `WLDFWatchNotificationBean.Enabled` attribute is `false`, all individual policies in the diagnostic system modules are disabled regardless of whether its policies are individually configured as enabled.

Configuring Scheduled Policies

Scheduled policies monitor diagnostic data that consists of data coming from MBeans within the WebLogic Server Runtime MBean Server, including the read-only configuration MBeans in the WebLogic Server Runtime MBean Server. These values, called **metrics**, originate from common WebLogic Server JMX data sources such as the following:

- WebLogic Server Runtime MBean Server
- Domain Runtime MBean Server
- JVM platform MBean server

Scheduled policies are useful for monitoring run-time state information in the WebLogic Server environment. Examples of diagnostic data that is useful to monitor using scheduled policies are:

- Changes over time in average JVM heap usage
If the average amount of free heap reaches a particular threshold that is defined in the policy expression, the configured action is executed, such as sending an email to the system administrator.
- Data from multiple services that are considered together, such as response-time metrics reported by a load balancer and message-backlog metrics from a message queue
If the combination of data meets a particular set of criteria defined in the policy expression, the policy can fire a scaling action

See also [Chaining Policies](#) for information about how to create a policy expression that can reference the state of other policies defined within the same WLDF module as the beans. Policy chaining allows the state of one policy to be part of the expression of another.

The following sections explain how to configure, and show examples of, the three scheduled policy types:

- [Configuring Calendar Based Policies](#)
- [Configuring Smart Rule Based Policies](#)
- [Chaining Policies](#)

Configuring Calendar Based Policies

The simplest type of scheduled policy is the calendar based policy. You can use a calendar based policy to fire any associated actions according to the policy's schedule.

Calendar-based policies are simply scheduled policies with no associated expression. This enables purely schedule-driven action execution; that is, the ability to unconditionally perform a set of actions on a specified schedule. If no expression is provided, when the scheduled time occurs, the policy treats the empty expression as a `true` result and executes the associated actions.

**Note:**

Calendar based policies are supported only for policies that: have the following configuration attributes:

- The rule type specified as `'Harvester'`
- The expression language specified as `'EL'`

The following example shows the configuration of a calendar based policy using WLST. This policy fires a scale up action at 3:00 a.m. on December 26.

```
calendarScaleUp=wn.lookupWatch('ChristmasReturnsScaleUpWatch')
if calendarScaleUp == None:
    print "Creating scale-up for the post-Christmas returns rush on Dec 26 at 3am"
    calendarScaleUp=wn.createWatch('ChristmasReturnsScaleUpWatch')
calendarScaleUp.setRuleType('Harvester')
calendarScaleUp.setExpressionLanguage('EL')
calendarScaleUp.getSchedule().setHour('3')
calendarScaleUp.getSchedule().setMinute('0')
calendarScaleUp.getSchedule().setSecond('0')
calendarScaleUp.getSchedule().setDayOfMonth('26')
calendarScaleUp.getSchedule().setMonth('Dec')
calendarScaleUp.setEnabled(false)
calendarScaleUp.addNotification(scaleUp)
```

Configuring Smart Rule Based Policies

Smart rules are prepackaged functions that greatly simplify the creation of policy expressions. The WebLogic Server Administration Console and Fusion Middleware Control, in particular, each contain a smart rule editor to greatly simplify the task of configuring a smart rule for the policy you are creating.

Smart rules perform a number of complex operations, but surface only a small number of configuration parameters that you set. Details about the specific low level metrics that are collected, how they are used, and so on, are hidden, thereby making them easy to use. Smart rules return only a Boolean value, which determines whether the policy is evaluated to `true`.

You use a smart rule as a predicate in policy expression by simply specifying the parameters required by that smart rule. For example, to evaluate whether a particular increase exists in the average thread pool queue length in the local server, you create a policy that specifies the `ServerHighQueueLength` smart rule as the policy expression and provide the following parameters:

- The sampling period for collecting the value of the `ThreadPoolRuntimeMBean.QueueLength` attribute
- Duration, or the most recent window of time, in which samples are retained

- A threshold value that establishes the maximum acceptable number of threads in the queue

The smart rule takes responsibility for sampling the appropriate metrics over the specified time interval, computing averages, comparing threshold values, and determining whether the smart rule evaluates to `true`.

**Note:**

Smart rules are supported for use only in scheduled policies that are configured with Java EL as the expression language.

- [Types of Diagnostic Data that Smart Rules Evaluate](#)
- [Smart Rule Example](#)

Types of Diagnostic Data that Smart Rules Evaluate

Smart rules can monitor trends in metrics in a server or cluster over time, including:

- Average system throughput
- Process CPU load
- Pending user request count
- Idle or stuck thread count
- Incoming request queue size
- System load average
- JVM free heap size
- Any metric value visible from JMX, such as custom MBean values

You can use smart rules as building blocks in policy expressions. In the simplest case, a single smart rule can be used by itself in a policy expression. You can also combine a smart rule with others, as well as with other EL constructs, to form more complex expressions.

For example, you can construct a policy that sends an email notification if all of the following conditions occur simultaneously in a server instance or cluster:

- Low JVM free heap percentage
- High number of stuck threads
- High incoming requests queue size

For details about all the smart rules provided by WLDF, see [Smart Rule Reference](#).

Smart Rule Example

The [ClusterLowHeapFreePercent](#) smart rule compares the average free heap across all Managed Servers in a cluster by monitoring the value of the [JVMRuntimeMBean.HeapFreePercent](#) attribute. A policy expression that uses this smart rule will be evaluated to `true` if a minimum percentage of Managed Servers in the cluster have an average free heap that is less than a particular threshold value.

The `ClusterLowHeapFreePercent` smart rule takes the following input parameters:

- Cluster name
- Sampling period — The frequency with which the value of the `HeapFreePercent` metric is collected
- Retention window — A sliding window of time during which samples are retained. For example, the most recent five minutes.
- `percentFreeLimit` — A value that represents the low free heap percentage threshold.
- `percentServersLimit` — A percentage of Managed Servers in the cluster that must have an average free heap that is less than `percentFreeLimit` to cause the expression to evaluate to `true`.

The following is an example configuration of the `ClusterLowHeapFreePercent` smart rule:

```
wls:ClusterLowHeapFreePercent("myCluster","30 seconds","10 minutes",20,60)
```

For every Managed Server in `myCluster`, this smart rule collects the value of the `HeapFreePercent` every 30 seconds, retaining the most recent 10 minutes of data, and evaluates to `true` if at least 60 per cent of the Managed Servers in `myCluster` have an average free heap percentage that is less than 20 per cent.

This smart rule could be configured to fire an action when it evaluates to `true`, such as sending an email to the system administrator to report that a low free heap condition exists in the cluster. The system administrator can then take remedial action as necessary.

You can use smart rules in conjunction with scaling actions, described in [Configuring Elastic Actions](#), to configure policy based scaling of a dynamic cluster. This capability enables automated elasticity in that cluster. For more information, including a demo that you can download and run, see Policy-Based Scaling in *Configuring Elasticity in Dynamic Clusters for Oracle WebLogic Server*.

Chaining Policies

Within the same diagnostics system module, the expression in one policy can reference other policies as beans within that expression. In this way, complex policy expressions can be reused and "chained" together to allow the state of one policy to be part of the expression of another. This allows more complex, interrelated policies to be written, while keeping such policy configurations more readable and maintainable.

To allow access to policy states within an expression, you can use the `resource` bean within the global bean name space for each policy. The `resource` bean supports a `Map` attribute named `watches`, where each key in the map is the name of a policy defined within the same diagnostics system module.

Each value in the policy's map is a bean representing the named policy. These policy beans support a simple Boolean alarm attribute, which has the following semantics:

- If the policy is configured with an alarm type other than `None`, the alarm attribute returns `true` if the policy is currently in the alarm state.
- If no alarm type is configured on the policy, the alarm attribute yields the most recently evaluated result.

- If the alarm attribute on a policy bean is accessed before the named policy has successfully completed an evaluation cycle, a `NotEnoughDataException` is thrown. This occurrence also has the effect of invalidating the expression during that evaluation cycle: the policy isn't disabled, but it is effectively a non-result when it occurs.

Configuring Log Policies

Use log policies to monitor the occurrence of specific messages or strings in the server or domain log. Policies of this type are triggered as a result of a log message containing the specified data being issued.

When creating a log policy, you can use the `log` bean in a policy expression to obtain access to data to log message fields. See [log](#) for details about the available log bean attributes.

The following example looks for a log message indicating that the server is entering the RUNNING state:

```
w=cmo.createWatch("ServerLogRunningState")
w.setExpressionLanguage('EL')
w.setRuleType('Log')
w.setRuleExpression("log.messageId == 'BEA-000365' and
log.logMessage.contains('RUNNING')")
```

You can also use java methods and field accessors to access the data in log, since the log bean is a simple `JavaBean` object. An equivalent policy expression of the above example is:

```
w=cmo.createWatch("ServerLogRunningState2")
w.setExpressionLanguage('EL')
w.setRuleType('Log')
w.setRuleExpression("log.getMessageId().contains('000365') &&
log.getLogMessage().contains('RUNNING')")
```

Note:

Any log policies that search for the RUNNING state message ID should search for message ID `BEA-000365`, and not `BEA-000360`. The message ID `BEA-000360` is issued immediately before the state change to RUNNING, and `BEA-000365` is issued immediately afterward. WLDF does not start rule evaluation until the server is in the RUNNING state. Therefore, such log policies are able to find only message ID `BEA-000365`.

Configuring Instrumentation Policies

You use instrumentation policies to monitor the events from the WLDF Instrumentation component. Policies of this type are evaluated as a result of an event being posted by the Instrumentation component, which occurs when code that matches a deployed Instrumentation monitor is exercised.

Instrumentation policy expressions utilize a single bean named `instrumentationEvent`. This bean provides access to the data that is captured in an Instrumentation event. As with Log,

DomainLog, and Collected Metrics policies, you can access data in the Instrumentation event using JavaBean conventions in the policy expression. See the set of fields that are accessible on the [instrumentationEvent](#) bean.

The following example shows how to access data in an Instrumentation policy using the `instrumentationEvent` bean:

```
instrumentationEvent.payload > 100000000 && instrumentationEvent.monitor
== 'Servlet_Around_Service'
```

This policy triggers when the monitor event is of type “Servlet_Around_Service” and the payload value (in this case, the execution time of the servlet recorded by the Servlet_Around_Service monitor) is greater than 100000000 nanoseconds (100 milliseconds). You can also use java methods and field accessors to access data in `instrumentationEvent`, since the `instrumentationEvent` bean is a simple JavaBean object. An equivalent policy expression of the example above can be given as:

```
instrumentationEvent.getPayload() > 100000000 &&
instrumentationEvent.getMonitor().equals('Servlet_Around_Service')
```

[Example 10-1](#) shows an example configuration for an Instrumentation policy.

Example 10-1 Sample Configuration for an Instrumentation Policy (in DIAG_MODULE.xml)

```
<watch-notification>
  <watch>
    <name>myInstWatch</name>
    <enabled>true</enabled>
    <rule-type>EventData</rule-type>
    <rule-expression>instrumentationEvent.payload > 100000000 & & &
instrumentationEvent.monitor == 'Servlet_Around_Service'</rule-expression>
    <expression-language>EL</expression-language>
    <alarm-type>ManualReset</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>
```

Creating Complex Policy Expressions Using WLDF Java EL Extensions

Oracle expects that the library of smart rules packaged with WLDF are sufficient for meeting the needs of creating scheduled policies that evaluate runtime performance data in a server or cluster. However, if you have a specific scheduled policy need that cannot be satisfied by a smart rule, WLDF also provides a set of extensions to Java EL. These extensions are intended for use in policies that evaluate very specific characteristics or trends in metrics collected from runtime MBean servers in your WebLogic domain.

The contents of this section are targeted to developers who are knowledgeable of complex programming techniques. Experience with Java EL is highly recommended.

Using WLDF Beans and Functions

WLDF leverages Java EL as the language for writing policy expressions. Java EL is a standard, extensible, and robust scripting language. WLDF has adopted and extended Java EL to provide access to WebLogic diagnostic data and events for writing policy expressions. WLDF provides a set of functions and JavaBean objects for writing policy expressions that use the following diagnostic data and events:

- WebLogic Runtime MBean data
- WebLogic Logging events
- WebLogic Instrumentation events

You can utilize all the features available within Java EL in conjunction with these WLDF extensions to write policy expressions. Collected metrics based policies, which are a type of scheduled policy, can use WLDF-provided beans and functions within their policy expressions. These beans are JavaBean objects that can obtain access to common WebLogic Server JMX data sources, such as the following:

- WebLogic Server Runtime MBean Server
- Domain Runtime MBean Server
- JVM platform MBean server

The following sections explain how to configure collected metrics based policies using beans and functions:

- [Writing Collected Metrics Policy Expressions Using Beans](#)
- [Writing Collected Metrics Policy Expressions Using Functions](#)

Writing Collected Metrics Policy Expressions Using Beans

[Table 10-2](#) summarizes the beans provided by WebLogic Server. For complete reference information about each of these beans, see [WLDF Beans Reference](#).

Table 10-2 Beans Provided by WebLogic Server

Name	Prefix	Scope	Summary
runtime	wls	Only available from partition scope diagnostic system module deployments and partitions	Provides access to MBeans in the local WebLogic Server Runtime MBean Server. These MBeans include both the read-only configuration MBean and RuntimeMBean instances.
domainRuntime	wls	Administration Server	Provides access to MBeans on the Domain Runtime MBean Server (Administration Server only).
clusterRuntime	wls	Administration Server	Provides domain-wide access to cluster member data (Administration Server only).

Table 10-2 (Cont.) Beans Provided by WebLogic Server

Name	Prefix	Scope	Summary
platform	wls	Administration Server or Managed Server	Provides access to the JVM's platform MBean server. Note that in the majority of cases, the <code>platform</code> bean is functionally equivalent to the <code>runtime</code> bean: WebLogic Server uses the JVM's platform MBean server to contain the WebLogic run-time MBeans by default.
partition	wls	Partition scope WLDF diagnostic system module deployments.	Provides access to partition scope metrics. This bean is available only to policies that are configured in a diagnostic system module that is deployed in the same partition to which this bean is scoped.
resource	n/a	Administration Server, Managed Server, and partitions	Provides access to beans and state information within a diagnostic system module deployment. Access is restricted to policies that are configured within the same diagnostic system module.

 **Note:**

WebLogic Server Multitenant domain partitions, resource groups, resource group templates, and virtual targets are deprecated in WebLogic Server 12.2.1.4.0 and will be removed in the next release.

- [Accessing MBean Data in Collected Metrics](#)
- [Working with Complex MBean Attributes](#)
- [Performing Bulk Queries on Collected Metrics from MBeans](#)

Accessing MBean Data in Collected Metrics

The beans described in [Table 10-2](#) provide access to WebLogic Server Runtime MBean metrics. In policy expressions that use Java EL, metric data from each of these runtime MBeans is accessed using a WLDF-provided bean using the following syntax:

```
wls.bean-name.attribute-or-operation.attribute-or-operation...
```

All EL-based policy expressions that use the WLDF beans must begin with the namespace prefix `wls`. The prefix `wls` is similar to a namespace that contains all the

WLDF beans that can be used in the policy expressions. Beans and their attributes and methods are accessed using standard JavaBean conventions. The following example shows a simple policy expression that returns `true` when the value of `HeapFreePercent` attribute of `JVMRuntimeMBean` is less than 20:

```
wls.runtime.serverRuntime.JVMRuntime.heapFreePercent < 20
```

The preceding policy expression example accesses the value of `HeapFreePercent` in the following sequence:

1. The `runtime` bean is accessed from the `wls` bean namespace.
The `runtime` bean provides an entry point into the metrics collected by the local runtime MBean and also into the read-only configuration MBean data in the WebLogic Server Runtime MBean Server.
2. The `serverRuntime` attribute is accessed from the `runtime` bean.
The `serverRuntime` attribute of the `runtime` bean corresponds directly to the `ServerRuntimeMBean` instance in the local running server instance wherever the expression is being evaluated.
3. The `JVMRuntime` attribute, which corresponds to the `JVMRuntimeMBean` instance under the local `ServerRuntimeMBean`, is accessed from the `serverRuntime` bean.
4. The `heapFreePercent` attribute is accessed from the returned `JVMRuntime` instance.

From the `runtime` bean, runtime metrics and monitoring data are available through the `serverRuntime` attribute, and the `domain` attribute provides access to the current configuration data in the local read-only `DomainMBean` tree. This access allows policies to examine the current in-memory configuration within a policy expression.

MBeans that are accessed as bean attributes from the WLDF-provided expression beans have read-only access to most of the attributes and some operations available to the expression as defined in the [MBean Reference for Oracle WebLogic Server](#), with some exceptions for security purposes.

Note:

There are slight differences in syntax between JMX and JavaBean conventions when accessing attributes. For example, JavaBean conventions for accessing the JMX attribute `HeapFreePercent` require using “camel-case” syntax. When using JMX, the attribute is accessed by the name `HeapFreePercent`. However, in EL expressions, the same attribute is accessed as `heapFreePercent`.

Working with Complex MBean Attributes

Some MBean attributes return complex objects; for example, the `HealthState` attribute of the `ServerRuntimeMBean`. Such attributes can be accessed using JavaBean conventions. In the following example, the policy expression returns `true` if the health state of the server is a non-zero value:

```
wls.runtime.serverRuntime.healthState.state != 0
```

Working with Array Attributes

Many WLDF bean attributes return arrays of child MBeans. To work with collections, such as arrays, Java EL provides the `stream` operator to convert arrays and lists into stream objects that can be fed into other Java EL and WLDF functions and operators. In the following example, the policy expression examines the state of all `JDBCDataSourceRuntimeMBean` instances in the local server instance, and returns `true` if any of them are in the `Overloaded` state:

```
wls.runtime.serverRuntime.JDBCServiceRuntime.JDBCDataSourceRuntimeMBeans.stream().anyMatch(ds -> ds.state == "Overloaded")
```

The policy expression executes in the following sequence:

1. The `JDBCServiceRuntimeMBean` child is accessed from the `ServerRuntimeMBean`.
2. The array attribute `JDBCDataSourceRuntimeMBeans` is accessed from the `JDBCServiceRuntimeMBean`.
3. The Java EL `stream` operator is utilized to convert the array to a stream so that it can be used with WLDF and standard Java EL collection operations.
4. The `anyMatch` collection operation is used to look for the `Overloaded` state on any of the returned `JDBCDataSourceRuntimeMBean` instances.
5. If the `anyMatch` operation matches the `Overloaded` state, returns `true`.

Performing Bulk Queries on Collected Metrics from MBeans

The MBeans defined in [Table 10-2](#) are used in collected metrics policy expressions. All of these beans support a `query` method that allows to perform a query for a set of MBean attribute values against a homogeneous set of MBeans.

The method takes the following syntax:

```
query(target-list, object-name-pattern, attribute-expression)
```

The `query` method returns an iterable list of values that is obtained using the `attribute-expression` on each matching MBean instance.

Table 10-3 Method Parameters

Parameter	Description
<i>target-list</i>	This argument is applicable only for <code>domainRuntime</code> bean which is available only for policies executing on the Administration Server. The bean supports an overloaded variant that takes an array of targets. It is a list of servers or clusters in the domain. The argument allows the policy expression to examine MBean values across the domain in the same expression.
<i>object-name-pattern</i>	This argument takes any valid JMX ObjectName pattern that is specified as a string value enclosed by single quote (') characters. For example: 'com.bea:Type=ServletRuntime,*'

Table 10-3 (Cont.) Method Parameters

Parameter	Description
<i>attribute-expression</i>	<p>This argument is a quoted EL subexpression that is used to access an attribute from each of the MBeans matching the <i>object-name-pattern</i> argument. The <i>attribute-expression</i> argument can be either of the following types:</p> <ul style="list-style-type: none"> • A simple attribute available on the MBean. • An attribute of a complex type that uses a JavaBean-style expression to access the values within that complex structure. <p>Note: It is expected that <i>attribute-expression</i> ultimately resolves to a single scalar value, and not a complex structure.</p>

The values returned by the `query` method can be used as a part of the larger policy expression that examines those values.

 **Note:**

The intended use of the `query` method is to operate against a homogeneous set of MBean instances, but there is no enforcement mechanism to ensure that the specified MBeans must all be of the same type. Therefore, if you do specify an *object-name-pattern* that encompasses MBeans of different types, errors can result when the policy expression is evaluated.

Example 10-2 Examples of Using the query Method

Table 10-4 lists some examples of using the `query` method in policy expressions.

 **Note:**

The examples show how to use the `query` method and are not complete policy expressions.

Table 10-4 query Method Examples

Example	Description
<code>wls.runtime.query('com.bea:Type=ServletRuntime,*', 'ExecutionTimeAverage')</code>	The <code>query</code> method is used for all the instances of <code>ServletRuntimeMBean</code> in the local server and returns the value of <code>ExecutionTimeAverage</code> for each instance in the returned iterable stream.

Table 10-4 (Cont.) query Method Examples

Example	Description
<pre>wls.domainRuntime.query(['cluster1'], 'com.bea:Type=ThreadPoolRuntime,*', 'PendingUserRequestCount')</pre>	The domainRuntime bean is used to query all values of PendingUserRequestCount across all instances of ThreadPoolRuntimeMBean in the cluster cluster1. Any values found are returned in the Iterable set returned by the method call.

The use of query method in policy expression and the result set are represented in the following illustration:

Figure 10-1 Result Set of query in Policy Expression



The following is a complete example of a policy expression that uses the query method to determine whether the StuckThreadCount attribute on any WorkManagerRuntimeMBean in the local WebLogic Server instance is greater than zero:

```
wls.runtime.query('com.bea:Type=WorkManagerRuntime,*',
'StuckThreadCount').stream().anyMatch( x -> x > 0 )
```

The values of StuckThreadCount for all instances of WorkManagerRuntimeMBean are queried, and each value is examined to see if it is greater than zero, which indicates a stuck thread in the server. The stream collection operation is part of the Java EL standard, and is used for converting an iterable set into a stream that can be used with Java EL collection operations, such as anyMatch in the example.

Writing Collected Metrics Policy Expressions Using Functions

In addition to the bundled functions and collection operations that come with Java EL by default, there are also a set of WLDF-provided functions for use within policy expressions for common operations with metric data and for retaining a set of metrics with history.

The set of WLDF-provided functions includes:

- `wls:tableChanges`

- `wls:tableAverages`
- `wls:extract`
- `wls:average`
- `wls:changes`
- `wls:aliveServersCount`

For complete details about each EL function provided by WLDF, see [Functions Reference](#).

Functions are invoked using the prefix `wls:`

```
wls:<function-call>
```

For example, `wls:aliveServersCount('cluster1')` invokes the `aliveServersCount()` function provided by WLDF for the cluster `cluster1`.

Collection Operations

WLDF also provides a set of collection operations that can be invoked similar to the collection operations provided by Java EL. The set of WLDF-provided collection operations includes:

- `tableAverages`
- `percenMatch`
- `collection`
- `flatten`
- [Examining Trends in Metric Values over Time](#)
- [Extracting and Examining Collected Metrics in Policy Expressions](#)
- [Lifecycle of Data Collection](#)

Examining Trends in Metric Values over Time

You can look for trends in metric data over time instead of assessing the instantaneous values. Use the `wls:extract` function to extract a table of time series from a specified set of input sources, based on a specified sampling rate schedule and time window.

The `extract` function has the following syntax:

```
wls:extract(sources, sampling rate, retention window)
```

The method returns an iterable set that consists of a two dimensional set of results. The metric input to the function comes from multiple MBean instances during the course of a specific interval of time defined by the `retention window` parameter. The resulting data is similar to a table where each row is a set of values from a particular MBean instance over the time window.

Parameters

Table 10-5 Parameters Description for `extract()` Function

Parameters	Description
<code>sources</code>	Set of metric sources, which can be identified as a <code>query</code> method or as a quoted Java EL expression.

Table 10-5 (Cont.) Parameters Description for extract() Function

Parameters	Description
sampling rate	String that identifies the frequency with which data is collected. You can specify this string as hours, minutes, or seconds. The syntax is flexible, allowing you to specify 30 seconds, for example, as "30s", "30sec", or "30 seconds". Note: The frequency only applies to the rate of collection of the metric, and is independent of the overall policy evaluation schedule.
retention window	String that identifies the retention window over which to observe values from the sources input with syntax identical to the <code>sampling rate</code> parameter. It implements the sliding window algorithm in which the oldest data in the set is aged out when the array is full. See retention window .

Example 10-3 Examples of Using the extract Function

[Table 10-6](#) lists example usages of the `extract` function.

 **Note:**
 The examples show how to invoke the `extract` function and are not complete policy expressions.

Table 10-6 extract Function Examples

Example	Description
<code>wls:extract("wls.runtime.serverRuntime.threadPoolRuntime.pendingUserRequestCount", "30s", "2m")</code>	The <code>extract</code> function is invoked with an EL expression as the first argument which observes and collects the values of the <code>PendingUserRequestCount</code> attribute on the <code>ThreadPoolRuntimeMBean</code> at 30-second intervals and retains them over a period of 2 minutes. In this example, <code>ThreadPoolRuntimeMBean</code> is a singleton, and only the local WebLogic Server instance is monitored. Therefore, only a single row of values is returned in the table of values.
<code>wls:extract(wls.runtime.query("com.bea:type=ThreadPoolRuntime,*", "PendingUserRequestCount"), "30s", "2m")</code>	The <code>extract</code> function is used with the result of a query method invocation as input.

Table 10-6 (Cont.) extract Function Examples

Example	Description
<pre>wls:extract(wls.domainRuntime.query(['cluster1'], 'com.bea.Type=ThreadPoolRuntime,*', 'PendingUserRequestCount'), '30s', '2m')</pre>	<p>The <code>extract</code> function is used with the <code>query</code> method of the <code>domainRuntime</code> bean to collect the value of <code>PendingUserRequestCount</code> attribute on all <code>ThreadPoolRuntimeMBean</code> instances on every server in <code>cluster1</code>. The result set for this call consists of a row of values for each <code>ThreadPoolRuntimeMBean</code> instance in each active server instance in <code>cluster1</code>.</p>

Extracting and Examining Collected Metrics in Policy Expressions

The `extract` function returns a table of scalar values. You can use any collection operation to examine or manipulate the result set. WLDF provides more collection operations that are intended for use with the data returned from `extract` function, such as `tableAverages`, `percentMatch`, `collection`, and `flatten`.

Operations	Description
<code>tableAverages</code>	Computes the average value for each row in the table.
<code>percentMatch</code>	Examines all the computed averages from <code>tableAverages</code> .
<code>collection</code>	Returns the two dimensional set of values in tabular form, which can be then converted to Java EL collection stream using the <code>stream</code> operator and can be directly manipulated in other Java EL collection operators.
<code>flatten</code>	Converts the two dimensional set of values returned by <code>extract</code> function into a linear collection of values.

The result of `extract` can then be fed into other functions or operations as part of an overall policy expression.

In the following example of a policy expression, the `extract` function collects the value for the `PendingUserRequestCount` attribute across the servers in `cluster1`. The result is combined with the `tableAverages` and `percentMatch` collection operations to produce a boolean value.

```
wls:extract(wls.domainRuntime.query({'cluster1'},
'com.bea.Type=ThreadPoolRuntime,*', 'PendingUserRequestCount'), '30s',
'2m').tableAverages().stream().percentMatch(pendingCount -> pendingCount > 100)
> 0.75
```

This policy expression returns `true` when the average value of the attribute `PendingUserRequestCount` over the 2-minutes window is greater than 100 on 75% of the servers in `cluster1`. The policy expression executes in the following sequence:

1. The `extract` function creates a table of values for the attribute `PendingUserRequestCount`, where each row is one set of values from a server in `cluster1` over a 2-minutes window.
2. The `tableAverages` operation computes the average value over the 2-minutes window for each row in the table returned by the `extract` function.

3. `stream` is a standard Java EL collection operation used to convert the vector result of `tableAverages` to a Java EL stream.
4. The `percentMatch` operation examines all the computed averages from `tableAverages`, and computes the percentage of values in that set that are greater than 100.
5. The result of `percentMatch` is a value between 0 and 1 and is compared with 0.75, the desired threshold.

Lifecycle of Data Collection

The `extract` function extracts data from a specified input source over a defined period of time. When the `extract` function is first encountered in an expression by the WLDF policy engine, it starts the collection of the desired metrics indicated in the policy expression. Samples are collected by the policy engine until the policy using the `extract` function is disabled or undeployed.

Policy expressions that use the `extract` function is not evaluated until enough data has been collected for the desired metrics to satisfy the sliding window interval specified in the invocation. If the function invocation specifies that a 5-minutes window of data is required, then 5 minutes of data collection must take place from the moment the policy is deployed before the expression can be successfully evaluated.

In the following example, the expression does not evaluate until 2 minutes of data for the `PendingUserRequestCount` attribute is collected.

```
wls:extract(wls.runtime.query("com.bea:Type=ThreadPoolRuntime,*",  
"PendingUserRequestCount"), "30s", "2m")
```

11

Configuring Actions

The WebLogic Diagnostics Framework (WLDF) provides several types of actions that can be executed when a policy evaluates to `true`, such as triggering an elastic scaling action, sending a JMS notification, executing an external command line script, and more.

For information about how to create an action using the WebLogic Server Administration Console, see [Create actions for policies in a diagnostic system module](#) in *Oracle WebLogic Server Administration Console Online Help*.

- [Actions Overview](#)
An action is an operation that is executed when a policy expression evaluates to true. WLDF supports different types of action based on the delivery mechanism of the notification.
- [Configuring JMX Actions](#)
WLDF issues JMX events when an associated policy is triggered for each defined JMX action. You can configure the JMX action to receive all the JMX notification and filter the output as required.
- [Configuring JMS Actions](#)
You can configure JMS actions to send JMS notifications through the JMS topics or queues when the corresponding policy is triggered. You can define how the notification must be delivered such as defining the destination and the connection factory.
- [Configuring SNMP Actions](#)
- [Configuring Log Actions](#)
You can create a log action to send a customized message to the server log.
- [Configuring REST Actions](#)
You can use a REST action to send a notification to a REST endpoint that includes a customized message in the notification payload. You can configure the REST endpoint invocation for no authentication or basic authentication.
- [Configuring SMTP Actions](#)
Simple Mail Transfer Protocol (SMTP) actions are used to send messages (e-mail) over the SMTP protocol in response to the triggering of an associated policy. You provide a list of recipients to whom the message is distributed through the configured SMTP session.
- [Configuring Image Actions](#)
An image action causes a diagnostic image to be generated in response to the triggering of an associated policy. You can configure two options for image actions: a directory and a lockout period.
- [Configuring Elastic Actions](#)
WLDF provides scale up and scale down elastic actions that can be performed on dynamic clusters.
- [Configuring Script Actions](#)
You can use the script action to execute an external command-line script. The script can be written in any scripting language.

- [Configuring Heap Dump Actions](#)
You can use a heap dump action to capture heap dumps when certain runtime conditions, defined by a policy expression, are met. Each heap dump is produced in HPROF format, which you can analyze with tools such as the `jmap` utility, which is available in the JDK.
- [Configuring Thread Dump Actions](#)
You can use a thread dump action to capture a specific number of thread dumps, separated by configured time interval, when the runtime conditions that are specified in a corresponding policy are met. Each thread dump file is produced in an individual text file.

Actions Overview

An action is an operation that is executed when a policy expression evaluates to true. WLDF supports different types of action based on the delivery mechanism of the notification.

Topics

The following sections contain background information pertaining to WLDF actions:

- [Types of Actions](#)
- [Variables for Customizable Actions](#)
The log, SMTP, and REST action types support the generation of customized strings that contain one or more of the variables listed in this topic.
- [Action Timeout](#)
All WLDF actions support a timeout, which determines the time, in seconds, for the action to complete execution. By default, the timeout is 0, which disables the action timeout.

Types of Actions

WLDF supports the following types of diagnostic actions, based on the delivery mechanism:

- Java Management Extensions (JMX)
- Java Message Service (JMS)
- Simple Network Management Protocol (SNMP)
- Simple Mail Transfer Protocol (SMTP)
- Diagnostic image capture
- Elasticity framework
- REST
- WebLogic logging system
- WebLogic Scripting Tool (WLST)
- Heap dump
- Thread dump

In the configuration file for a diagnostic module, the different types of actions are identified by the following elements in the `config.xml` file for the domain:

- <jmx-notification>
- <jms-notification>
- <snmp-notification>
- <smtp-notification>
- <image-notification>
- <scale-up-action>
- <scale-down-action>
- <rest-notification>
- <log-action>
- <script-action>
- <heap-dump-action>
- <thread-dump-action>

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

Variables for Customizable Actions

The log, SMTP, and REST action types support the generation of customized strings that contain one or more of the variables listed in this topic.

When a triggered policy invokes one of these action types, each variable used in the customized string that is generated by the action is replaced with the value shown in the following table.

Table 11-1 Substitution Variables

Variable Name	Value
WatchName	Name of policy that corresponds to the action
WatchRuleType	Policy type (for example, Harvester, Log, or EventData)
WatchRule	Policy expression
WatchTime	Timestamp identifying when the corresponding policy was triggered
WatchSeverityLevel	Policy severity option
WatchData	Log message
WatchAlarmType	Specifies the policy alarm type, which can be None, AutomaticReset, or ManualReset.
WatchAlarmResetPeriod	Alarm reset period configured on the WLDWatchNotificationRuntimeMBean.
WatchDomainName	WebLogic domain name
WatchServerName	Server instance name

Log, REST, and SMTP actions send different types of messages when executed. Each of these actions, while different, has one or more properties that support the use of one or more of the variables defined in . For example, an SMTP message body can be specified as follows to include the policy name, expression, and timestamp indicating when the policy was triggered:

```
"Test ${WatchName} with policy ${WatchRule} fired at ${WatchTime}."
```

For more information about using these substitution variables, see:

- [Configuring Log Actions](#)
- [Configuring REST Actions](#)
- [Configuring SMTP Actions](#)

Action Timeout

All WLDF actions support a timeout, which determines the time, in seconds, for the action to complete execution. By default, the timeout is 0, which disables the action timeout.

You can specify the action timeout using the `WLDFNotificationBean.Timeout` attribute.

You can also set the timeout when configuring an action in either the WebLogic Server Administration Console or Fusion Middleware Control. See the following topics:

- [Create actions for policies in a diagnostic system module](#) in *Oracle WebLogic Server Administration Console Online Help*
- [Configure an action in Administering Oracle WebLogic Server with Fusion Middleware Control](#)

Configuring JMX Actions

WLDF issues JMX events when an associated policy is triggered for each defined JMX action. You can configure the JMX action to receive all the JMX notification and filter the output as required.

For each defined JMX action, WLDF issues JMX events (notifications) whenever an associated policy is triggered. Applications can register an action listener with the server's `WLDFWatchNotificationSourceRuntimeMBean` to receive all JMX 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 11-1](#) shows an example of a JMX action configuration.

Example 11-1 Example Configuration for a JMX Action

```
<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/2.0/
weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <watch-notification>
    <!-- One or more policy configurations -->
    <jmx-notification>
      <name>myJMXNotif</name>
      <enabled>true</enabled>
```

```

    </jmx-notification>
    <!-- Other action configurations -->
  </watch-notification>
</wldf-resource>

```

Here is an example of a JMX action:

```

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

```

Configuring JMS Actions

You can configure JMS actions to send JMS notifications through the JMS topics or queues when the corresponding policy is triggered. You can define how the notification must be delivered such as defining the destination and the connection factory.

In the system resource configuration file, the elements `<destination-jndi-name>` and `<connection-factory-jndi-name>` define how the notification is to be delivered.

[Example 11-2](#) shows two JMS actions that cause JMS notifications 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 11-2 Example JMS Actions

```

<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/2.0/
weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <watch-notification>
    <!-- One or more policy 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 action configurations -->
  </watch-notification>
</wldf-resource>

```

The content of the action message gives details of the policy and action.

Configuring SNMP Actions

Simple Network Management Protocol (SNMP) actions are used to post SNMP traps when an associated policy is triggered. Provide the action name to define an SNMP action. To define an SNMP action, provide the action name as shown in [Example 11-3](#). Generated traps contain the names of both the policy and action 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 11-3 An Example Configuration for an SNMP Action

```
<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/2.0/
weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <watch-notification>
    <!-- One or more policy configurations -->
    <snmp-notification>
      <name>mySNMPNotif</name>
    </snmp-notification>
    <!-- Other action configurations -->
  </watch-notification>
</wldf-resource>
```

The trap resulting from the SNMP action configuration shown in [Example 11-3](#) is of type 85. It contains the following values (configured values are shown in angle brackets "<>"):

```
.1.3.6.1.4.1.140.625.100.5   timestamp (e.g. Dec 9, 2004 6:46:37 PM EST)
.1.3.6.1.4.1.140.625.100.145  domainName (e.g. mydomain)
.1.3.6.1.4.1.140.625.100.10   serverName (e.g. myserver)
.1.3.6.1.4.1.140.625.100.120  <severity> (e.g. Notice)
.1.3.6.1.4.1.140.625.100.105  <name> [of watch] (e.g.
    simpleWebLogicMBeanWatchRepeatingAfterWait)
.1.3.6.1.4.1.140.625.100.110  <rule-type> (e.g. HarvesterRule)
.1.3.6.1.4.1.140.625.100.115  <rule-expression>
.1.3.6.1.4.1.140.625.100.125  values which caused rule to
    fire (e.g..State =
    null,weblogic.management.runtime.WLDFHarvesterRuntimeMBean.
    TotalSamplingTime = 886,.Enabled =
    null,weblogic.management.runtime.ServerRuntimeMBean.
OpenSocketsCurrentCount = 1,)
.1.3.6.1.4.1.140.625.100.130  <alarm-type> (e.g. None)
.1.3.6.1.4.1.140.625.100.135  <alarm-reset-period> (e.g. 10000)
.1.3.6.1.4.1.140.625.100.140  <name> [of notification]
    (e.g.mySNMPNotif)
```

Configuring Log Actions

You can create a log action to send a customized message to the server log.

The customized message can optionally include any of the variables described in [Variables for Customizable Actions](#). The following WLST example shows the configuration of a log action:

```
wn=res.getWatchNotification()
```

```

actionName="myaction"
action = wn.lookupLogAction(actionName);
if action is None:
    action = wn.createScriptAction(actionName);
action.setMessage("Message with substitution on server ${WatchServerName} in domain ${WatchDomainName}");
action.setSubsystemName("SpecialLogAction");
action.setSeverity("Info");

```

When the preceding log action is executed, the custom message, shown in **bold**, uses variables to identify:

- The WebLogic Server instance name, represented by the `${WatchServerName}` variable
- The WebLogic domain name, represented by the variable `${WatchDomainName}`

Configuring REST Actions

You can use a REST action to send a notification to a REST endpoint that includes a customized message in the notification payload. You can configure the REST endpoint invocation for no authentication or basic authentication.

When configuring a REST action, you can create a customized set of notification properties that can optionally use any of the variables described in [Variables for Customizable Actions](#). For example, the following WLST example shows the configuration of a REST action that sends a customized message:

```

wn = res.getWatchNotification();

#No Auth REST invocation
rest1 = wn.createRESTNotification('r1')
rest1.setEndpointURL("http://localhost:7001/rest-no-auth/resources/watch-listener")
customNotif = java.util.Properties()
customNotif.put('message','Policy ${WatchName} with rule ${WatchRule} fired.')
rest1.setCustomNotificationProperties(customNotif)
rest1.setEnabled(true)

#Basic Auth REST invocation
rest2 = wn.createRESTNotification('r2')
rest2.setEndpointURL("http://localhost:7001/rest-basic-auth/resources/watch-listener")
rest2.setHttpAuthenticationMode('Basic')
rest2.setHttpAuthenticationUserName('restuser1')
rest2.setHttpAuthenticationPassword('restuser1')
rest2.setEnabled(true)

```

When the preceding REST action is executed, the REST endpoint is invoked with a message, shown in **bold**, that identifies:

- The name of the triggered policy that executed the corresponding REST action, represented by the `${WatchName}` variable
- The policy expression, represented by the `${WatchRule}` variable

Configuring SMTP Actions

Simple Mail Transfer Protocol (SMTP) actions are used to send messages (e-mail) over the SMTP protocol in response to the triggering of an associated policy. You provide a list of recipients to whom the message is distributed through the configured SMTP session.

To define an SMTP action, 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 subelement `<mail-session-jndi-name>`, and provide a list of at least one recipient using subelement `<recipients>`. An optional subject and/or body can be provided using subelements `<subject>` and `<body>` respectively. If these are not provided, they will be defaulted.

[Example 11-4](#) shows an SMTP action that causes an SMTP (e-mail) message to be distributed through the configured SMTP session, to the configured recipients. In this action configuration, a custom subject and body are provided. If a subject or body are not specified, defaults are provided, showing details of the policy and action.

Example 11-4 Sample Configuration for SMTP Action (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/2.0/weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <watch-notification>
    <!-- One or more policy 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 action configurations -->
  </watch-notification>
</wldf-resource>
```

The content of the action message gives details of the policy and action.

WLDF also supports customizing the subject and body elements in the sent email by using any of the variables described in [Variables for Customizable Actions](#).

The following WLST example shows the configuration of an SMTP action that contains customized subject and body text. The subject and body of the message utilize variables to specify the policy name and the timestamp indicating when the policy was triggered:

```
smtp=wn.lookupSMTPNotification('smtp1')
if smtp is None:
  smtp=wn.createSMTPNotification('smtp1')

smtp.setMailSessionJNDIName('test.MailSession')
smtp.setSubject("WatchRule ${WatchName} alert")
smtp.setBody("Test ${WatchName} with rule ${WatchRule} fired at ${WatchTime}.")
smtp.setRecipients(["john.smith@example.com"])
```

When the preceding SMTP action is executed, an email is generated with a custom subject and body, shown in bold, that identifies:

- The name of the policy that executed the SMTP action, represented by the variable `${WatchName}`. This variable is used in both the subject and body.
- The policy expression, represented by the `${WatchRule}` variable

- The timestamp identifying when the corresponding policy was triggered, represented by the `#{WatchTime}` variable

Configuring Image Actions

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

The directory name indicates where the 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_HOME\servers\SERVER_NAME`. The default directory is `DOMAIN_HOME\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 an action can execute many times, resulting in a separate image file each time.

The configuration is persisted in the `DIAG_MODULE.xml` configuration file. [Example 11-5](#) shows an image action configuration that specifies that the lockout time will be two minutes and that the image will be generated to the `DOMAIN_HOME\servers\SERVER_NAME\images` directory.

Example 11-5 Sample Configuration for Image Action (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/2.0/
weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <watch-notification>
    <!-- One or more policy configurations -->
    <image-notification>
      <name>myImageNotif</name>
      <enabled>true</enabled>
      <image-lockout>2</image-lockout>
      <image-directory>images</image-directory>
    </image-notification>
    <!-- Other action configurations -->
  </watch-notification>
</wldf-resource>
```

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

Configuring Elastic Actions

WLDF provides scale up and scale down elastic actions that can be performed on dynamic clusters.

- scale up — Configured using the [WLDfScaleUpActionBean](#)
- scale down — Configured using the [WLDfScaleDownActionBean](#)

Each action bean has the following configuration attributes:

- `clusterName` — The name of the dynamic cluster that needs to be scaled
- `scalingSize` — The number of Managed Server instances by which the dynamic cluster needs to be scaled up or down

The scale up and scale down actions attempt to scale the dynamic cluster specified by the `clusterName` parameter, by the number of servers specified as the `scalingSize` value. WLDF interacts with the elasticity framework to scale the dynamic cluster accordingly.

 **Note:**

Note the following:

- To configure automated elasticity for a dynamic cluster, you must create a domain-scope diagnostic system module in which you define the scaling policy, along with its corresponding elastic action, and then target that diagnostic module to the Administration Server.
- After a scale up or scale down action has been invoked, the scaling action can't be subsequently cancelled.

The following WLST snippet shows the commands for configuring a scale up action. In this example, the dynamic cluster `myCluster` is scaled up by one Managed Server instance:

```
wn=res.getWatchNotification()

scaleUp=wn.lookupScaleUpAction('scaleUp')
if scaleUp == None:
    print "Creating scale up action"
    scaleUp=wn.createScaleUpAction('scaleUp')
scaleUp.setScalingSize(1)
scaleUp.setClusterName(myCluster)
```

The following example shows the WLST commands for configuring a scale down action on `myCluster`:

```
wn=res.getWatchNotification()

scaleDown=wn.lookupScaleDownAction('scaleDown')
if scaleDown == None:
    print "Creating scale down action"
    scaleDown=wn.createScaleDownAction('scaleDown')
scaleDown.setScalingSize(1)
scaleDown.setClusterName(myCluster)
```

For complete details about using these elastic actions, see:

- Elastic Actions in *Configuring Elasticity in Dynamic Clusters for Oracle WebLogic Server*
- Expanding or Reducing Dynamic Clusters in *Administering Clusters for Oracle WebLogic Server*
- [Elastic Scaling Operations Cannot Be Cancelled After Starting](#)
Note that the moment a scaling operation has begun, regardless of whether it is a scale up or scale down operation, it cannot be cancelled. If you configure

automated elasticity in a dynamic cluster, such as with calendar-based or policy-based scaling, the elasticity framework does not provide the means to cancel a scaling operation after it has been initiated.

- [Limiting Server Shutdown Time During Scale Down Operations](#)
Shutting down servers during a scale down operation can take a significant amount of time, especially if there are unreplicated sessions. Until unreplicated sessions time out, which can potentially be a long time, the server will not be shut down.

Elastic Scaling Operations Cannot Be Cancelled After Starting

Note that the moment a scaling operation has begun, regardless of whether it is a scale up or scale down operation, it cannot be cancelled. If you configure automated elasticity in a dynamic cluster, such as with calendar-based or policy-based scaling, the elasticity framework does not provide the means to cancel a scaling operation after it has been initiated.

Consequently, if a postprocessor script (invoked by a script interceptor) fails, the parts of the scaling operation that were completed can't be reverted. For more information about script interceptors and postprocessor scripts, see *Configuring the Script Interceptor in Configuring Elasticity in Dynamic Clusters for Oracle WebLogic Server*.

Limiting Server Shutdown Time During Scale Down Operations

Shutting down servers during a scale down operation can take a significant amount of time, especially if there are unreplicated sessions. Until unreplicated sessions time out, which can potentially be a long time, the server will not be shut down.

To limit the length of time required to complete a scale down operation, you can configure the following attributes on the [DynamicServersMBean](#):

Attribute	Description
DynamicClusterShutdownTimeoutSeconds	Timeout period, in seconds, to use while gracefully shutting down a dynamic server instance. If the dynamic server instance does not shut down before the specified timeout period, then it will be forcibly shut down. The default value is 0.
IgnoreSessionsDuringShutdown	Specifies whether to ignore inflight HTTP requests while shutting down dynamic server instances.
WaitForAllSessionsDuringShutdown	Specifies whether to wait for all persisted and nonpersisted inflight HTTP sessions to complete before shutting down dynamic server instances.

By specifying a timeout or ignoring inflight HTTP sessions during shutdown, the shutdown time can be limited. However, note that remaining inflight HTTP sessions may be lost.

Configuring Script Actions

You can use the script action to execute an external command-line script. The script can be written in any scripting language.

To set the execution environment in which the script is run, you can configure the following attributes of the [WLDFScriptActionBean](#):

- [PathToScript](#) — The full path to the script, which must be located in the `DOMAIN_HOME/bin/scripts` directory
- [WorkingDirectory](#) — The directory from which the WebLogic Server process was run, which is typically the domain root directory.
- [Environment](#) — A map of environment variables to set for the child process
- [Parameters](#) — An array of parameters or command options to pass to the script
- [Timeout](#) — The time, in seconds, for the script action to complete execution. By default, the timeout is 0, which disables the script action timeout.

When the script action is executed by a triggered policy, WLDF invokes the configured script, which is run with the identity of the configured script. The script process executes as a child process of the WebLogic Server process that spawned it. Therefore, the script process has the same operating system identity as the WebLogic Server process; however, it does not inherit any of the parent process environment.

The following example shows configuring a script action using WLST:

```
wn=res.getWatchNotification()

actionName="myaction"
action = wn.lookupScriptAction(actionName);
if action is None:
    action = wn.createScriptAction(actionName);

action.setWorkingDirectory("somedir");
action.setPathToScript("myScript.sh");
action.setParameters(["param1", "param2"]);
action.setTimeout(300);
```

Configuring Heap Dump Actions

You can use a heap dump action to capture heap dumps when certain runtime conditions, defined by a policy expression, are met. Each heap dump is produced in HPROF format, which you can analyze with tools such as the `jmap` utility, which is available in the JDK.

You create a heap dump action by configuring the [WLDFHeapDumpActionBean](#) and the [WLDFServerDiagnosticMBean](#) in a domain scope diagnostic system module – that is, a diagnostic system modules that is deployed in the domain partition. You cannot use this action in a diagnostic system module that is scoped to a partition. When configuring a heap dump action, you can specify the following:

- Whether or not to include only objects that can be referenced (that is, not garbage-collected, or awaiting garbage collection), which you specify in the [LiveSetOnly](#) attribute of the [WLDFHeapDumpActionBean](#). The default value is `true`.
- The location each server's diagnostic dumps directory where the heap dumps are stored. You can specify this directory in the [DiagnosticDumpsDir](#) attribute of the [WLDFServerDiagnosticMBean](#).
- The number of heap dump files that are retained, which prevents filling up the file system with generated heap dumps. You can specify the number in the [MaxHeapDumpCount](#) attribute of the [WLDFServerDiagnosticMBean](#). The default value is 8.

The generated heap dump files are named using the following syntax:

```
HeapDump_$$SERVER_$$MODULE_$$POLICY_$$ACTION_$$timestamp.hprof
```

In the preceding syntax:

- `$$SERVER` represents the name of the WebLogic Server instance that generated the heap dump.
- `$$MODULE` represents the name of the diagnostics system module that contains the action configuration.
- `$$POLICY` represents the name of the policy that executed the heap dump action.
- `$$ACTION` represents the name of the `WLDFHeapDumpActionBean`.
- `$$timestamp` represents time when the heap dump was generated, which takes the form of `yyyy_mm_dd_HH_MM_SS`.

Note:

Note the following:

- Heap dumps may contain sensitive information. Therefore, make sure that you place appropriate access protections on the directories into which heap dumps are generated.
- If a heap dump action is in progress, an attempt by another heap dump action to generate a heap dump is rejected and a message is generated in the server log.

For information about how to create and configure a heap dump action using the WebLogic Server Administration Console, see [Create a heap dump action](#) and [Configure a heap dump action](#) in *Oracle WebLogic Server Administration Console Online Help*.

The `jmap` utility is described in the Java SE 8 documentation, available at <http://docs.oracle.com/javase/8/>.

Example 11-6 An Example Configuration for a Heap Dump Action

The following WLST example shows the configuration of a heap dump action:

```
# Start an edit session in edit tree
edit()
startEdit()
cd("/")

if cmo.lookupWLDFSystemResource("mywldf") == None:
    print "Creating WLDF resource"
    cmo.createWLDFSystemResource("mywldf")

cd("/WLDFSystemResources/mywldf/WLDFResource/mywldf/WatchNotification/mywldf")

# Create a heap dump action
cmo.createHeapDumpAction('myHeapDump')
cd("HeapDumpActions/myHeapDump")
# Set it to capture a full heap, not just the live setLiveSetOnly - default is "true"
cmo.setLiveSetOnly(false)

save()
activate()
```

Configuring Thread Dump Actions

You can use a thread dump action to capture a specific number of thread dumps, separated by configured time interval, when the runtime conditions that are specified in a corresponding policy are met. Each thread dump file is produced in an individual text file.

You create a thread dump action by configuring the `WLDFThreadDumpActionBean` and the `WLDServerDiagnosticMBean` in a domain scope diagnostic system module – that is, a diagnostic system modules that is deployed in the domain partition. You cannot use this action in a diagnostic system module that is scoped to a partition. When configuring a thread dump action, you specify the following:

- The number of thread dumps to be captured, which you specify in the `ThreadDumpCount` attribute of the `WLDFThreadDumpActionBean`. The default value is 3.
- The interval between successive thread dumps, which you specify in the `ThreadDumpDelaySeconds` attribute of the `WLDFThreadDumpActionBean`. The default value is 10 seconds.
- The location each server's diagnostic dumps directory where the thread dumps are stored, which you can specify with the `DiagnosticDumpsDir` attribute of the `WLDServerDiagnosticMBean`.
- The number of thread dump files that are retained, which prevents filling up the file system with generated thread dumps. You specify the number using the `MaxThreadDumpCount` attribute of the `WLDServerDiagnosticMBean`. The default value is 100.

The generated thread dump files are named using the following syntax:

```
HeapDump_{$SERVER_}$MODULE_{$POLICY_}$ACTION_{$timestamp}.hprof
```

In the preceding syntax:

- `{$SERVER_}` represents the name of the WebLogic Server instance that generated the thread dump.
- `{$MODULE_}` represents the name of the diagnostics system module that contains the action configuration.
- `{$POLICY_}` represents the name of the policy that executed the thread dump action.
- `{$ACTION_}` represents the name of the `WLDFThreadDumpActionBean`.
- `{$timestamp}` represents time when the thread dump was generated, which takes the form of `yyyy_mm_dd_HH_MM_SS`.

 **Note:**

Note the following:

- Thread dumps may contain sensitive information. Therefore, make sure that you place appropriate access protections on the directories into which thread dumps are generated.
- If a thread dump action is in progress, an attempt by another thread dump action to generate a thread dump is rejected and a message is generated in the server log.

For information about how to create and configure a thread dump action using the WebLogic Server Administration Console, see [Create a thread dump action](#) and [Configure a thread dump action](#) in *Oracle WebLogic Server Administration Console Online Help*.

Example 11-7 An Example Configuration for a Thread Dump Action

The following WLST example shows the configuration of a thread dump action:

```
# Start an edit session in edit tree
edit()
startEdit()
cd("/")

if cmo.lookupWLDFSystemResource("mywldf") == None:
    print "Creating WLDF resource"
    cmo.createWLDFSystemResource("mywldf")

cd("WLDFSystemResources/mywldf/WLDFResource/mywldf/WatchNotification/mywldf")

# Create a Thread Dump action
cmo.createThreadDumpAction('myThreadDump')
cd("ThreadDumpActions/myThreadDump")

# set it to capture 5 dumps at 30 second intervals
cmo.setThreadDumpCount(5)
cmo.setThreadDumpDelaySeconds(30)

save()
activate()
```

Configuring Instrumentation

The Instrumentation component of the WebLogic Diagnostics Framework (WLDF) 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 [Configuring the DyelInjection 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.

The following sections introduce the Instrumentation components and explain how to configure them and also the different kinds of diagnostic monitors and actions:

- [Concepts and Terminology](#)

Learn a comprehensive list of common terms and some basic concepts that apply to the Instrumentation component of WLDF.

- [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.

- [XML Elements Used for Instrumentation](#)

You can configure instrumentation and diagnostic monitors using the XML elements such as `<Instrumentation>` and `<wldf-instrumentation-monitor>`.

- [Configuring Server-Scoped Instrumentation](#)

You can configure instrumentation as part of diagnostic descriptor file to implement the system-level instrumentation. You can define the configuration of one or more server-scoped diagnostic monitors in the descriptor file.

- [Configuring Application-Scoped Instrumentation](#)

Instrumentation is the only component that is deployable to applications. It must be enabled on the server to which the application is deployed. You can enable and disable diagnostic monitors without redeploying an application.

- [Creating Request Performance Data](#)
You can display request performance data in the WebLogic Server Administration Console for configured server-scoped or application-scoped instrumentation. In the console, the Request Performance page displays information about the real-time and historical views of method performance.

Concepts and Terminology

Learn a comprehensive list of common terms and some basic concepts that apply to the Instrumentation component of WLDF.

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

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 Server instances. The scope is built in to each diagnostic monitor; you cannot modify a monitor's scope.

Configuration and Deployment

Server-scoped instrumentation for a server or cluster is configured and deployed as part of a diagnostic module, an XML configuration file located in the `DOMAIN_HOME/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.

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 [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>.

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. See [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 WebLogic Server 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 [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 12-1](#) summarizes the differences among the types of monitors.

Table 12-1 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 [<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 [Configuring the DyeInjection Monitor to Manage Diagnostic Contexts](#).

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`
- `MemoryAllocationStatisticsAction`
- `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 [WLDF Instrumentation Library](#), for more information.

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.

The Instrumentation component is configured as follows:

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

```
DOMAIN_HOME/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
- Policies and Actions

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_HOME/config/config.xml
```

See [Configuring Diagnostic System Modules](#). For general information about the creation, content, and parsing of configuration files in WebLogic Server, see Domain Configuration Files in *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 [Using Deployment Plans to Dynamically Control Instrumentation Configuration](#).

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

The diagnostics XML schema is located at:

```
http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/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 [Understanding WLDF Configuration](#).

XML Elements Used for Instrumentation

You can configure instrumentation and diagnostic monitors using the XML elements such as `<Instrumentation>` and `<wldf-instrumentation-monitor>`.

This section provides descriptor fragments and tables that summarize information about the XML elements used to configure:

- [<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.
- [<Instrumentation> XML Elements](#)
- [<wldf-instrumentation-monitor> XML Elements](#)
- [Mapping <wldf-instrumentation-monitor> XML Elements to Monitor Types](#)

<Instrumentation> XML Elements

[Table 12-2](#) 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>example.com.*</include>
    <!-- <wldf-instrumentation-monitor> elements to define diagnostic
         monitors for this diagnostic module -->
</instrumentation>
<!-- Other elements to configure this diagnostic module -->
</wldf-resource>

```

Table 12-2 <instrumentation> XML Elements in the DIAG_MODULE.xml Configuration File

Element	Description
<instrumentation>	The element that begins an instrumentation configuration.
<enabled>	<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>
<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 12-1.</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>

<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.example. * 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 [Configuring the DyeInjection Monitor to Manage Diagnostic Contexts](#), for information about configuring the *DyeInjection* monitor.

[Table 12-3](#) describes the <wldf-instrumentation-monitor> elements.

Table 12-3 <wldf-instrumentation-monitor> XML Elements in the *DIAG_MODULE.xml* or *weblogic-diagnostics.xml* file

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

Table 12-3 (Cont.) <wldf-instrumentation-monitor> XML Elements in the DIAG_MODULE.xml or weblogic-diagnostics.xml file

Element	Description
<name>	The name of the monitor. For standard and delegating monitors, use the names of the predefined monitors in 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.
<description>	An optional element describing the monitor.
<action>	An optional element, which applies to delegating and custom monitors. If you do not specify at least one action, the monitor will not generate any information. You can specify multiple <action> elements. An action must be compatible with the monitor type. For the list of predefined actions for use by delegating and custom monitors, see WLDF Instrumentation Library .
<dye-filtering-enabled>	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 DyeInjection monitor must be configured appropriately at the server level.
<dye-mask>	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 Configuring the DyeInjection Monitor to Manage Diagnostic Contexts , for information about dyes and dye filtering.
<properties>	An optional element. Sets name=value pairs for dye flags. Currently applies only to the DyeInjection monitor.
<location-type>	An optional element, whose value is one of before, after, or around. The location type determines when an action is triggered at a pointcut: before the pointcut, after the pointcut, or both before and after the pointcut. Applies only to custom monitors; standard and delegating monitors have predefined location types. A custom monitor must define a location type and a pointcut.
<pointcut>	An optional element. A pointcut element contains an expression that defines joinpoints where diagnostic code will be inserted. Applies only to custom monitors; standard and delegating monitors have predefined pointcuts. A custom monitor must define a location type and a pointcut. Pointcut syntax is documented in Defining Pointcuts for Custom Monitors .

Table 12-3 (Cont.) <wldf-instrumentation-monitor> XML Elements in the DIAG_MODULE.xml or weblogic-diagnostics.xml 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. 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 12-1.</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 [Configuring the](#)

[DyeInjection Monitor to Manage Diagnostic Contexts](#), for detailed information about diagnostic contexts and dye vectors.

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

Table 12-4 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 12-4 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.

Configuring Server-Scoped Instrumentation

You can configure instrumentation as part of diagnostic descriptor file to implement the system-level instrumentation. You can define the configuration of one or more server-scope diagnostic monitors in the descriptor file.

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. In addition, for each server or cluster in a domain, you can deploy multiple diagnostic descriptor files simultaneously. However, one reason for creating more than one file is for flexibility. For example, you could have five diagnostic descriptor files in the *DOMAIN_HOME/config/diagnostics* directory. Each file contains a different instrumentation (and perhaps Harvester and Policies and Actions) 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 WebLogic Server 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 12-1](#) 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 12-1 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/2.0/
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>
```

Configuring Application-Scoped Instrumentation

Instrumentation is the only component that is deployable to applications. It must be enabled on the server to which the application is deployed. You can enable and disable diagnostic monitors without redeploying an application.

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

 **Note:**

Application classes and libraries that are put on the system classpath are not instrumented. Application class instrumentation works only on classes that are loaded by application classloaders. If application classes are put on the system classpath, either deliberately or inadvertently, they will be loaded by the system classloader. As a result no deployment time weaving is performed on those classes.

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](#)

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`. See the entry for `HttpSessionDebug` in [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 [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 12-5 compares the function and scope of system and application diagnostic modules.

Table 12-5 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

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 WebLogic Server 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 [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 [Creating a Descriptor File for a Delegating Monitor](#), and [Creating a Descriptor File for a Custom Monitor](#), for samples of well-formed descriptor files.

See [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 [Deploying WLDF Application Modules](#).

Keep the following points in mind:

- The diagnostic monitors defined in `weblogic-diagnostics.xml` is listed on the **Deployments: <server_name>: Configuration: Instrumentation** page of the WebLogic Server Administration Console.
- If the `META-INF/weblogic-diagnostics.xml` descriptor in the application archive defines a monitor, it cannot be removed using the WebLogic Server Administration Console. However, it can be disabled or enabled using the WebLogic Server Administration Console.
- You can add additional monitors from the WebLogic Server Administration Console. Any monitors you add from the WebLogic Server 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 WebLogic Server Administration Console.
- Application classes and libraries that are put on the server's classpath are not instrumented. Application class instrumentation works only on classes that are loaded by application classloaders.

If application classes are put on the system classpath, either deliberately or inadvertently, they will be loaded by the system classloader. As a result no deployment time weaving is performed on those classes.

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/2.0/
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.

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 12-2 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/2.0/
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.example.* 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 12-6](#) shows how the pointcut expression from [Example 12-2](#) is parsed. (Note the use of wildcard characters.)

Table 12-6 Description of a Sample Pointcut Expression

Pointcut Expression	Description
<code>call(* com.example.* 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.example.* get* (...)</code>	<code>*</code> : Return value. The wildcard indicates that the methods can have any type of return value.
<code>call(* com.example.* get* (...)</code>	<code>com.example.*</code> : Methods from class <code>com.example</code> and its sub-packages are eligible.
<code>call(* com.example.* get* (...)</code>	<code>get*</code> : Any methods whose name starts with the string <code>get</code> is eligible.
<code>call(* com.example.* 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.example` 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 [Defining Pointcuts for Custom Monitors](#), for a description of the grammar used to define pointcuts.

- [Defining Pointcuts for Custom Monitors](#)
- [Annotation-based Pointcuts](#)

Defining Pointcuts for Custom Monitors

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

A joinpoint is 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.example.* initialize(...))
```

The following pointcut matches the method calls (call sites) on all classes that directly or indirectly implement the `com.example.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.example.MyInterface get*(java.lang.String))
```

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

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

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*(...));
```

Creating Request Performance Data

You can display request performance data in the WebLogic Server Administration Console for configured server-scoped or application-scoped instrumentation. In the console, the Request Performance page displays information about the real-time and historical views of method performance.

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 [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 [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 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 Oracle 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 WebLogic Server Administration Console automatically creates a deployment plan for you and prompts you for the location in which to save it.
- If Oracle HotSwap is not enabled in your deployment plan, or if you do not use a deployment plan, changes to some instrumentation settings require redeployment.

See [Deploying WLDF Application Modules](#).

See [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

The Instrumentation component of the WebLogic Diagnostics Framework (WLDF) also 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 of every request that enters the system, such as the originating user or client address, and attach a *diagnostic context* to that request. This feature allows you to take measurements of specific requests, such as elapsed time, 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 [Accessing Diagnostic Data With the Data Accessor](#)).

For an example of how to use WLST to create a DyeInjection monitor dynamically, see [Example: Dynamically Creating DyeInjection Monitors](#).

This chapter includes the following sections:

- [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.
- [Overview of the Process](#)
The DyeInjection monitor examines the request to see if any of the configured dye values in the dye vector match attributes of the request. You can configure the DyeInjection monitor to identify the requests and track their flow. The tracking of the requests helps to see how the requests are processed as they flow through the system.
- [Configuring the Dye Vector by Using the DyeInjection Monitor](#)
You configure the Dye Vector by using the DyeInjection monitor to monitor the requests in a system. Every request is checked against the configuration of the DyeInjection monitor, and a diagnostic context is created and attached to the request.
- [Configuring Delegating Monitors to Use Dye Filtering](#)
You can use the DyeInjection monitor as a mechanism to restrict when a delegating or custom diagnostic monitor in the diagnostic module is triggered. This process is called *dye filtering*.
- [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.

- [Using Throttling to Control the Volume of Instrumentation Events](#)
You can use throttling to control the number of requests that the monitors process in a diagnostic module.
- [Using `weblogic.diagnostics.context`](#)
The `weblogic.diagnostics.context` package provides applications with access to a diagnostic context.

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 13-1](#)).

- [Context Life Cycle and the Context ID](#)
- [Dyes, Dye Flags, and Dye Vectors](#)
- [Where Diagnostic Context Is Configured](#)

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.

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 [THROTTLE Dye Flag](#).

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

Where Diagnostic Context Is Configured

Diagnostic context is configured as part of a diagnostic module. You use the DyelInjection monitor at the server level to configure the diagnostic context. The DyelInjection monitor is a *standard* diagnostic monitor, so you cannot modify its behavior. The joinpoints where the DyelInjection 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 DyelInjection 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 [Configuring Instrumentation](#).

Overview of the Process

The DyelInjection monitor examines the request to see if any of the configured dye values in the dye vector match attributes of the request. You can configure the DyelInjection monitor to identify the requests and track their flow. The tracking of the requests helps to see how the requests are processed as they flow through the system.

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

1. Configure a dye vector via the DyInjection Module. See [Configuring the Dye Vector via the DyInjection 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 DyInjection 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 DyInjection monitor sets the associated dye bits within the diagnostic context. For example, if the DyInjection 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 [Configuring the Dye Vector via the DyInjection 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 [Configuring Delegating Monitors to Use Dye Filtering](#), for more information.
7. 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 [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.

Configuring the Dye Vector by Using the DyInjection Monitor

You configure the Dye Vector by using the DyInjection monitor to monitor the requests in a system. Every request is checked against the configuration of the DyInjection monitor, and a diagnostic context is created and attached to the request.

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

1. Create a diagnostic module for the server (or servers) you want to monitor in the **Summary of Diagnostic Modules** page. See [Creating a Custom Diagnostic System Module Based on a Built-in](#).
2. Click the name of the newly created module to open the **Settings for <MODULE_NAME>** page.
3. In the **Configuration - Instrumentation** tab, select the **Enabled** check box.
4. In the **Diagnostic Monitors in this Module** tab, add the **DyeInjection** monitor by using the **Add/Remove** button.
5. Click the DyeInjection monitor to open the **Settings for DyeInjection** page.
6. Select the **Enable** check box. (Only one DyeInjection monitor can be used with a diagnostic module at a time.)

You configure the DyeInjection monitor by assigning values to dyes. The available dye flags are described in [Table 13-1](#).

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 WebLogic Server 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*.

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

Example 13-1 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>
```

- [Dyes Supported by the DyeInjection Monitor](#)
- [PROTOCOL Dye Flags](#)
- [THROTTLE Dye Flag](#)
- [When Diagnostic Contexts Are Created](#)

Dyes Supported by the DyeInjection Monitor

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

Table 13-1 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 WebLogic Server Administration Console or in the descriptor files. The connector drivers can assign values to these dyes (using the Connector API), so information about the connections can be carried in the diagnostic context.
COOKIE1 COOKIE2 COOKIE3 COOKIE4	COOKIE1, COOKIE2, COOKIE3 and COOKIE4 are set in the diagnostic context for an HTTP/S request, if the request contains the cookie named weblogic.diagnostics.dye and its value is equal to the value of the respective property (COOKIE1, COOKIE2, COOKIE3, COOKIE4) of the DyeInjection monitor.
DYE_0 DYE_1 DYE_2 DYE_3 DYE_4 DYE_5 DYE_6 DYE_7	DYE_0 to DYE_7 are available only for use by application developers. See Using weblogic.diagnostics.context .
PROTOCOL_HTTP PROTOCOL_IIOB PROTOCOL_JRMP PROTOCOL_RMI PROTOCOL_SOAP PROTOCOL_SSL PROTOCOL_T3	The DyeInjection monitor implicitly identifies the protocol used for a request and sets the appropriate dye(s) in the dye vector, according to the protocol(s) used. PROTOCOL_HTTP is set in the diagnostic context of a request if the request uses HTTP or HTTPS protocol. PROTOCOL_IIOB is set in the diagnostic context of a request if it uses Internet Inter-ORB Protocol (IIOB). PROTOCOL_JRMP is set in the diagnostic context of a request if it uses the Java Remote Method Protocol (JRMP). PROTOCOL_RMI is set in the diagnostic context of a request if it uses the Java Remote Method Invocation (RMI) protocol. PROTOCOL_SSL is set in the diagnostic context of a request if it uses the Secure Sockets Layer (SSL) protocol. PROTOCOL_T3 is set in the diagnostic context of a request if the request uses T3 or T3s protocol

Table 13-1 (Cont.) Request Protocols for Supported Diagnostic Context Dyes

Dye Flags	Description
THROTTLE	The THROTTLE dye is set in the diagnostic context of a request if it satisfies requirements specified by THROTTLE_INTERVAL and/or THROTTLE_RATE properties of the DyeInjection monitor.
USER1 USER2 USER3 USER4	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.

PROTOCOL Dye Flags

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

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 [Using Throttling to Control the Volume of Instrumentation Events](#), for more information.

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.

Configuring Delegating Monitors to Use Dye Filtering

You can use the DyelInjection 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*.



Note:

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

Each monitor can have a *dye mask*, which specifies a selection of the dyes from the DyelInjection 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 13-1 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 DyelInjection 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 13-1 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 WebLogic Server 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 WebLogic Server 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 13-2](#), but this is not recommended. It is better to change the configuration via the WebLogic Server Administration Console on a running server. Any changes you make to `DIAG_MODULE.xml` will not take effect until you redeploy the application.

Example 13-2 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>
  </instrumentation>
</wldf-resource>
```

```

    </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.

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 WebLogic Server 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.)
- [Dye Filtering Example](#)

Dye Filtering Example

[Figure 13-2](#) illustrates how dye filtering works, using a diagnostic module with three diagnostic monitors:

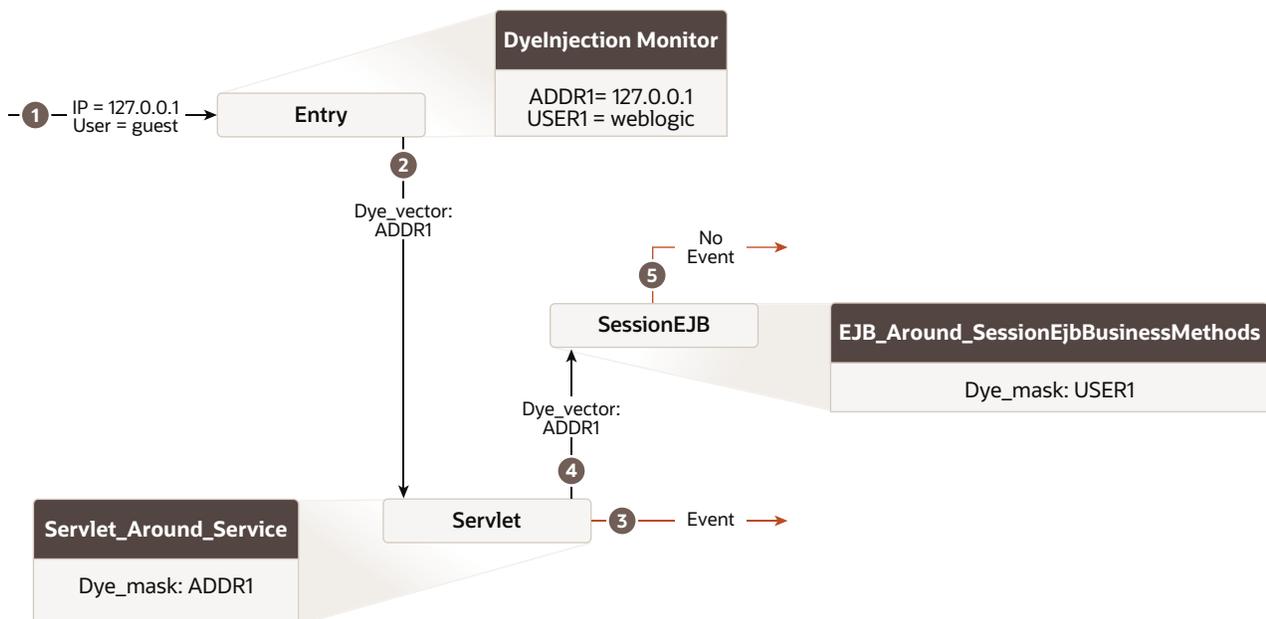
- The `DyeInjection` monitor is configured as follows:


```

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 13-2 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.

Using Throttling to Control the Volume of Instrumentation Events

You can use throttling to control the number of requests that the monitors process 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.

- [Configuring the THROTTLE Dye](#)
- [How Throttling is Handled by Delegating and Custom Monitors](#)

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.

[Example 13-3](#) shows the resulting configuration in the descriptor file for the diagnostics module.

Example 13-3 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>
</wldf-resource>
```

```

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

```

Example 13-4 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 13-4 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>

```

How Throttling is Handled by Delegating and Custom Monitors

Dye masks and dye filtering provide a mechanism for restricting which requests are passed to delegating and custom monitors for handling, based on properties of the requests. The presence of a property in a request is indicated by the presence of a dye, as discussed in [Configuring the Dye Vector via the DyeInjection Monitor](#). 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.

Using `weblogic.diagnostics.context`

The `weblogic.diagnostics.context` package provides applications with 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 13-5](#) 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 13-5 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

The Data Accessor component of the WebLogic Diagnostics Framework (WLDF) accesses 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
- Perform time-based filtering and, when accessing events, filtering by severity, source, and content
- Access diagnostic data in tabular form

You can also use the Data Accessor online (when a server is running) and offline (when a server is not running).

- [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, called diagnostic data stores, which are separated by the types of diagnostic data. For example, server logs, HTTP logs, and harvested metrics are captured in separate data stores.
- [Accessing Diagnostic Data Online](#)
Data Accessor provides access to data stores for individual servers. You can access diagnostic data from a running server.
- [Accessing Diagnostic Data Offline](#)
- [Accessing Diagnostic Data Programmatically](#)
- [Resetting the System Clock Can Affect How Data Is Archived and Retrieved](#)

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, called diagnostic data stores, which 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.

Diagnostic 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:
 - `HarvestedDataArchive`

- EventsDataArchive
 - ServerLog
 - DomainLog
 - HTTPAccessLog
 - DataSourceLog
 - WebAppLog
 - ConnectorLog
 - JMSMessageLog
 - JMSSAFMessageLog
 - CUSTOM
- 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 *Understanding WebLogic Logging Services* in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*.

Accessing Diagnostic Data Online

Data Accessor provides access to data stores for individual servers. You can access diagnostic data from a running server.

You can access the data using one of the following ways:

- WebLogic Server Administration Console
- JMX APIs
- WebLogic Scripting Tool (WLST)
- WLDF query language
- [Accessing Data Using the Administration Console](#)
- [Accessing Data Programmatically Using Runtime MBeans](#)
- [Using WLST to Access Diagnostic Data Online](#)
- [Using the WLDF Query Language with the Data Accessor](#)

Accessing Data Using the Administration Console

You do not use the Data Accessor explicitly in the WebLogic Server 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*.

Accessing Data Programmatically Using Runtime MBeans

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

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

See [WLDFAccessRuntimeMBean](#) in the *MBean Reference for Oracle WebLogic Server*.

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

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

The client can retrieve the `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");
rmbs.getAttribute(objName, "WLDFDataAccessRuntimes");
```

See [WLDFDataAccessRuntimeMBean](#) in the *MBean Reference for Oracle WebLogic Server*.

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 *WLST Command Reference for WebLogic Server*.

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 [WLDF Query Language](#).

Accessing Diagnostic Data Offline

You can use the WLST `exportDiagnosticData()` command to access historical diagnostic data from an offline server. For the syntax and examples of this command, see *Diagnostics Commands* in the *WLST Command Reference for WebLogic Server*.

 **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.

Accessing Diagnostic Data Programmatically

You can use the JMX API to access diagnostic data stored by WLDF. [Example 14-1](#) shows the source Java code for a utility that uses the Accessor to query the different archive data stores.

Example 14-1 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 WLAcessor {

    /** Creates a new instance of WLAcessor */
    public WLAcessor(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", "password");
    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 WLAccessor [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: 'password'");
        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 WLAccessor user=system pass=gumby1234 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;
}

```

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

The WebLogic Diagnostics Framework (WLDF) supports the ability to configure and manage instrumentation for an application by configuring and deploying a diagnostics application module as resource that is scoped to that application. The configuration of the diagnostics application module is persisted in a descriptor file that you deploy with the application. A diagnostic application module deployed in this way is available only to the application in which it is enclosed. Using application-scoped diagnostic application modules ensures that an application always has access to the required resources and simplifies the process of deploying the application in new environments.

 **Note:**

Note the following:

- Only the Instrumentation component can be used with applications (see [Configuring Application-Scoped Instrumentation](#)).
- 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 deploy an application using a deployment plan, which permits dynamic configuration updates.

The following sections explain how to deploy diagnostic application modules:

- [Deploying a Diagnostic Module as an Application-Scoped Resource](#)
- [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 the configuration of an application after it is built, without having to modify the application archives.
- [Using a Deployment Plan: Overview](#)
You can use a deployment plan to dynamically control the configuration options of an application-scoped diagnostic module.
- [Creating a Deployment Plan Using `weblogic.PlanGenerator`](#)
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.
- [Sample Deployment Plan for Diagnostics](#)
You can create a simple deployment plan for diagnostics using PlanGenerator.
- [Enabling Java HotSwap](#)
You can enable Java HotSwap to update the configuration of the application with the modified deployment plan values.

- [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.
- [Updating an Application with a Modified Plan](#)
You can change configuration settings by modifying the deployment plan and then updating or redeploying the application, depending on whether HotSwap is enabled.

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:

```
C:\Oracle\Middleware\Oracle_Home\user_projects\applications\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 15-1](#). The details of how to work with diagnostic application modules is described throughout this section. See [Configuring Instrumentation](#), for information about working with diagnostic system modules.

Table 15-1 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

Table 15-1 (Cont.) 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
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 [Using Deployment Plans to Dynamically Control Instrumentation Configuration](#), for information about HotSwap.

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 the configuration of an application after it 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 [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 15-2](#).

Table 15-2 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, HotSwap 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>

Using a Deployment Plan: Overview

You can use a deployment plan to dynamically control the configuration options of an application-scoped diagnostic module.

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 [Configuring Application-Scoped Instrumentation](#).

The schema for `weblogic-diagnostics.xml` is available at <http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/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 [Creating a Deployment Plan Using weblogic.PlanGenerator](#).
4. Start the server, optionally enabling Java HotSwap. See [Enabling Java HotSwap](#).
5. Deploy the application using the deployment plan. See [Deploying an Application with a Deployment Plan](#).
6. When needed, edit the plan and update the application with the plan. See [Updating an Application with a Modified Plan](#).

Creating a Deployment Plan Using `weblogic.PlanGenerator`

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.

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

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*

Sample Deployment Plan for Diagnostics

You can create a simple deployment plan for diagnostics using `PlanGenerator`.

[Example 15-1](#) 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 15-1 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 [WLDF Instrumentation Library](#).

Enabling Java HotSwap

You can enable Java HotSwap to update the configuration of the application with the modified deployment plan values.

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

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

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 WebLogic Server 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.

Updating an Application with a Modified Plan

You can change configuration settings by modifying the deployment plan and then updating or redeploying the application, depending on whether HotSwap is enabled.

See [Enabling Java HotSwap](#) 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 WebLogic Server 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

Monitoring Dashboard 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.

- [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.
- [Scope of the Diagnostic Information Displayed](#)
- [About the Monitoring Dashboard Interface](#)
The Monitoring Dashboard includes two main panels. Use the explorer panel to select the view you want to display, and use the view display panel to see charts of the metric values that have been captured in that view.
- [Understanding How Metrics Are Collected and Presented](#)
The Monitoring Dashboard displays metrics from two sources: realtime polled metrics, and metrics previously harvested and stored in the Archive.
- [The Parts of a Chart](#)
A chart contains several key parts, including the chart name, viewport, X- and Y- axes for plotting data, a legend for each displayed metric, view controls, and more.

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 WebLogic Server Administration Console to use the Monitoring Dashboard; but if you are not logged in, you are prompted for your username and password credentials.

See [Launch the Monitoring Dashboard](#) in *Oracle WebLogic Server Administration Console Online Help*.

Scope of the Diagnostic Information Displayed

The diagnostic data displayed by the Monitoring Dashboard consists of runtime 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 runtime MBean instances from one or more servers in the domain.

The Monitoring Dashboard obtains metrics from two sources:

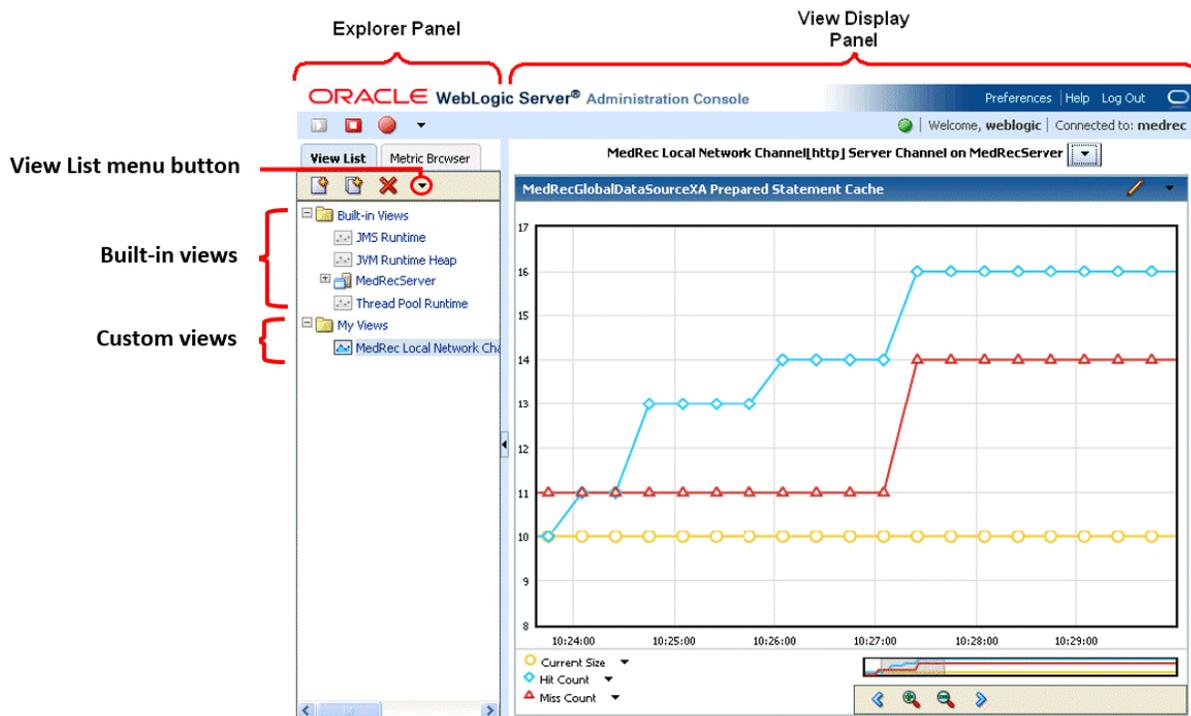
- Directly from active runtime MBean instances — these metrics are sometimes called *polled metrics* in this chapter.

- 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 runtime MBean instances and returned to the Monitoring Dashboard.

About the Monitoring Dashboard Interface

The Monitoring Dashboard includes two main panels. Use the explorer panel to select the view you want to display, and use the view display panel to see charts of the metric values that have been captured in that view.

Figure 16-1 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. See [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. See [Metric Browser](#).
- [View List](#)
- [Metric Browser](#)
- [View Display Panel](#)

View List

To display a view, select it from the View List, shown in [Figure 16-1](#).

Views are presented in two primary categories:

- **Built-in views**

The built-in views are a set of predefined views of available runtime metrics for all running WebLogic Server instances in the domain. These views surface some of the more critical runtime 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 WebLogic Server 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.

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](#)

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 [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 [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 16-2](#), 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 runtime 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 16-2 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.

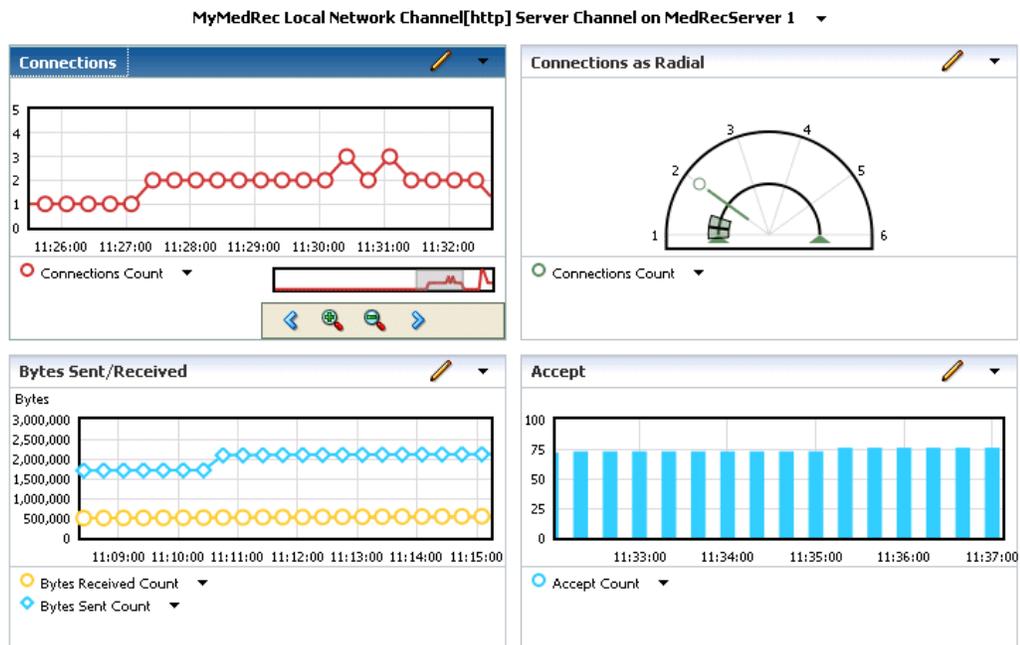
For information about using the Metric Browser, see the following topics in *Oracle WebLogic Server Administration Console Online Help*:

- [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](#)

View Display Panel

A view is a collection of one or more charts that display captured metric values, as shown in [Figure 16-3](#). Only one view is displayed at a time in the Monitoring Dashboard; however, multiple views can be running simultaneously.

Figure 16-3 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 [The Parts of a Chart](#).

Understanding How Metrics Are Collected and Presented

The Monitoring Dashboard displays metrics from two sources: realtime polled metrics, and metrics previously harvested and stored in the Archive.

Realtime, polled metric values are obtained at regular intervals from running WebLogic Server instances and returned to the Monitoring Dashboard. To view these 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 runtime 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 runtime 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 WebLogic Server Administration Console as well as the Monitoring Dashboard. For information about configuring the Harvester to collect metrics, see [Configuring the Harvester for Metric Collection](#).

- [About Metrics and Chart Types](#)
- [Sequence in which Metrics Data is Displayed](#)
- [Notes about Metric Data Retention](#)

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.

- [Current Time Range Charts](#)

- [Custom Time Range Charts](#)

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.

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.

Sequence in which Metrics Data is Displayed

If the Harvester is configured to collect runtime 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 runtime MBean instances corresponding to the metrics contained in the chart.
 - If the Harvester has collected data for this metric in the Archive, that data is 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.

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. See [Retiring Data from the Archives](#).

The Parts of a Chart

A chart contains several key parts, including the chart name, viewport, X- and Y- axes for plotting data, a legend for each displayed metric, view controls, and more.

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*.

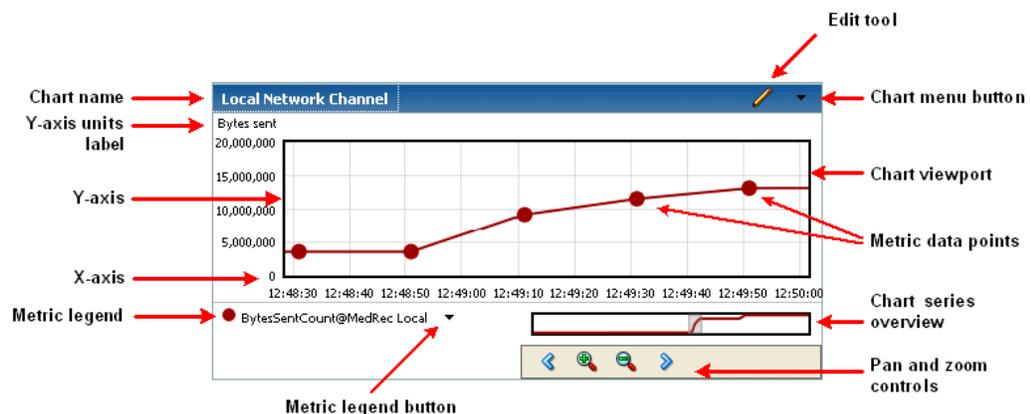
- Buttons for panning the and zooming the data displayed on the chart's X-axis. These buttons are part of the chart series overview, so the display properties set for the chart series overview also apply to these buttons.
- Optional Y-axis units label
- Chart menu, available by selecting the chart menu 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 16-4 shows each of these parts as they appear in a line plot chart.

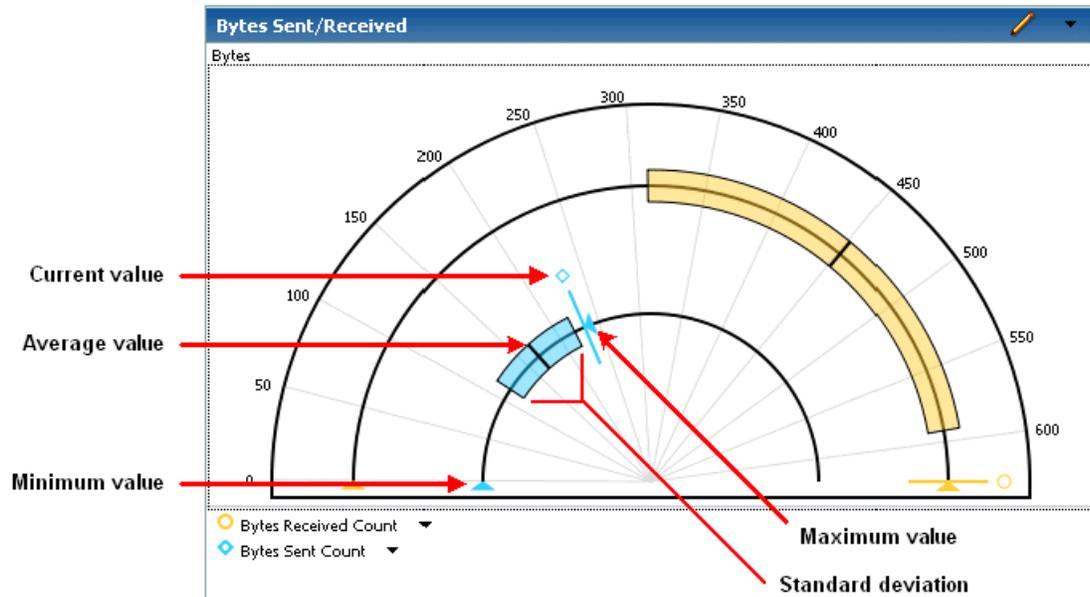
Figure 16-4 Parts of a Chart



A gauge chart, shown in [Figure 16-5](#), contains the following additional information about each metric that has been added to it:

- Minimum and maximum values
- Average value
- Standard deviation

Figure 16-5 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, indicated in [Figure 16-5](#) 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

As an alternative to using the WebLogic Server Administration Console or Fusion Middleware Control to enable, configure, and monitor the WebLogic Diagnostics Framework (WLDF), you can also use the JMX API or the WebLogic Scripting Tool (WLST) to perform these tasks programmatically.

See the following for additional information about how to develop and deploy JMX applications and to use WLST:

- *Developing Applications for Oracle WebLogic Server*
- *Developing Manageable Applications Using JMX for Oracle WebLogic Server*
- *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*
- *Deploying Applications to Oracle WebLogic Server*
- *Understanding the WebLogic Scripting Tool*
- [How WLDF Generates and Retrieves Data](#)
The process WLDF uses to generate and retrieve diagnostic data largely depends on how its main components are configured.
- [Mapping WLDF Components to Beans and Packages](#)
- [Programming Tools](#)
WLDF supports the use of multiple tools, such as WLST, JMX, and REST, for performing tasks programmatically.
- [WLDF Packages](#)
WLDF provides two packages you can use to perform select operations programmatically.
- [Programming WLDF: Examples](#)

How WLDF Generates and Retrieves Data

The process WLDF uses to generate and retrieve diagnostic data largely depends on how its main components are configured.

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 Policies and Actions 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 executed, 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 WebLogic Server 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 [Understanding WLDF Configuration](#).)

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

Mapping WLDF Components to Beans and Packages

When you create diagnostic system modules using the WebLogic Server Administration Console or WLST, WebLogic Server creates MBeans (managed beans) for each module. You can access these MBeans using JMX or WLST. Because WLST is a JMX client; any task you can perform using WLST you can also perform programmatically through JMX.

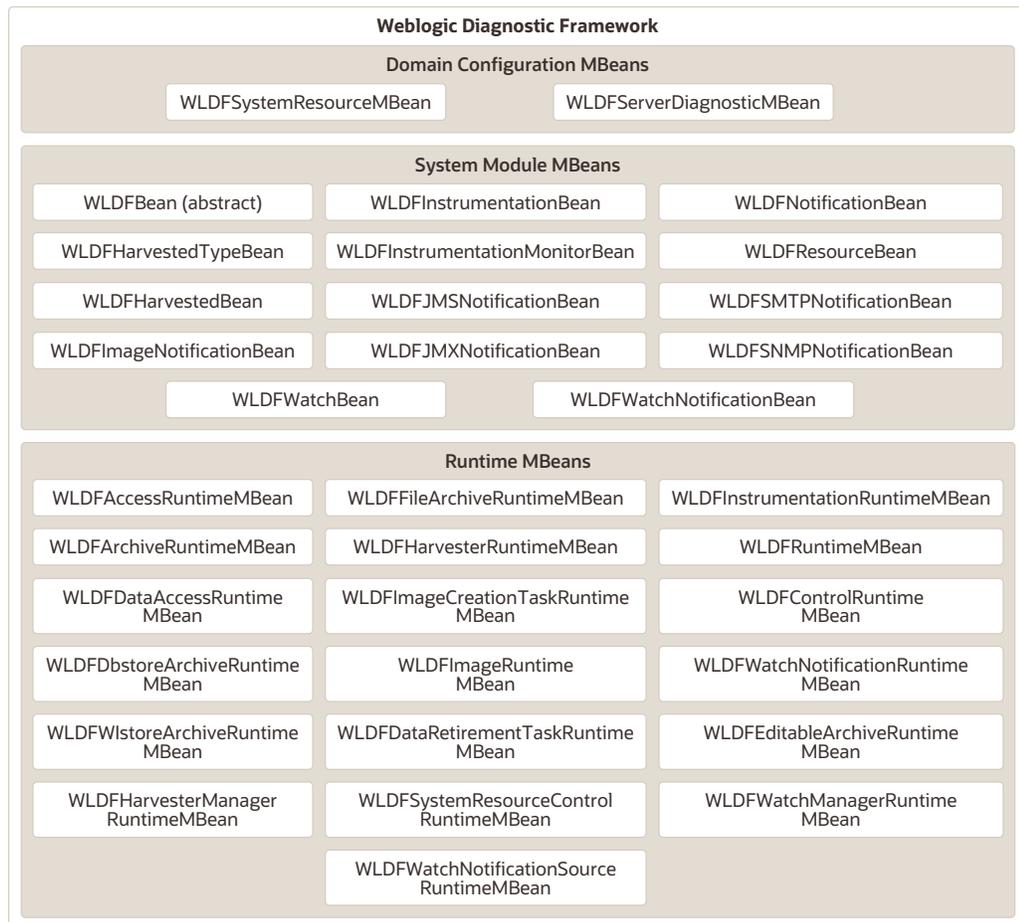
[Table 17-1](#) lists the beans and packages associated with WLDF and its components. [Figure 17-1](#) groups the beans by type.

Table 17-1 Mapping WLDF Components to Beans and Packages

Component	Beans / Packages
WLDF	WLDFServerDiagnosticMBean WLDFSystemResourceMBean WLDFBean (abstract) WLDFResourceBean WLDFRuntimeMBean
Diagnostic Image	WLDFImageNotificationBean WLDFImageCreationTaskRuntimeMBean WLDFImageRuntimeMBean
Instrumentation	WLDFInstrumentationBean WLDFInstrumentationMonitorBean WLDFInstrumentationRuntimeMBean
Diagnostic Context	Package: weblogic.diagnostics.context DiagnosticContextHelper DiagnosticContextConstants
Harvester	WLDFHarvesterBean WLDFHarvestedTypeBean WLDFHarvesterRuntimeMBean

Table 17-1 (Cont.) Mapping WLDF Components to Beans and Packages

Component	Beans / Packages
Policies and Actions	WLDFNotificationBean WLDFWatchNotificationBean WLDFJMSNotificationBean WLDFJMXNotificationBean WLDFSMTPNotificationBean WLDFSNMPNotificationBean WLDFWatchNotificationRuntimeMBean Package: weblogic.diagnostics.watch JMXWatchNotification WatchNotification
Archive	WLDFArchiveRuntimeMBean WLDFDbstoreArchiveRuntimeMBean WLDFFileArchiveRuntimeMBean WLDFWlstoreArchiveRuntimeMBean
Accessor	WLDFAccessRuntimeMBean WLDFDataAccessRuntimeMBean
Runtime Control	WLDFControlRuntimeMBean WLDFSystemResourceControlRuntimeMBean

Figure 17-1 WLDf Configuration MBeans, Runtime MBeans, and System Module Beans

Programming Tools

WLDf supports the use of multiple tools, such as WLST, JMX, and REST, for performing tasks programmatically.

For example, you can use these tools to do the following:

- Create and modify diagnostic descriptor files to configure the WLDf Harvester, Instrumentation, and Policies and Actions 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 policies and actions 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.)
- [Configuration and Runtime APIs](#)

Configuration and Runtime APIs

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

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

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

- [Configuration APIs](#)
- [Runtime APIs](#)

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.
- Policies and Actions: You can use the configuration APIs to enable, disable, create, and destroy policies and actions. You can also use the configuration APIs to:
 - Set the policy type, policy expressions, and severity for policies
 - Set alarm type and alarm reset period for actions

- Configure a policy to execute a diagnostic image capture
- Add and remove actions from policies
- Archive: Set the archive type and the archive directory

Runtime APIs

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

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

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

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

WLDF Packages

WLDF provides two packages you can use to perform select operations programmatically.

- [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 an action for a policy.

Programming WLDF: Examples

WLDF provides a number of beans and packages you can use to access and modify information about a running server. The following examples show how to use these components:

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

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

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 [Configuring the DyeInjection Monitor to Manage Diagnostic Contexts](#).)

To compile and run the program:

1. Copy the `DiagnosticContextExample.java` example ([Example 17-1](#)) 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 17-1 Example: DiagnosticContextExample.java

```
package weblogic.diagnostics.examples;
import weblogic.diagnostics.context.DiagnosticContextHelper;
public class DiagnosticContextExample {
```

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

Example: HarvesterMonitor.java

The HarvesterMonitor program uses the Harvester JMX notification to identify when a harvest cycle has occurred. It then retrieves the new values using the Accessor. All access is performed through JMX. 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 [Configuring the Harvester for Metric Collection](#).

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

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 send 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 17-2](#) describes each notification listener type that is provided with WebLogic Server and the relevant configuration settings for each type.

Table 17-2 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 executed from policies are stored in the server log file in addition to being executed through the configured medium.

HarvesterMonitor.java

To compile and run the HarvesterMonitor program:

1. Copy the HarvesterMonitor.java example ([Example 17-2](#)) to a directory and compile it with:

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

This creates the `./weblogic/diagnostics/examples` directory and populates it with `HarvesterMonitor.class` and `HarvesterMonitor$HarvestCycleHandler.class`.

2. Start the monitor. The command syntax is:

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

You need access to a WebLogic Server instance, and 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 displays 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 policies (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` (and also the password, shown as *password*):

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

See [Example 17-3](#) for an example of output from the HarvesterMonitor.

Example 17-2 Example: HarvesterMonitor.java

```
package weblogic.diagnostics.examples;
import weblogic.management.mbeanservers.runtime.RuntimeServiceMBean;
import javax.management.*;
import javax.management.remote.*;
import javax.naming.Context;
import java.util.*;
public class HarvesterMonitor {

    private static String accessorRuntimeMBeanName;
    private static ObjectName accessorRuntimeMBeanObjectName;
    private static String harvRuntimeMBeanName;
    private static ObjectName harvRuntimeMBeanObjectName;
    private static MBeanServerConnection rmbs;
    private static ObjectName getObjectName(String objectNameStr) {
        try { return new ObjectName(getCanonicalName(objectNameStr)); }
        catch (RuntimeException x) { throw x; }
        catch (Exception x) { x.printStackTrace(); throw new
            RuntimeException(x); }
    }
    private static String getCanonicalName(String objectNameStr) {
        try { return new ObjectName(objectNameStr).getCanonicalName(); }
        catch (RuntimeException x) { throw x; }
        catch (Exception x) { x.printStackTrace(); throw new
            RuntimeException(x); }
    }
    private static String serverName;
    private static int port;
    private static String userName;
    private static String password;
    private static ArrayList typesToMonitor = null;
    public static void main(String[] args) throws Exception {
        if (args.length < 4) {
            System.out.println(
                "Usage: java weblogic.diagnostics.harvester.HarvesterMonitor " +
                "<serverName> <port> <userName> <password> [<types>]" +
                weblogic.utils.PlatformConstants.EOL +
                " where <types> (optional) is a comma-separated list " +
                "of types to monitor.");
            System.exit(1);
        }
        serverName = args[0];
        port = Integer.parseInt(args[1]);
        userName = args[2];
        password = args[3];
        accessorRuntimeMBeanName = getCanonicalName(
            "com.bea:ServerRuntime=" + serverName +
            ",Name=HarvestedDataArchive,Type=WLDFDataAccessRuntime" +
            ",WLDFAccessRuntime=Accessor,WLDFRuntime=WLDFRuntime");
        accessorRuntimeMBeanObjectName =
            getObjectName(accessorRuntimeMBeanName);
        harvRuntimeMBeanName = getCanonicalName(
            "com.bea:ServerRuntime=" + serverName +
            ",Name=WLDFHarvesterRuntime,Type=WLDFHarvesterRuntime" +
            ",WLDFRuntime=WLDFRuntime");
        harvRuntimeMBeanObjectName = getObjectName(harvRuntimeMBeanName);
        if (args.length > 4) {
            String typesStr = args[4];
            typesToMonitor = new ArrayList();

```

```

int index;
while ((index = typesStr.indexOf(",") > 0) {
    String typeName = typesStr.substring(0,index).trim();
    typesToMonitor.add(typeName);
    typesStr = typesStr.substring(index+1);
}
typesToMonitor.add(typesStr.trim());
}
rmbs = getRuntimeMBeanServerConnection();
new HarvesterMonitor().new HarvestCycleHandler();
while(true) {Thread.sleep(100000);}
}
static protected String JNDI = "/jndi/";
static public MBeanServerConnection getRuntimeMBeanServerConnection()
    throws Exception {
    JMXServiceURL serviceURL;
    serviceURL =
        new JMXServiceURL("t3",
            "localhost",
            port,
            JNDI + RuntimeServiceMBean.MBEANSERVER_JNDI_NAME);
    System.out.println("ServerName=" + serverName);
    System.out.println("URL=" + serviceURL);
    Hashtable h = new Hashtable();
    h.put(Context.SECURITY_PRINCIPAL, userName);
    h.put(Context.SECURITY_CREDENTIALS, password);
    h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
        "weblogic.management.remote");
    JMXConnector connector = JMXConnectorFactory.connect(serviceURL,h);
    return connector.getMBeanServerConnection();
}
class HarvestCycleHandler implements NotificationListener {
    // used to track harvest cycles
    private int timestampIndex;
    private int domainIndex;
    private int serverIndex;
    private int typeIndex;
    private int instNameIndex;
    private int attrNameIndex;
    private int attrTypeIndex;
    private int attrValueIndex;
    long lastSampleTime = System.currentTimeMillis();
    HarvestCycleHandler() throws Exception{
        System.out.println("Harvester monitor started...");
        try {
            setUpRecordIndices();
            rmbs.addNotificationListener(harvRuntimeMBeanObjectName,
                this, null, null);
        }
        catch (javax.management.InstanceNotFoundException x) {
            System.out.println("Cannot find JMX data. " +
                "Is the server name correct?");
            System.exit(1);
        }
    }
    private void setUpRecordIndices() throws Exception {
        Map columnIndexMap = (Map)rmbs.getAttribute(
            accessorRuntimeMBeanObjectName, "ColumnIndexMap");
        timestampIndex =
            ((Integer)columnIndexMap.get("TIMESTAMP")).intValue();
        domainIndex =

```

```

        ((Integer)columnIndexMap.get("DOMAIN")).intValue();
serverIndex =
        ((Integer)columnIndexMap.get("SERVER")).intValue();
typeIndex =
        ((Integer)columnIndexMap.get("TYPE")).intValue();
instNameIndex =
        ((Integer)columnIndexMap.get("NAME")).intValue();
attrNameIndex =
        ((Integer)columnIndexMap.get("ATTRNAME")).intValue();
attrTypeIndex =
        ((Integer)columnIndexMap.get("ATTRTYPE")).intValue();
attrValueIndex =
((Integer)columnIndexMap.get("ATTRVALUE")).intValue();
    }
    public synchronized void handleNotification(Notification notification,
                                                Object handback) {
        System.out.println("\n-----");
        long thisSampleTime = System.currentTimeMillis()+1;
        try {
            String lastTypeName = null;
            String lastInstName = null;
            String cursor = (String)rmbs.invoke(accessorRuntimeMBeanObjectName,
                                                "openCursor",
                                                new Object[]{new Long(lastSampleTime),
                                                            new Long(thisSampleTime), null},
                                                new String[]{"java.lang.Long",
                                                            "java.lang.Long", "java.lang.String" });
            while ((Boolean)rmbs.invoke(accessorRuntimeMBeanObjectName,
                                        "hasMoreData",
                                        new Object[]{cursor},
                                        new String[]{"java.lang.String"}).booleanValue()) {
                Object[] os = (Object[])rmbs.invoke(accessorRuntimeMBeanObjectName,
                                                    "fetch",
                                                    new Object[]{cursor},
                                                    new String[]{"java.lang.String"});
                for (int i = 0; i < os.length; i++) {
                    Object[] values = (Object[])os[i];
                    String typeName = (String)values[typeIndex];
                    String instName = (String)values[instNameIndex];
                    String attrName = (String)values[attrNameIndex];
                    if (!typeName.equals(lastTypeName)) {
                        if (typesToMonitor != null &&
                            !typesToMonitor.contains(typeName)) continue;
                        System.out.println("\nType " + typeName);
                        lastTypeName = typeName;
                    }
                    if (!instName.equals(lastInstName)) {
                        System.out.println("\n Instance " + instName);
                        lastInstName = instName;
                    }
                    Object attrValue = values[attrValueIndex];
                    System.out.println("    - " + attrName + "=" + attrValue);
                }
            }
            lastSampleTime = thisSampleTime;
        }
        catch (Exception e) {e.printStackTrace();}
    }
}

```

Example 17-3 contains sample output from the `HarvesterMonitor` program:

Example 17-3 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=WLDFHarvesterRuntime,WLDF
Runtime=WLDFRuntime
  - TotalSamplingTime=202048863
  - CurrentSnapshotElapsedTime=1839619
Type weblogic.management.runtime.ServerRuntimeMBean
Instance com.bea:Name=myserver,Type=ServerRuntime
  - RestartRequired=false
  - ListenPortEnabled=true
  - ActivationTime=1118319317071
  - ServerStartupTime=40671
  - ServerClasspath= [deleted long classpath listing]
  - CurrentMachine=
  - SocketsOpenedTotalCount=1
  - State=RUNNING
  - RestartsTotalCount=0
  - AdminServer=true
  - AdminServerListenPort=7001
  - ClusterMaster=false
  - StateVal=2
  - CurrentDirectory=C:\testdomain\
  - AdminServerHost=10.40.8.123
  - OpenSocketsCurrentCount=1
  - ShuttingDown=false
  - SSLListenPortEnabled=false
  - AdministrationPortEnabled=false
  - AdminServerListenPortSecure=false
  - Registered=true

```

Example: JMXAccessorExample.java

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

To compile and run the program:

1. Copy the `JMXAccessorExample.java` example ([Example 17-4](#)) to a directory and compile it with:

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

This creates the `./weblogic/diagnostics/examples` directory and populates it with `JMXAccessorExample.class`.

2. Start the program. The command syntax is:

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

You need access to a WebLogic Server instance, and have 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.

Construct the query using the syntax described in [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 shown as *password*, 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 17-4 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.OBJECT_NAME);
            ObjectName serverRuntime =
                (ObjectName) mbeanServerConnection.getAttribute(service,
                    "ServerRuntime");
            ObjectName wldfRuntime =
                (ObjectName) mbeanServerConnection.getAttribute(serverRuntime,
                    "WLDFRuntime");
            ObjectName wldfAccessRuntime =
                (ObjectName) mbeanServerConnection.getAttribute(wldfRuntime,
                    "WLDFAccessRuntime");
```

```
ObjectName wldfDataAccessRuntime =
    (ObjectName) mbeanServerConnection.invoke(wldfAccessRuntime,
        "lookupWLDfDataAccessRuntime", new Object[] {logicalName},
        new String[] {"java.lang.String"});
String cursor =
    (String) mbeanServerConnection.invoke(wldfDataAccessRuntime,
        "openCursor", new Object[] {query},
        new String[] {"java.lang.String"});
int fetchedCount = 0;
do {
    Object[] rows =
        (Object[]) mbeanServerConnection.invoke(wldfDataAccessRuntime,
            "fetch", new Object[] {cursor},
            new String[] {"java.lang.String"});
    fetchedCount = rows.length;
    for (int i=0; i<rows.length; i++) {
        StringBuffer sb = new StringBuffer();
        Object[] cols = (Object[]) rows[i];
        for (int j=0; j<cols.length; j++) {
            sb.append("Index " + j + "=" + cols[j].toString() + " ");
        }
        System.out.println("Found row = " + sb.toString());
    }
} while (fetchedCount > 0);
mbeanServerConnection.invoke(wldfDataAccessRuntime,
    "closeCursor", new Object[] {cursor},
    new String[] {"java.lang.String"});
} catch(Throwable th) {
    th.printStackTrace();
    System.exit(1);
}
}

private static MBeanServerConnection lookupMBeanServerConnection ()
    throws Exception {
    // construct JMX service URL
    JMXServiceURL serviceURL;
    serviceURL = new JMXServiceURL("iiop", "localhost", 7001,
        JNDI + "weblogic.management.mbeanservers.runtime");
    // Specify the user, password, and WebLogic provider package
    Hashtable h = new Hashtable();
    h.put(Context.SECURITY_PRINCIPAL, "weblogic");
    h.put(Context.SECURITY_CREDENTIALS, "password");
    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
}
```

18

Using Debug Patches

The WebLogic Diagnostics Framework (WLDF) supports the ability for you to apply debug patches dynamically, allowing you to capture diagnostic information using a patch that you can activate and deactivate without the need of a server restart.

- [Dynamic Application of Debug Patches](#)
Dynamic application of debug patches allows you to avoid the server restarts while applying instrumented debug patches to gather additional information about an error.
- [Specifying the Debug Patch Directory](#)
Debug patch JAR files are picked up from a specific directory called the debug patch directory.
- [Configuring the WLDF Debug Patch Agent](#)
To apply debug patches dynamically, the target WebLogic Server instances must be started on the command line with the WLDF debug patch agent.
- [WLST Commands for Debug Patches](#)
WLDF provides a set of WLST commands you can use to list, activate, and deactivate dynamic debug patches.

Dynamic Application of Debug Patches

Dynamic application of debug patches allows you to avoid the server restarts while applying instrumented debug patches to gather additional information about an error.

Debug patches, packaged as JAR files, are generated through My Oracle Support (<https://support.oracle.com/>) and used to gather additional information about an error when it occurs in a production environment. Typically, the debug patch JAR files are added to the classpath and all server instances must be restarted for the JAR files to take effect. This can present problems, as it might not be possible to restart the server instances in a production environment due to scheduling and other constraints. Additionally, after the server instances are restarted, in-memory states are lost and the problem may disappear or take awhile to reappear. Also, when these debug patches are no longer needed, they can be deactivated without server restarts.

When dynamically applying debug patches, WebLogic Server uses Java HotSwap to replace the loaded classes with the versions provided in the debug patch JAR files. See [Enabling Java HotSwap](#).

Specifying the Debug Patch Directory

Debug patch JAR files are picked up from a specific directory called the debug patch directory.

This directory is specified domain-wide using the `DebugPatchDirectory` attribute of the `DebugPatchesMBean`. By default, the `debug_patches` directory under the `DOMAIN_HOME` directory is used as the debug patch directory.

This feature is available to users with administrative privileges in the domain. Only authorized users are able to post debug patch JAR files in the debug patch directory. This directory must be properly protected with file system permissions.

Configuring the WLDF Debug Patch Agent

To apply debug patches dynamically, the target WebLogic Server instances must be started on the command line with the WLDF debug patch agent.

The WLDF debug patch agent handles the following:

- Replaces the loaded classes with the instrumented classes from the debug patch JAR.
- Makes sure that the replacement classes in the debug patch JAR have the same shape as the original classes. If any of the classes do not meet this requirement, none of the classes in the debug patch JAR are swapped in and an error message is logged.
- Logs informational messages to indicate the start and completion of debug patch activation or deactivation.
- Allows only properly authenticated users with administrative privileges to apply a debug patch.

To specify the WLDF debug patch agent on the command line, update your startup script to include the following:

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



Note:

New startup scripts will automatically include the `debug-agent.jar` on the command line unless the `disableDebugPatches` option is specified on the startup script command line.

WLST Commands for Debug Patches

WLDF provides a set of WLST commands you can use to list, activate, and deactivate dynamic debug patches.

[Table 18-1](#) summarizes the list of WLST commands used with debug patches.

Table 18-1 WLST Commands Used With Debug Patches

Command	Summary
<code>activateDebugPatch</code>	Activates a debug patch on the specified targets.
<code>deactivateAllDebugPatches</code>	Deactivates all debug patches on the specified targets.
<code>deactivateDebugPatches</code>	Deactivates a debug patch on the specified targets.

Table 18-1 (Cont.) WLST Commands Used With Debug Patches

Command	Summary
<code>listDebugPatches</code>	Lists the active and available debug patches on the specified targets.
<code>listDebugPatchTasks</code>	Lists the debug patch (activated or deactivated) tasks from the specified targets.
<code>purgeDebugPatchTasks</code>	Purges the debug patch (activated or deactivated) tasks on the specified targets.
<code>showDebugPatchInfo</code>	Displays details about a debug patch on the specified targets.

- [Dynamically Activating a Debug Patch](#)
- [Dynamically Deactivating Debug Patches](#)

Dynamically Activating a Debug Patch

[Example 18-1](#), [Example 18-2](#), and [Example 18-3](#) demonstrate how to use the `activateDebugPatch` command to activate a debug patch on the desired targets. Note that if a specified debug patch is not available in the debug patch directory on a target, a warning is issued and WebLogic Server will attempt to proceed and activate the debug patch on the remaining targets. If one of the classes in the debug patch fails to replace the original class on a target, the entire debug patch JAR file is rejected on that target and WebLogic Server will attempt to activate the debug patch on the remaining targets. Additionally, several debug patches may be activated over time and each debug patch will overlay the original classes and previously activated debug patches. If a class is contained in multiple activated debug patches, the class in the debug patch that was last activated has precedence. The `activateDebugPatch` command returns an array of tasks, each element corresponding to the activation activity on an affected target server instance.

Example 18-1 Activating a Debug Patch on Two Managed Servers

```
# Connected to admin server: Activate debug-patch-01.jar on managed servers
# MS1 and MS2
tasks=activateDebugPatch(Patch='debug-patch-01.jar', Target='MS1,MS2')
```

Example 18-2 Activating a Debug Patch on a Server Instance and a Cluster

```
# Connected to admin server: Activate debug-patch-01.jar on myserver and all
# members of cluster Cluster-0
tasks=activateDebugPatch(Patch='debug-patch-01.jar', Target='myserver,Cluster-0')
```

Example 18-3 Activating a Debug Patch on an Application Targeted to a Cluster

```
# Connected to admin server: Activate debug-patch-03.jar on application 'medrec'
# targeted to cluster Cluster-1
tasks=activateDebugPatch(Patch='debug-patch-03.jar', Target='Cluster-1',
    Application='medrec')
```

Dynamically Deactivating Debug Patches

[Example 18-4](#), [Example 18-5](#), and [Example 18-6](#) demonstrate how to use the `deactivateDebugPatches` command to deactivate debug patches. To specify more than one debug patch, use a comma-separated list. If a specified debug patch is not active on a target,

a warning is issued and the command continues. If no debug patches are specified, all active patches are deactivated on the specified targets and the original classes are activated. After successful deactivation, all targets are left in the same state they were in prior to running this command. The `deactivateDebugPatches` command returns an array of tasks.

Example 18-4 Deactivating Debug Patches on a Managed Server

```
# Connected to MS1: deactivate debug-patch-01.jar
tasks=deactivateDebugPatches(Patches='debug-patch-01.jar')
```

Example 18-5 Deactivating Debug Patches on All Members of a Cluster

```
# Connected to admin server: de-activate debug-patch-01.jar
# and debug-patch-02.jar on all members of cluster Cluster-0
tasks=deactivateDebugPatches(Patches='debug-patch-01.jar,debug-patch-02.jar',
    Target='Cluster-0')
```

Example 18-6 Deactivating Debug Patches on an Application Targeted to a Cluster

```
# Connected to admin server: de-activate debug-patch-03.jar on application
# 'medrec' targeted to cluster Cluster-1
tasks=deactivateDebugPatches(Patches='debug-patch-03.jar', Target='Cluster-1',
    Application='medrec')
```

A

Smart Rule Reference

Smart rules are prepackaged functions provided by the WebLogic Diagnostics Framework (WLDF) that simplify the creation of policy expressions. When used in scheduled policy expressions, as described in [Configuring Smart Rule Based Policies](#), smart rules can execute elastic actions on dynamic clusters, as well as be used in conjunction with any WLDF action. For example, a smart rule that monitors stuck threads in a cluster can be used to execute an SMTP action that sends an email to the system administrator. The smart rules are organized into Cluster Scope Smart Rules and Server Scope Smart Rules.

- [About the Parameters You Specify for Smart Rules](#)
- [Cluster Scope Smart Rules](#)
- [Server Scope Smart Rules](#)

About the Parameters You Specify for Smart Rules

All smart rules involve the collection of metric values, which is the process of gathering data needed for monitoring system state and performance. Metrics are exposed to WLDF as attributes on qualified MBeans. Smart rules cause WLDF to gather values from selected MBean attributes at a specified sampling rate and retain those values for a specified duration of time. This allows you to track trends in metric changes in a server or cluster over time. When you configure a smart rule, you always specify the following parameters:

- [sampling rate](#)
- [retention window](#)
- [threshold value](#)



Note:

Sampling rates and retention windows are completely independent of policy schedules. A policy schedule determines only when a smart rule is evaluated; the policy schedule does not determine the sampling rate or retention window.

sampling rate

The sampling rate is the frequency with which a metric value is collected. For example, a sampling rate of 30 seconds means that the value of an MBean attribute is collected every 30 seconds.

Each smart rule has a default sampling rate. When you are configuring a smart rule using either the WebLogic Server Administration Console or Fusion Middleware Control, you can accept the default sampling rate that is provided in the configuration assistant. However, when you configure a smart rule using WLST, REST, or JMX, you need to explicitly specify the sampling rate.

The sampling rate is a `String` value that can be specified using the following syntax:

```
amount[unit]
```

In the preceding syntax:

- `amount` represents an integer.
- `unit` represents `seconds`, `minutes`, or `hours`. Each can be abbreviated to the first letter. For example: `seconds` can be abbreviated to `s`.

The default sampling rate time unit is `seconds`.

- You may include a space character between `amount` and `unit`.

For example, any of the following can be used to specify 30 seconds:

- `"30"`
- `"30 seconds"`
- `"30snds"`
- `"30s"`

retention window

The retention window is the period of time during which collected samples are retained in an internal buffer for evaluation. For example, a retention window of 5 minutes causes the samples collected during the previous 5 minutes to be retained. As each new sample is collected, the oldest sample is removed.

Smart rules function by calculating the average value of a particular metric that has been collected over the period of time corresponding to the retention window. Obtaining average values allows you to obtain a more representative view of changes, and trends in those changes, that are occurring in a server, cluster, or operational environment of WebLogic Server.

The retention window you specify is a `String` value that uses the same syntax as the [sampling rate](#):

```
amount[unit]
```

The time unit can be `seconds`, `minutes`, or `hours`, and each can be abbreviated. The default time unit in smart rule retention windows is `minutes`, which can be abbreviated to `m`. For example, any of the following can be used to specify 10 minutes:

- `"10"`
- `"10 minutes"`
- `"10mts"`
- `"10m"`

threshold value

The threshold value is an arbitrary value against which the average value of all metrics collected during a retention window is compared. If the average value meets the smart rule's comparison criteria for the threshold value, the smart rule can be evaluated to `true`, assuming all other conditions set in the smart rule are met.

For example, if you want a smart rule to be evaluated as `true` if the average number of idle threads in a cluster is greater than or equal to a specific number, you can enter that number as the threshold value in the [ClusterHighIdleThreads](#) smart rule, which monitors a cluster for a high idle thread count. In this context, the threshold value you specify for this smart rule is referred to as the **high threshold value** because the cluster is monitored to measure whether the average number of idle threads *is greater than or equal to* that threshold.

By contrast, if you want a smart rule to be evaluated as `true` if the average free heap in a cluster falls below a certain amount, you enter that amount as the threshold value in the [ClusterLowHeapFreePercent](#) smart rule, which monitors a cluster for a low free heap. In this context, this threshold value you specify for this smart rule is referred to as the **low threshold value** because the cluster is monitored to measure whether the average free heap amount *is less than* that threshold.

Note that smart rules vary with regard to how the average collected metric value must compare to the threshold value. Some smart rules require that the average collected value must be greater than or equal to the threshold; some require that the average must be greater than the threshold; some require the average to be less than or equal to the threshold; and so on.

Cluster Scope Smart Rules

A cluster scope smart rule is one that is applied to all active nodes in a cluster, and that must be executed from a policy on the Administration Server. The set of cluster scope smart rules provided by WLDF are listed and summarized in [Table A-2](#). For each smart rule, [Table A-2](#) identifies the following:

- The specific metric, typically an MBean attribute, that is sampled
- The condition that causes the smart rule to be evaluated to `true` if, over the course of the [retention window](#), the number of servers with an average metric value that meets specific comparison criteria against the [threshold value](#) is greater than or equal to a specified percentage of all servers in the cluster.

Table A-1 Summary of Administration Server Scope Smart Rules

Smart Rule	Metric	Condition Required for Evaluation to <code>true</code>
ClusterLowThroughput	Throughput metric of the <code>ThreadPoolRuntimeMBean</code>	The average <code>Throughput</code> value is less than the low threshold value.
ClusterHighProcessCpuLoadAverage	<code>ProcessCpuLoad</code> value of the <code>java.lang:type=OperatingSystemMXBean</code>	The average <code>ProcessCpuLoad</code> value is greater than or equal to the high threshold value.
ClusterHighThroughput	Throughput metric of the <code>ThreadPoolRuntimeMBean</code>	The average <code>Throughput</code> value is greater than or equal to the high threshold value.
ClusterLowPendingUserRequests	PendingUserRequestCount value of the <code>ThreadPoolRuntimeMBean</code>	The average <code>PendingUserRequestCount</code> value is less than the low threshold value.
ClusterHighStuckThreads	StuckThreadCount value of the <code>ThreadPoolRuntimeMBean</code>	The average <code>StuckThreadCount</code> value is greater than or equal to the high threshold value.
ClusterLowQueueLength	QueueLength value of the <code>ThreadPoolRuntimeMBean</code>	The average <code>QueueLength</code> value is less than the low threshold value.
ClusterHighPendingUserRequests	PendingUserRequestCount value of the <code>ThreadPoolRuntimeMBean</code>	The average <code>PendingUserRequestCount</code> value is greater than or equal to the high threshold value.

Table A-1 (Cont.) Summary or Administration Server Scope Smart Rules

Smart Rule	Metric	Condition Required for Evaluation to true
ClusterLowProcessCpuLoadAverage	ProcessCpuLoad value of the java.lang:type=OperatingSystem MBean	The average ProcessCpuLoad value is less than the low threshold value.
ClusterHighIdleThreads	ExecuteThreadIdleCount value of the ThreadPoolRuntimeMBean	The average ExecuteThreadIdleCount value is greater than or equal to the high threshold value.
ClusterLowSystemLoadAverage	SystemLoadAverage value of the java.lang:type=OperatingSystem MBean	The average SystemLoadAverage value is less than the low threshold value.
ClusterHighQueueLength	QueueLength value of the ThreadPoolRuntimeMBean	The average QueueLength value is greater than or equal to the high threshold value.
ClusterLowHeapFreePercent	HeapFreePercent value of the JVMRuntimeMBean	The average HeapFreePercent value is less than the low threshold value.
ClusterHighSystemLoadAverage	SystemLoadAverage value of the java.lang:type=OperatingSystem MBean	The average SystemLoadAverage value is greater than or equal to the high threshold value.
ClusterHighHeapFreePercent	HeapFreePercent value of the JVMRuntimeMBean	The average HeapFreePercent value is greater than or equal to the high threshold value.
ClusterLowSystemCpuLoadAverage	SystemCpuLoad value of the java.lang:type=OperatingSystem MBean	The average SystemCpuLoad value is less than the low threshold value.
ClusterLowIdleThreads	ExecuteThreadIdleCount value of the ThreadPoolRuntimeMBean	The average ExecuteThreadIdleCount value is less than the low threshold value.
ClusterGenericMetricRule	Specified MBean attribute value	Any metric visible through JMX satisfies the specified comparison criteria with the threshold value. (This smart rule is a general form of cluster scope rule.)
ClusterHighSystemCpuLoadAverage	SystemCpuLoad value of the java.lang:type=OperatingSystem MBean	The average SystemCpuLoad value is greater than or equal to the high threshold value.

- [ClusterLowThroughput](#)
- [ClusterHighProcessCpuLoadAverage](#)
- [ClusterHighThroughput](#)
- [ClusterLowPendingUserRequests](#)
- [ClusterHighStuckThreads](#)
- [ClusterLowQueueLength](#)
- [ClusterHighPendingUserRequests](#)
- [ClusterLowProcessCpuLoadAverage](#)
- [ClusterHighIdleThreads](#)
- [ClusterLowSystemLoadAverage](#)
- [ClusterHighQueueLength](#)

- [ClusterLowHeapFreePercent](#)
- [ClusterHighSystemLoadAverage](#)
- [ClusterHighHeapFreePercent](#)
- [ClusterLowSystemCpuLoadAverage](#)
- [ClusterLowIdleThreads](#)
- [ClusterGenericMetricRule](#)
The `ClusterGenericMetricRule` smart rule is typically used to observe trends in JMX metrics that are published through the Server Runtime MBean Server and that are not provided through the other cluster scope smart rules.
- [ClusterHighSystemCpuLoadAverage](#)

ClusterLowThroughput

The `ClusterLowThroughput` smart rule measures whether the average throughput in a cluster is decreasing, as indicated by the average value of the `ThreadPoolRuntimeMBean.Throughput` attribute in each Managed Server. You can use this rule to determine whether cluster capacity can be safely reduced; for example, by executing a scale down action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.Throughput` value that satisfies the low threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.Throughput` attribute
- Low Throughput [threshold value](#)
- Percentage of servers in the cluster with an average Throughput value that must be less than the low Throughput threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterLowThroughput("clusterName", "period", "duration", throughputLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for Throughput values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.

Parameter	Description
<code>duration</code>	Retention window during which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> The default time unit is minutes. The default value is 10m. See retention window for more information about specifying this parameter.
<code>throughputLimit</code>	Value established as the low threshold value of the <code>ThreadPoolRuntimeMBean.Throughput</code> attribute.
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average <code>Throughput</code> value that must be less than the value of the <code>throughputLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>15</code>
<code>throughputLimit</code>	<code>5</code>
<code>percentServersLimit</code>	<code>75</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowThroughput("myCluster","30 seconds","15 minutes",5,75)
```

If configured with a scale down action, this example smart rule does the following:

1. Samples the value of the `Throughput` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `Throughput` value, over the last 15 minutes, is less than 5 on at least 75 per cent of the Managed Servers in the cluster.

ClusterHighProcessCpuLoadAverage

The `ClusterHighProcessCpuLoadAverage` smart rule measures an increase in system load across the cluster, as indicated by the average value of the `ProcessCpuLoad` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be increased; for example, by executing a scale up action.
Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ProcessCpuLoad` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the operating system's `ProcessCpuLoad` value
- High `ProcessCpuLoad` [threshold value](#)
- Percentage of servers in the cluster with an average `ProcessCpuLoad` value that must be greater than or equal to the high `ProcessCpuLoad` threshold value in order for the rule to evaluate to `true`

Note:

The value of the `ProcessCpuLoad` metric is platform-specific and is not available on all platforms. The MXBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html#getProcessCpuLoad-->.

Syntax

```
wls:ClusterHighProcessCpuLoadAverage("clusterName", "period", "duration",
procCpuLoadLimit, percentServersLimit)
```

Parameter	Description
<i>clusterName</i>	Name of target dynamic cluster, expressed as a <code>String</code> .
<i>period</i>	Sampling rate for <code>ProcessCpuLoad</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Retention window during which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>procCpuLoadLimit</i>	Value established as the high threshold value of the <code>ProcessCpuLoad</code> metric.
<i>percentServersLimit</i>	Percentage of servers in the cluster with an average <code>ProcessCpuLoad</code> value that must be greater than or equal to the value of the <i>procCpuLoadLimit</i> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	10
<code>procCpuLoadLimit</code>	0.8
<code>percentServersLimit</code>	60

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighProcessCpuLoadAverage("myCluster","30 seconds","10 minutes",0.8,60)
```

If configured with a scale up action, this smart rule does the following:

1. Samples the value of the `ProcessCpuLoad` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `ProcessCpuLoad` value, over the last 10 minutes, is greater than or equal to 0.8 on at least 60 per cent of the Managed Servers in the cluster.

ClusterHighThroughput

The `ClusterHighThroughput` smart rule measures an increase in system throughput across the cluster, as indicated by the average value of the `ThreadPoolRuntimeMBean.Throughput` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.Throughput` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The **sampling rate** and **retention window** of the `ThreadPoolRuntimeMBean.Throughput` metric
- High Throughput **threshold value**
- Percentage of servers in the cluster whose average `Throughput` value during the sampling period must be greater than or equal to the high `Throughput` threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterHighThroughput("clusterName", "period", "duration", throughputLimit,
percentServersLimit)
```

Parameter	Description
<i>clusterName</i>	Name of target dynamic cluster, expressed as a String.
<i>period</i>	Sampling rate for Throughput values, expressed as a String. For example, 30s specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> The default time unit is seconds. The default value is 30s. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a String. <ul style="list-style-type: none"> The default time unit is seconds. The default value is 10m. See retention window for more information about specifying this parameter.
<i>throughputLimit</i>	Value established as the high threshold value of the Throughput attribute.
<i>percentServersLimit</i>	Percentage of servers in the cluster with an average Throughput value that must be greater than or equal to the value of the <i>throughputLimit</i> parameter in order for the smart rule to be evaluated as true. This parameter is expressed as a float value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>clusterName</i>	myCluster
<i>period</i>	30
<i>duration</i>	10
<i>throughputLimit</i>	100
<i>percentServersLimit</i>	60

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighThroughput("myCluster","30 seconds","10 minutes",100,60)
```

When configured with a scale up action, this smart rule does the following:

1. Samples the value of the `Throughput` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average Throughput value, over the last 10 minutes, is greater than or equal to 100 on at least 60 per cent of the Managed Servers in the cluster.

ClusterLowPendingUserRequests

The `ClusterLowPendingUserRequests` smart rule measures a reduction in pending requests across the cluster as indicated by the average value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute in each Managed Server. You can use this rule to determine whether cluster capacity can be reduced; for example, by executing a scale down action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.PendingUserRequestCount` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.PendingUserRequestCount` metric
- Low `PendingUserRequestCount` [threshold value](#)
- Percentage of servers in the cluster with an average `PendingUserRequestCount` value that must be less than the low `PendingUserRequestCount` [threshold value](#) in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterLowPendingUserRequests("clusterName", "period", "duration",
pendingRequestsLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for <code>PendingUserRequestCount</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>pendingRequestsLimit</code>	Value established as the low threshold value of the <code>PendingUserRequestCount</code> attribute.

Parameter	Description
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average <code>PendingUserRequestCount</code> value that must be less than the value of the <code>pendingRequestsLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	10
<code>pendingRequestsLimit</code>	5
<code>percentServersLimit</code>	75

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowPendingUserRequests("myCluster","30 seconds","10 minutes",5,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the `PendingUserRequestCount` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `PendingUserRequestCount` value, over the last 10 minutes, is less than 5 on at least 75 per cent of the Managed Servers in the cluster.

ClusterHighStuckThreads

The `ClusterHighStuckThreads` smart rule measures whether the number of stuck threads is rising and may soon become deadlocked, as indicated by the average value of the `ThreadPoolRuntimeMBean.StuckThreadCount` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.StuckThreadCount` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.StuckThreadCount` attribute

- High `StuckThreadCount` [threshold value](#)
- Percentage of servers in the cluster with an average `ThreadPoolRuntimeMBean.StuckThreadCount` value that must be greater than or equal to the high `StuckThreadCount` threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterHighStuckThreads("clusterName", "period", "duration",
stuckThreadsLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for <code>StuckThreadCount</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>stuckThreadsLimit</code>	Value established as the high threshold value of the <code>StuckThreadCount</code> attribute.
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average <code>StuckThreadCount</code> value that must be greater than or equal to the value of the <code>stuckThreadsLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>10</code>
<code>stuckThreadsLimit</code>	<code>5</code>
<code>percentServersLimit</code>	<code>60</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighStuckThreads("myCluster","30 seconds","10 minutes",5,60)
```

When configured with a scale up action, this smart rule does the following:

1. Samples the value of the `StuckThreadCount` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `StuckThreadCount` value, over the last 10 minutes, is greater than or equal to 5 on at least 60 per cent of the Managed Servers in the cluster.

ClusterLowQueueLength

The `ClusterLowQueueLength` smart rule measures a decrease in system load across the cluster, as indicated by the average value of the `ThreadPoolRuntimeMBean.QueueLength` attribute in each Managed Server. You can use this rule to determine whether cluster capacity can be safely reduced; for example, by executing a scale down action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.QueueLength` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.QueueLength` metric
- Low `QueueLength` [threshold value](#)
- Percentage of servers in the cluster with an average `QueueLength` value that must be less than the low `QueueLength` threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterLowQueueLength("clusterName", "period", "duration", queueLengthLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for <code>QueueLength</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>queueLengthLimit</code>	Value established as the low threshold value of the <code>QueueLength</code> attribute.

Parameter	Description
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average <code>QueueLength</code> value that must be less than the value of the <code>queueLengthLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	15
<code>queueLengthLimit</code>	5
<code>percentServersLimit</code>	75

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowQueueLength("myCluster","30 seconds","15 minutes",5,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the `QueueLength` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `QueueLength` value, over the last 15 minutes, is less than 5 on at least 75 per cent of the Managed Servers in the cluster.

ClusterHighPendingUserRequests

The `ClusterHighPendingUserRequests` smart rule measures an increase in system load across the cluster, as indicated by the average value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be increased; for example, by executing a scale up action.
Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.PendingUserRequestCount` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.PendingUserRequestCount` metric

- High PendingUserRequestCount **threshold value**
- Percentage of servers in the cluster with an average PendingUserRequestCount value that must be greater than or equal to the high PendingUserRequestCount threshold value in order for the rule to evaluate to true

Syntax

```
wls:ClusterHighPendingUserRequests("clusterName", "period", "duration",
pendingRequestsLimit, percentServersLimit)
```

Parameter	Description
<i>clusterName</i>	Name of target dynamic cluster, expressed as a String.
<i>period</i>	Sampling rate for PendingUserRequestCount values, expressed as a String. For example, 30s specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is 30s. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a String. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is 10m. See retention window for more information about specifying this parameter.
<i>pendingRequestsLimit</i>	Value established as the high threshold value of the PendingUserRequestCount attribute.
<i>percentServersLimit</i>	Percentage of servers in the cluster with an average PendingUserRequestCount value that must be greater than or equal to the value of the <i>pendingRequestsLimit</i> parameter in order for the smart rule to be evaluated as true. This parameter is expressed as a float value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>clusterName</i>	myCluster
<i>period</i>	30
<i>duration</i>	10
<i>pendingRequestsLimit</i>	100
<i>percentServersLimit</i>	60

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighPendingUserRequests("myCluster","30 seconds","10 minutes",100,60)
```

When configured with a scale up action, this smart rule does the following:

1. Samples the value of the PendingUserRequestCount metric from each Managed Server instance in myCluster every 30 seconds over a retention window of 10 minutes.

- At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `PendingUserRequestCount` value, over the last 10 minutes, is greater than or equal to 100 on at least 60 per cent of the Managed Servers in the cluster.

ClusterLowProcessCpuLoadAverage

The `ClusterLowProcessCpuLoadAverage` smart rule measures a reduction of system CPU load across a cluster, as indicated by the average value of the `ProcessCpuLoad` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be decreased; for example, by executing a scale down action.
Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ProcessCpuLoad` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

Note that the value of `ProcessCpuLoad` is platform specific and is not available on all platforms.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `java.lang:type=OperatingSystem ProcessCpuLoad` metric
- Low `ProcessCpuLoad` [threshold value](#)
- Percentage of servers in the cluster with an average `ProcessCpuLoad` value that must be less than the low `ProcessCpuLoad` threshold value in order for the rule to evaluate to `true`

Note:

The value of the `ProcessCpuLoad` metric is platform-specific and is not available on all platforms. The MXBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html#getProcessCpuLoad>.

Syntax

```
wls:ClusterLowProcessCpuLoadAverage("clusterName", "period", "duration",
procCpuLoadLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a String.

Parameter	Description
<i>period</i>	<p>Sampling rate for <code>ProcessCpuLoad</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<i>duration</i>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<i>procCpuLoadLimit</i>	Value established as the low threshold value of the <code>ProcessCpuLoad</code> attribute.
<i>percentServersLimit</i>	<p>Percentage of servers in the cluster with an average <code>ProcessCpuLoad</code> value that must be less than the value of the <i>procCpuLoadLimit</i> parameter in order for the smart rule to be evaluated as <code>true</code>.</p> <p>This parameter is expressed as a <code>float</code> value between 0.0 and 100.0</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>clusterName</i>	<code>myCluster</code>
<i>period</i>	<code>30</code>
<i>duration</i>	<code>15</code>
<i>procCpuLoadLimit</i>	<code>0.2</code>
<i>percentServersLimit</i>	<code>75</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowProcessCpuLoadAverage("myCluster","30 seconds","10 minutes",0.2,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the `ProcessCpuLoad` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `ProcessCpuLoad` value, over the last 15 minutes, is less than 0.2 on at least 75 per cent of the Managed Servers in the cluster.

ClusterHighIdleThreads

The `ClusterHighIdleThreads` smart rule measures an increase in the number of idle threads in a cluster, as indicated by the average value of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute in each Managed Server. You can use this rule to determine whether cluster capacity can be safely reduced; for example, by executing a scale down action.
Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` metric
- High `ExecuteThreadIdleCount` [threshold value](#)
- Percentage of Managed Servers in the cluster with an average `ExecuteThreadIdleCount` value that must be greater than or equal to the high `ExecuteThreadIdleCount` [threshold value](#) in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterHighIdleThreads("clusterName", "period", "duration",
idleThreadsLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for <code>ExecuteThreadIdleCount</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>idleThreadsLimit</code>	Value established as the high threshold value of the <code>ExecuteThreadIdleCount</code> attribute.

Parameter	Description
<i>percentServersLimit</i>	Percentage of servers in the cluster with an average <code>ExecuteThreadIdleCount</code> value that must be greater than or equal to the value of the <i>idleThreadsLimit</i> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>clusterName</i>	myCluster
<i>period</i>	30
<i>duration</i>	10
<i>idleThreadsLimit</i>	20
<i>percentServersLimit</i>	75

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighIdleThreads("myCluster","30 seconds","10 minutes",20,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the `ExecuteThreadIdleCount` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `ExecuteThreadIdleCount` value, over the last 10 minutes, is greater than or equal to 20 on at least 75 per cent of the Managed Servers in the cluster.

ClusterLowSystemLoadAverage

The `ClusterLowSystemLoadAverage` smart rule measures a decrease in system load across a cluster, as indicated by the average value of the `SystemLoadAverage` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be decreased; for example, by executing a scale down action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `SystemLoadAverage` value that satisfies the threshold comparison criteria is equal to or greater than the specified percentage of all servers in the cluster.

Note that the value of `SystemLoadAverage` is system dependent.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `java.lang:type=OperatingSystem SystemLoadAverage` metric
- Low `SystemLoadAverage` [threshold value](#)
- Percentage of Managed Servers in the cluster with an average `SystemLoadAverage` value that must be less than the low `SystemLoadAverage` threshold value in order for the rule to evaluate to `true`

 **Note:**

The value of the `SystemLoadAverage` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <http://docs.oracle.com/javase/8/docs/api/java/lang/management/OperatingSystemMBean.html#getSystemLoadAverage-->.

Syntax

```
wls:ClusterLowSystemLoadAverage("clusterName", "period", "duration", loadLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for <code>SystemLoadAverage</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>loadLimit</code>	Value established as the low threshold value of the <code>SystemLoadAverage</code> attribute.
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average <code>SystemLoadAverage</code> value that must be less than the value of the <code>loadLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	15
<code>loadLimit</code>	0.2
<code>percentServersLimit</code>	75

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowSystemLoadAverage("myCluster","30 seconds","15 minutes",0.2,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the `SystemLoadAverage` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `SystemLoadAverage` value, over the last 15 minutes, is less than 0.2 on at least 75 per cent of the Managed Servers in the cluster.

ClusterHighQueueLength

The `ClusterHighQueueLength` smart rule measures an increase in system load across the cluster, as indicated by the average value of the `ThreadPoolRuntimeMBean.QueueLength` attribute in each Managed Server. You can use this rule to determine whether the cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.QueueLength` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.QueueLength` metric
- High `QueueLength` [threshold value](#)
- Percentage of Managed Servers in the cluster with an average `QueueLength` value that must be greater than or equal to the high `QueueLength` threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterHighQueueLength("clusterName", "period", "duration", queueLengthLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .

Parameter	Description
<code>period</code>	<p>Sampling rate for <code>QueueLength</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>queueLengthLimit</code>	<p>Value established as the high threshold value of the <code>QueueLength</code> attribute.</p>
<code>percentServersLimit</code>	<p>Percentage of servers in the cluster with an average <code>QueueLength</code> value that must be greater than or equal to the value of the <code>queueLengthLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code>.</p> <p>This parameter is expressed as a <code>float</code> value between 0.0 and 100.0</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>10</code>
<code>queueLengthLimit</code>	<code>100</code>
<code>percentServersLimit</code>	<code>60</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighQueueLength("myCluster","30 seconds","10 minutes",100,60)
```

When configured with a scale up action, this smart rule does the following:

1. Samples the value of the `QueueLength` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `QueueLength` value, over the last 10 minutes, is greater than or equal to 100 on at least 60 per cent of the Managed Servers in the cluster.

ClusterLowHeapFreePercent

The `ClusterLowHeapFreePercent` smart rule measures an increase in heap stress across a cluster, as indicated by the average value of the `JVMRuntimeMBean.HeapFreePercent` attribute in each Managed Server. You can use

this rule to determine whether the cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `JVMRuntimeMBean.HeapFreePercent` value that satisfies the threshold comparison criteria is greater than or equal to a specific percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `JVMRuntimeMBean.HeapFreePercent` metric
- Low `HeapFreePercent` [threshold value](#)
- Percentage of Managed Servers in the cluster with an average `HeapFreePercent` value during the sampling period that must be less than the low `HeapFreePercent` threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterLowHeapFreePercent("clusterName", "period", "duration", percentFreeLimit, percentServersLimit)
```

Parameter	Description
<i>clusterName</i>	Name of target dynamic cluster, expressed as a <code>String</code> .
<i>period</i>	Sampling rate for <code>HeapFreePercent</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>percentFreeLimit</i>	Value established as the low threshold value of the <code>HeapFreePercent</code> attribute.
<i>percentServersLimit</i>	Percentage of servers in the cluster with an average <code>HeapFreePercent</code> value that must be less than the value of the <i>percentFreeLimit</i> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	10
<code>percentFreeLimit</code>	20
<code>percentServersLimit</code>	60

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowHeapFreePercent("myCluster","30 seconds","10 minutes",20,60)
```

When configured with a scale up action, this smart rule does the following:

1. Samples the value of the `HeapFreePercent` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `HeapFreePercent` value, over the last 10 minutes, is less than 20 on at least 60 per cent of the Managed Servers in the cluster.

ClusterHighSystemLoadAverage

The `ClusterHighSystemLoadAverage` smart rule measures an increase on system load across a cluster, as indicated by the average value of the `SystemLoadAverage` attribute in each Managed Server. You can use this rule to determine if cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `java.lang:type=OperatingSystem SystemLoadAverage` value that satisfies the threshold comparison criteria is greater than or equal to the specified percentage of all servers in the cluster.

Note that the value of the `SystemLoadAverage` is system dependent.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `java.lang:type=OperatingSystem SystemLoadAverage` metric
- High `SystemLoadAverage` [threshold value](#)
- Percentage of Managed Servers in the cluster with an average `SystemLoadAverage` value that must be greater than or equal to the high `SystemLoadAverage` threshold value in order for the rule to evaluate to `true`

 **Note:**

The value of the `SystemLoadAverage` metric is platform-specific and is not available on all platforms. The MXBean attribute from which this metric originates is described at <http://docs.oracle.com/javase/8/docs/api/java/lang/management/OperatingSystemMXBean.html#getSystemLoadAverage-->.

Syntax

```
wls:ClusterHighSystemLoadAverage("clusterName", "period", "duration", loadLimit,
percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	<p>Sampling rate for <code>SystemLoadAverage</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>loadLimit</code>	Value established as the high threshold value of the <code>SystemLoadAverage</code> attribute.
<code>percentServersLimit</code>	<p>Percentage of servers in the cluster with an average <code>SystemLoadAverage</code> value that must be greater than or equal to the value of the <code>loadLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code>.</p> <p>This parameter is expressed as a <code>float</code> value between 0.0 and 100.0</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>5</code>
<code>loadLimit</code>	<code>0.8</code>
<code>percentServersLimit</code>	<code>60</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighSystemLoadAverage("myCluster", "30 seconds", "5 minutes", 0.8, 60)
```

When configured with a scale up action, this smart rule does the following:

1. Samples the value of the `SystemLoadAverage` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `SystemLoadAverage` value, over the last 5 minutes, is greater than or equal to 0.8 on at least 60 per cent of the Managed Servers in the cluster.

ClusterHighHeapFreePercent

The `ClusterHighHeapFreePercent` smart rule measures a reduction in heap stress across a dynamic cluster, as indicated by the average value of the `JVMRuntimeMBean.HeapFreePercent` attribute in each Managed Server. You can use this rule to determine if cluster capacity can be reduced; for example, by executing a scale down action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average JVM free heap percentage value that satisfies the threshold comparison criteria is greater than or equal to a specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the JVM free heap percentage metric
- High JVM free heap [threshold value](#)
- Percentage of Managed Servers in the cluster with an average JVM free heap value that must be greater than or equal to the high JVM free heap threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterHighHeapFreePercent("clusterName", "period", "duration",
percentFreeLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	Sampling rate for JVM free heap percentage values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.

Parameter	Description
<code>percentFreeLimit</code>	Value established as the high threshold value of the JVM free heap percentage.
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average JVM free heap percentage that must be greater than or equal to the <code>percentFreeLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	5
<code>percentFreeLimit</code>	60
<code>percentServersLimit</code>	75

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighHeapFreePercent("myCluster","30 seconds","5 minutes",60,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the JVM free heap percentage metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average JVM free heap percentage value, over the last 5 minutes, is greater than or equal to 60 on at least 75 per cent of the Managed Servers in the cluster.

ClusterLowSystemCpuLoadAverage

The `ClusterLowSystemCpuLoadAverage` smart rule measures a reduction of the system CPU load average across a cluster, as indicated by the average value of the `SystemCpuLoad` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be decreased; for example, by executing a scale down action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `java.lang:type=OperatingSystem SystemCpuLoad` value satisfies the threshold comparison criteria is greater than or equal to a specified percentage of all servers in the cluster.

Note that the value of the `SystemCpuLoad` metric is platform-specific and is not available on all platforms.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `java.lang:type=OperatingSystem SystemCpuLoad` metric
- Low `SystemCpuLoad` [threshold value](#)
- Percentage of Managed Servers in the cluster with an average `SystemCpuLoad` value that must be below the low `SystemCpuLoad` threshold value in order for the rule to evaluate to `true`



Note:

The value of the `SystemCpuLoad` metric is platform-specific and is not available on all platforms. The MXBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html#getSystemCpuLoad-->.

Syntax

```
wls:ClusterLowSystemCpuLoadAverage("clusterName", "period", "duration",
systemCpuLoadLimit, percentServersLimit)
```

Parameter	Description
<i>clusterName</i>	Name of target dynamic cluster, expressed as a <code>String</code> .
<i>period</i>	Sampling rate for <code>SystemCpuLoad</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>systemCpuLoadLimit</i>	Value established as the low threshold value of the <code>SystemCpuLoad</code> attribute.
<i>percentServersLimit</i>	Percentage of servers in the cluster with an average <code>SystemCpuLoad</code> value that must be less than the value of the <i>systemCpuLoadLimit</i> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	30
<code>duration</code>	15
<code>systemCpuLoadLimit</code>	0.2
<code>percentServersLimit</code>	75

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowSystemCpuLoadAverage("myCluster","30 seconds","15 minutes",0.2,75)
```

When configured with a scale down action, this smart rule does the following:

1. Samples the value of the `SystemCpuLoad` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires a scale down action if the following condition evaluates to `true`:

The average `SystemCpuLoad` value, over the last 15 minutes, is less than 0.2 on at least 75 per cent of the Managed Servers in the cluster.

ClusterLowIdleThreads

The `ClusterLowIdleThreads` smart rule measures an increase in load stress across the cluster, as indicated by the average value of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute in each Managed Server. You can use this rule to determine whether cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` value that satisfies the threshold comparison criteria is greater than or equal to a specified percentage of all servers in the cluster.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` metric
- Low `ExecuteThreadIdleCount` [threshold value](#)
- Percentage of Managed Servers in the cluster whose average `ExecuteThreadIdleCount` value is less than the low `ExecuteThreadIdleCount` threshold value in order for the rule to evaluate to `true`

Syntax

```
wls:ClusterLowIdleThreads("clusterName", "period", "duration", idleThreadsLimit",  
percentServerLimit")
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .
<code>period</code>	<p>Sampling rate for <code>ExecuteThreadIdleCount</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is minutes. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>idleThreadsLimit</code>	Value established as the low <code>ExecuteThreadIdleCount</code> threshold value.
<code>percentServersLimit</code>	<p>Percentage of servers in the cluster with an average <code>ExecuteThreadIdleCount</code> value that must be less than the value of the <code>idleThreadsLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code>.</p> <p>This parameter is expressed as a <code>float</code>.</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>10</code>
<code>idleThreadsLimit</code>	<code>5</code>
<code>percentServersLimit</code>	<code>60</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterLowIdleThreads("myCluster","30 seconds","10 minutes",5,60)
```

When configured with a scale up action, this example smart rule:

1. Samples the value of the `ExecuteThreadIdleCount` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `ExecuteThreadIdleCount` value, over the last 10 minutes, is less than 5 on at least 60 per cent of the Managed Servers in the cluster.

ClusterGenericMetricRule

The `ClusterGenericMetricRule` smart rule is typically used to observe trends in JMX metrics that are published through the Server Runtime MBean Server and that are not provided through the other cluster scope smart rules.

Target: Administration Server

Description

This smart rule allows you to view the average value of any metric obtained through JMX within a specific time interval, and compare that average value to a specified threshold value by using a specified comparison operator for each Managed Server in the cluster. If the percentage of servers matching the comparison criteria meets or exceeds the specified limit, the overall condition of the rule is satisfied and this rule returns `true`.

To use this smart rule, specify:

- Dynamic cluster name
- A valid JMX `ObjectName` or `ObjectName` pattern
- An attribute name, or attribute expression (as an EL expression), where the expression is an attribute expression relative to each MBean.

For example, if the MBean is the `ServerRuntimeMBean`, `'OpenSocketsCurrentCount'` obtains the value of the `ServerRuntimeMBean.OpenSocketsCurrentCount` attribute. In contrast, `'HealthState.State'` accesses the `State` value of the `HealthState` child object.

- A valid boolean comparison operator
- A **threshold value** against which the selected attribute is compared
- Percentage of Managed Servers in the cluster whose average attribute value during the sampling period must meet the threshold value in order for the rule to evaluate to `true`
- The **sampling rate** and **retention window** for the metric on each Managed Server instance in the cluster
- Period of time during which samples are collected

Syntax

```
wls:ClusterGenericMetricRule("clusterName", "instancePattern", "attribute",  
"operation", thresholdValue, percentServersLimit, "period", "duration")
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a String.
<code>instancePattern</code>	A valid JMX <code>ObjectName</code> or <code>ObjectName</code> pattern

Parameter	Description
<i>attribute</i>	<p>A Java EL expression that retrieves a value on each MBean instance that matches <i>instancePattern</i>, where the expression is an attribute expression relative to each MBean.</p> <p>For example, if the MBean is the <code>ServerRuntimeMBean</code>, the expression <code>'OpenSocketsCurrentCount'</code> obtains the value of the <code>OpenSocketsCurrentCount</code> attribute of the <code>ServerRuntimeMBean</code>. By contrast, the expression <code>'HealthState.State'</code> obtains the <code>State</code> value of the <code>HealthState</code> child object of that MBean.</p>
<i>operation</i>	A boolean comparison operator: <code><</code> , <code><=</code> , <code>==</code> , <code>>=</code> , or <code>></code> .
<i>thresholdValue</i>	Threshold value against which the value of the <i>attribute</i> parameter is compared.
<i>percentServersLimit</i>	<p>Percentage of servers in the cluster with an average attribute value that must satisfy the comparison criteria with the value of the <i>thresholdValue</i> parameter in order for the smart rule to be evaluated as true.</p> <p>This parameter is expressed as a <code>float</code> value between 0.0 and 100.0</p>
<i>period</i>	<p>Sampling rate for metric values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<i>duration</i>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>clusterName</i>	<code>myCluster</code>
<i>instancePattern</i>	<code>java.lang:type=OperatingSystem</code>
<i>attribute</i>	<code>ProcessCpuLoad</code>
<i>operation</i>	<code>>=</code>
<i>thresholdValue</i>	<code>0.9</code>
<i>percentServersLimit</i>	<code>75</code>
<i>period</i>	<code>30</code>
<i>duration</i>	<code>10</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterGenericMetricRule("myCluster", "java.lang:type=OperatingSystem", "ProcessCpuLoad", ">=", 0.9, 75, "30 seconds", "10 minutes")
```

This example smart rule:

1. Samples the value of the `ProcessCpuLoad` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, this smart rule evaluates to `true` in the following condition:

The average value of `ProcessCpuLoad` on the `OperatingSystemMXBean`, over the last 10 minutes, is greater than or equal to 0.9 on at least 75 per cent of the Managed Servers in the cluster.

ClusterHighSystemCpuLoadAverage

The `ClusterHighSystemCpuLoadAverage` smart rule measures an increase on system load across the cluster, as indicated by the average value of the operating system `SystemCpuLoad` attribute in each Managed Server. You use this rule to determine whether cluster capacity needs to be increased; for example, by executing a scale up action.

Target: Administration Server

Description

This smart rule evaluates to `true` if the number of Managed Servers with an average `java.lang:type=OperatingSystem SystemCpuLoad` value that satisfies the threshold comparison criteria is greater than or equal to a specified percentage of all servers in the cluster.

Note that the value of `SystemCpuLoad` is platform-specific and is not available on all platforms.

To use this smart rule, specify:

- The **sampling rate** and **retention window** for the `java.lang:type=OperatingSystem SystemCpuLoad` metric
- High `SystemCpuLoad` **threshold value**
- Percentage of Managed Servers in the cluster with an average `SystemCpuLoad` value that is greater than or equal to the high `SystemCpuLoad` threshold value in order for the rule to evaluate to `true`



Note:

The value of the `SystemCpuLoad` metric is platform-specific and is not available on all platforms. The MXBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html#getSystemCpuLoad-->.

Syntax

```
wls:ClusterHighSystemCpuLoadAverage("clusterName", "period", "duration",
systemCpuLoadLimit, percentServersLimit)
```

Parameter	Description
<code>clusterName</code>	Name of target dynamic cluster, expressed as a <code>String</code> .

Parameter	Description
<code>period</code>	Sampling rate for <code>SystemCpuLoad</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>systemCpuLoadLimit</code>	Value established as the high threshold value of the <code>SystemCpuLoad</code> attribute.
<code>percentServersLimit</code>	Percentage of servers in the cluster with an average <code>SystemCpuLoad</code> value that must be greater than or equal to the value of the <code>systemCpuLoadLimit</code> parameter in order for the smart rule to be evaluated as <code>true</code> . This parameter is expressed as a <code>float</code> value between 0.0 and 100.0

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>clusterName</code>	<code>myCluster</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>5</code>
<code>systemCpuLoadLimit</code>	<code>0.8</code>
<code>percentServersLimit</code>	<code>60</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ClusterHighSystemCpuLoadAverage("myCluster","30 seconds","5 minutes",0.8,60)
```

When configured with a scale up action, this example smart rule:

1. Samples the value of the `SystemCpuLoad` metric from each Managed Server instance in `myCluster` every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires a scale up action if the following condition evaluates to `true`:

The average `SystemCpuLoad` value, over the last 5 minutes, is greater than or equal to 0.8 on at least 60 per cent of the Managed Servers in the cluster.

Server Scope Smart Rules

A server scope smart rule is one that is applied only to the local WebLogic Server instance on which the policies associated with that smart rule are run. You can execute

policies containing server scope smart rules on the Administration Server or any individual Managed Server in the domain. The set of server scope smart rules packaged with the WebLogic Diagnostics Framework (WLDF) are listed and summarized in [Table A-2](#).

Table A-2 Summary of Managed Server Scope Smart Rules

After the retention window, the following smart rule returns true if . . .
ServerLowIdleThreads	The average ThreadPoolRuntimeMBean.ExecuteThreadIdleCount value on the local server is equal to or less than the low threshold value.
ServerHighThroughput	The average ThreadPoolRuntimeMBean.Throughput value on the local server is greater than or equal to the high threshold value.
ServerGenericMetricRule	The average value of a metric visible through JMX within the local JVM satisfies the comparison criteria with the threshold value.
ServerLowPendingUserRequests	The average ThreadPoolRuntimeMBean.PendingUserRequestCount value on the local server is less than the low threshold value.
ServerLowProcessCpuLoadAverage	The average value of the <code>ProcessCpuLoad</code> metric of the <code>java.lang:type=OperatingSystem</code> MBean on the local server is less than the low threshold value.
ServerHighSystemLoadAverage	The average value of the <code>SystemLoadAverage</code> metric from the <code>java.lang:type=OperatingSystem</code> MBean on the local server is greater than or equal to the high threshold value.
ServerLowQueueLength	The average ThreadPoolRuntimeMBean.QueueLength value on the local server is less than the low threshold value.
ServerLowThroughput	The average ThreadPoolRuntimeMBean.Throughput value on the local server is less than the low threshold value.
ServerHighQueueLength	The average ThreadPoolRuntimeMBean.QueueLength value on the local server is greater than or equal to the high threshold value.
ServerHighSystemCpuLoadAverage	The average <code>SystemCpuLoad</code> attribute of the <code>java.lang:type=OperatingSystem</code> MBean on the local server is greater than or equal to the high threshold value.
ServerHighPendingUserRequests	The average ThreadPoolRuntimeMBean.PendingUserRequestCount value on the local server is greater than or equal to the high threshold value.
ServerLowSystemCpuLoadAverage	The average <code>SystemCpuLoad</code> attribute of the <code>java.lang:type=OperatingSystem</code> MBean on the local server is less than the low threshold value.
ServerHighHeapFreePercent	The average percentage of free heap on the local server is greater than or equal to the high threshold value.

Table A-2 (Cont.) Summary of Managed Server Scope Smart Rules

After the retention window, the following smart rule returns true if . . .
ServerHighStuckThreads	The average <code>ThreadPoolRuntimeMBean.StuckThreadCount</code> value on the local server is greater than or equal to high threshold value.
ServerHighProcessCpuLoadAverage	The average <code>ProcessCpuLoad</code> value of the <code>java.lang:type=OperatingSystem</code> MBean on the local server is greater than or equal to the high threshold value.
ServerLowSystemLoadAverage	The average <code>SystemLoadAverage</code> value of the <code>java.lang:type=OperatingSystem</code> MBean on the local server is less than the low threshold value.
ServerLowHeapFreePercent	The average percentage of free heap on the local server is less than the low threshold value.
ServerHighIdleThreads	The average <code>ThreadPoolRuntimeMBean.ExecuteThreadIdleCount</code> value on the local server is greater than or equal to the high threshold value.

- [ServerLowIdleThreads](#)
The `ServerLowIdleThreads` smart rule detects if the average number of idle threads is below the specified threshold within the local server in which the rule is running, as indicated by the average value of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute.
- [ServerHighThroughput](#)
The `ServerHighThroughput` smart rule determines whether an increase in throughput exists within the local server in which the rule is running.
- [ServerGenericMetricRule](#)
- [ServerLowPendingUserRequests](#)
The `ServerLowPendingUserRequests` smart rule determines whether the average number of pending user requests within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute.
- [ServerLowProcessCpuLoadAverage](#)
The `ServerLowProcessCpuLoadAverage` smart rule determines whether a reduction exists in the average system load within the local server instance in which the rule is running.
- [ServerHighSystemLoadAverage](#)
The `ServerHighSystemLoadAverage` smart rule determines whether a reduction exists on the average system load within the local server in which the rule is running.
- [ServerLowQueueLength](#)
The `ServerLowQueueLength` smart rule determines whether a reduction exists in the average thread pool queue length within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.QueueLength` metric.

- **ServerLowThroughput**
The `ServerLowThroughput` smart rule determines whether a decrease exists in the average throughput within the local server in which the rule is running.
- **ServerHighQueueLength**
The `ServerHighQueueLength` smart rule determines whether an increase exists in the average thread pool queue length within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.QueueLength` attribute.
- **ServerHighSystemCpuLoadAverage**
The `ServerHighSystemCpuLoadAverage` smart rule determines whether an increase exists in the average system CPU load within the local server in which the rule is running.
- **ServerHighPendingUserRequests**
The `ServerHighPendingUserRequests` smart rule determines whether an increase exists in the number of pending user requests within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute.
- **ServerLowSystemCpuLoadAverage**
- **ServerHighHeapFreePercent**
The `ServerHighHeapFreePercent` smart rule determines whether an increase in heap stress exists within the local server in which the rule is running.
- **ServerHighStuckThreads**
- **ServerHighProcessCpuLoadAverage**
The `ServerHighProcessCpuLoadAverage` smart rule determines whether an decrease exists in the average system load within the local server in which the rule is running.
- **ServerLowSystemLoadAverage**
The `ServerLowSystemLoadAverage` smart rule determines whether a reduction exists in the average system load within the local server in which the rule is running.
- **ServerLowHeapFreePercent**
The `ServerLowHeapFreePercent` smart rule determines whether an increase exists in heap stress within the local server in which the rule is running.
- **ServerHighIdleThreads**
The `ServerHighIdleThreads` smart rule determines whether a reduction in average system load exists within the local server in which the rule is running, by measuring an increase in idle threads as indicated by the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute.

ServerLowIdleThreads

The `ServerLowIdleThreads` smart rule detects if the average number of idle threads is below the specified threshold within the local server in which the rule is running, as indicated by the average value of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This smart rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute is equal to or less than the specified threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) for the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` metric
- Low `ExecuteThreadIdleCount` [threshold value](#)

Syntax

```
wls:ServerLowIdleThreads("period", "duration", idleThreadsLimit)
```

Parameter	Description
<i>period</i>	<p>Sampling rate for <code>ExecuteThreadIdleCount</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<i>duration</i>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<i>idleThreadsLimit</i>	<p>Value established as the low threshold value of the <code>ExecuteThreadIdleCount</code> attribute.</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	5
<i>idleThreadsLimit</i>	0.8

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowIdleThreads("30 seconds","10 minutes",5)
```

This example smart rule:

1. Samples the value of the `ExecuteThreadIdleCount` metric from the local server instance every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `ExecuteThreadIdleCount` value, over the last 5 minutes, is less than or equal to 0.8 on this server instance.

ServerHighThroughput

The `ServerHighThroughput` smart rule determines whether an increase in throughput exists within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This smart rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.Throughput` attribute over the specified retention window is greater than or equal to the high threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `ThreadPoolRuntimeMBean.Throughput` attribute.
- High Throughput [threshold value](#)

Syntax

```
wls:ServerHighThroughput("period", "duration", throughputLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for Throughput values, expressed as a String. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a String. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>throughputLimit</i>	Value established as the high threshold value of the Throughput attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	10
<i>throughputLimit</i>	100

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighThroughput("30 seconds","10 minutes",100)
```

This example smart rule:

1. Samples the value of the `Throughput` metric from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `Throughput` value, over the last 10 minutes, is greater than or equal to 100 on this server instance.

ServerGenericMetricRule

The `ServerGenericMetricRule` smart rule is a general server scope smart rule that you can use to observe trends of any JMX metric that is published through the Server Runtime MBean Server and that is not provided by the other server scope smart rules. This smart rule allows you to collect the average value of the metric across a recent time interval and compare it to a threshold value using a specified comparison operator.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This smart rule returns `true` if the average value of the metric meets or exceeds the specified threshold value.

To use this smart rule, specify:

- A valid JMX `ObjectName` or `ObjectName` pattern
- A Java EL expression that retrieves a value on each matching MBean instance, where the expression is an attribute expression relative to each MBean.
- A boolean comparison operator using the specified comparison operator
- A **threshold value** against which the selected attribute is compared
- The **sampling rate** and **retention window** of the metric.

Syntax

```
wls:ServerGenericMetricRule("instancePattern", "attribute", "operation",
thresholdValue, "period", "duration")
```

Parameter	Description
<code>instancePattern</code>	A valid JMX <code>ObjectName</code> or <code>ObjectName</code> pattern
<code>attribute</code>	A Java EL expression that retrieves a value on each MBean instance that matches <code>instancePattern</code> , where the expression is an attribute expression relative to each MBean. For example, if the MBean is the <code>ServerRuntimeMBean</code> , the expression <code>'OpenSocketsCurrentCount'</code> obtains the value of the <code>OpenSocketsCurrentCount</code> attribute of the <code>ServerRuntimeMBean</code> . By contrast, the expression <code>'HealthState.State'</code> obtains the <code>State</code> value of the <code>HealthState</code> child object of that MBean.

Parameter	Description
<code>operation</code>	A boolean comparison operator: <code><</code> , <code><=</code> , <code>=</code> , <code>>=</code> , or <code>></code> .
<code>thresholdValue</code>	A threshold value with which to compare the selected attribute using the specified comparison operator.
<code>period</code>	Sampling rate for metric values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. See retention window for more information about specifying this parameter.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>instancePattern</code>	<code>java.lang:type=OperatingSystem</code>
<code>attribute</code>	<code>ProcessCpuLoad</code>
<code>operation</code>	<code>>=</code>
<code>thresholdValue</code>	<code>0.9</code>
<code>period</code>	<code>30</code>
<code>duration</code>	<code>10</code>

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerGenericMetricRule("java.lang:type=OperatingSystem", "ProcessCpuLoad", ">=", 0.9, "30 seconds", "10 minutes")
```

The smart rule:

1. Samples the value of the `ProcessCpuLoad` metric on the targeted server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, this smart rule evaluates to `true` in the following condition:

The average value of `ProcessCpuLoad` on the `OperatingSystemMXBean`, over the last 10 minutes, is greater than or equal to 0.9 on this server instance.

ServerLowPendingUserRequests

The `ServerLowPendingUserRequests` smart rule determines whether the average number of pending user requests within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This smart rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute over the specified retention window is less than the low threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute.
- Low `PendingUserRequestCount` [threshold value](#)

Syntax

```
wls:ServerLowPendingUserRequests("period", "duration", pendingRequestsLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for <code>PendingUserRequestCount</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>pendingRequestsLimit</i>	Value established as the low threshold value of the <code>PendingUserRequestCount</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	15

Parameter	Value
<code>pendingRequestsLimit</code>	5

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowPendingUserRequests("30 seconds","15 minutes",5)
```

This example smart rule:

1. Samples the value of the `PendingUserRequestCount` metric from the local server instance every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, evaluates to `true` if the following condition exists:

The average `PendingUserRequestCount` value, over the last 15 minutes, is less than 5 on this server instance.

ServerLowProcessCpuLoadAverage

The `ServerLowProcessCpuLoadAverage` smart rule determines whether a reduction exists in the average system load within the local server instance in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ProcessCpuLoad` metric of the `java.lang:type=OperatingSystem` MBean over the specified time interval is less than a specified threshold value.

Note:

The value of the `ProcessCpuLoad` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMBean.html#getProcessCpuLoad-->.

To use this smart rule, specify:

- The **sampling rate** and **retention window** of the `ProcessCpuLoad` attribute.
- Low `ProcessCpuLoad` **threshold value**

Syntax

```
wls:ServerLowProcessCpuLoadAverage("period", "duration", processCpuLoadLimit)
```

Parameter	Description
<code>period</code>	<p>Sampling rate for <code>ProcessCpuLoad</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>processCpuLoadLimit</code>	Value established as the low threshold value of the <code>ProcessCpuLoad</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>period</code>	30
<code>duration</code>	15
<code>processCpuLoadLimit</code>	0.2

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowProcessCpuLoadAverage("30 seconds","15 minutes",0.2)
```

This example smart rule:

1. Samples the value of the `ProcessCpuLoad` metric from the local server instance every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `ProcessCpuLoad` value, over the last 15 minutes, is less than 0.2 on this server instance.

ServerHighSystemLoadAverage

The `ServerHighSystemLoadAverage` smart rule determines whether a reduction exists on the average system load within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `SystemLoadAverage` metric from the `java.lang:type=OperatingSystem` MBean on the local server instance over specified interval is greater than or equal to a specific high threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `SystemLoadAverage` attribute.
- High `SystemLoadAverage` [threshold value](#)

 **Note:**

The value of the `SystemLoadAverage` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <http://docs.oracle.com/javase/8/docs/api/java/lang/management/OperatingSystemMXBean.html#getSystemLoadAverage-->.

Syntax

```
wls:ServerHighSystemLoadAverage("period", "duration", loadLimit)
```

Parameter	Description
<i>period</i>	<p>Sampling rate for <code>SystemLoadAverage</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<i>duration</i>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<i>loadLimit</i>	<p>Value established as the high threshold value of the <code>SystemLoadAverage</code> attribute.</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	5
<i>loadLimit</i>	0.8

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighSystemLoadAverage("30 seconds", "5 minutes", 0.8)
```

This example smart rule:

1. Samples the value of the `SystemLoadAverage` metric on the local server instance every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `SystemLoadAverage` value, over the last 5 minutes, is greater than or equal to 0.8 collected on this server instance.

ServerLowQueueLength

The `ServerLowQueueLength` smart rule determines whether a reduction exists in the average thread pool queue length within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.QueueLength` metric.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.QueueLength` attribute on the local server instance over specified interval is less than a specific low threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `ThreadPoolRuntimeMBean.QueueLength` attribute.
- Low `QueueLength` [threshold value](#)

Syntax

```
wls:ServerLowQueueLength("period", "duration", queueLengthLimit)
```

Parameter	Description
<code>period</code>	<p>Sampling rate for <code>QueueLength</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>queueLengthLimit</code>	<p>Value established as the low threshold value of the <code>QueueLength</code> attribute.</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	15
<i>queueLengthLimit</i>	5

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowQueueLength("30 seconds","15 minutes",5)
```

This example smart rule:

1. Samples the value of the `QueueLength` metric from the local server instance every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `QueueLength` value, over the last 15 minutes, is less than 5 on this server instance.

ServerLowThroughput

The `ServerLowThroughput` smart rule determines whether a decrease exists in the average throughput within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.Throughput` attribute on the local server over the specified interval is less than the specified low threshold value.

To use this smart rule, specify:

- The **sampling rate** and **retention window** of the `ThreadPoolRuntimeMBean.Throughput` attribute.
- Low Throughput **threshold value**

Syntax

```
wls:ServerLowThroughput("period", "duration", throughputLimit)
```

Parameter	Description
<code>period</code>	Sampling rate for <code>Throughput</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>throughputLimit</code>	Value established as the low threshold value of the <code>Throughput</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>period</code>	30
<code>duration</code>	15
<code>idleThreadsLimit</code>	5

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowThroughput("30 seconds","15 minutes",5)
```

This example smart rule:

1. Samples the value of the `Throughput` metric from the local server instance every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `Throughput` value, over the last 15 minutes, is less than 5 on this server instance.

ServerHighQueueLength

The `ServerHighQueueLength` smart rule determines whether an increase exists in the average thread pool queue length within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.QueueLength` attribute.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.QueueLength` attribute over a specific time interval is greater than or equal to a specific high threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `ThreadPoolRuntimeMBean.QueueLength` attribute.
- High `QueueLength` [threshold value](#)

Syntax

```
wls:ServerHighQueueLength("period", "duration", queueLengthLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for <code>QueueLength</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>queueLengthLimit</i>	Value established as the high threshold value of the <code>QueueLength</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	10
<i>queueLengthLimit</i>	100

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighQueueLength("30 seconds","10 minutes",100)
```

This example smart rule:

1. Samples the value of the `QueueLength` metric from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `QueueLength` value, over the last 10 minutes, is greater than or equal to 100 on this server instance.

ServerHighSystemCpuLoadAverage

The `ServerHighSystemCpuLoadAverage` smart rule determines whether an increase exists in the average system CPU load within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `SystemCpuLoad` attribute of the `java.lang:type=OperatingSystem` MBean over a specific time interval is greater than or equal to a specific high threshold.

Note:

The value of the `SystemCpuLoad` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMBean.html#getSystemCpuLoad-->.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `SystemCpuLoad` attribute.
- High `SystemCpuLoad` [threshold value](#)

Syntax

```
wls:ServerHighSystemCpuLoadAverage("period", "duration", systemCpuLoadLimit)
```

Parameter	Description
<code>period</code>	<p>Sampling rate for <code>SystemCpuLoad</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>

Parameter	Description
<code>systemCpuLoadLimit</code>	Value established as the high threshold value of the <code>SystemCpuLoad</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>period</code>	30
<code>duration</code>	10
<code>systemCpuLoadLimit</code>	0.8

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighSystemCpuLoadAverage("30 seconds","10 minutes",0.8)
```

This example smart rule:

1. Samples the value of the `SystemCpuLoad` metric from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `SystemCpuLoad` value, over the last 10 minutes, is greater than or equal to 0.8 on this server instance.

ServerHighPendingUserRequests

The `ServerHighPendingUserRequests` smart rule determines whether an increase exists in the number of pending user requests within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute over a specific interval is greater than or equal to a specific threshold value.

To use this smart rule, specify:

- The **sampling rate** and **retention window** of the `ThreadPoolRuntimeMBean.PendingUserRequestCount` attribute.
- High `PendingUserRequestCount` **threshold value**

Syntax

```
wls:ServerHighPendingUserRequests("period", "duration", pendingRequestsLimit)
```

Parameter	Description
<code>period</code>	Sampling rate for <code>PendingUserRequestCount</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<code>duration</code>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<code>pendingRequestsLimit</code>	Value established as the high threshold value of the <code>PendingUserRequestCount</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>period</code>	30
<code>duration</code>	10
<code>pendingRequestsLimit</code>	100

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighPendingUserRequests("30 seconds","10 minutes",100)
```

This example smart rule:

1. Samples the value of the `PendingUserRequestCount` metric from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `PendingUserRequestCount` value, over the last 10 minutes, is greater than or equal to 100 on this server instance.

ServerLowSystemCpuLoadAverage

The `ServerLowSystemCpuLoadAverage` smart rule determines whether a reduction exists in the average system CPU load within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `SystemCpuLoad` metric of the `java.lang:type=OperatingSystem` MBean over a specific interval is less than the specified low threshold value.

Note:

The value of the `SystemCpuLoad` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMBean.html#getSystemCpuLoad-->.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `SystemCpuLoad` attribute.
- Low `SystemCpuLoad` [threshold value](#)

Syntax

```
wls:ServerLowSystemCpuLoadAverage("period", "duration", systemCpuLoadLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for <code>SystemCpuLoad</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>systemCpuLoadLimit</i>	Value established as the low threshold value of the <code>SystemCpuLoad</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	15
<i>systemCpuLoadLimit</i>	0.8

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowSystemCpuLoadAverage("30 seconds", "15 minutes", 0.8)
```

This example smart rule:

1. Samples the value of the `SystemCpuLoad` metric from the local server instance every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `SystemCpuLoad` value, over the last 15 minutes, is less than 0.8 on this server instance.

ServerHighHeapFreePercent

The `ServerHighHeapFreePercent` smart rule determines whether an increase in heap stress exists within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average `JVMRuntimeMBean.HeapFreePercent` value over the specific time interval is greater than or equal to the specified high threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `JVMRuntimeMBean.HeapFreePercent` attribute.
- High JVM free heap percentage [threshold value](#)

Syntax

```
wls:ServerHighHeapFreePercent("period", "duration", percentFreeLimit)
```

Parameter	Description
<code>period</code>	<p>Sampling rate for JVM free heap percentage values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>percentFreeLimit</code>	<p>Value established as the high threshold of the JVM free heap percentage, specified as a <code>float</code> value between 0.0 and 100.0</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	10
<i>percentFreeLimit</i>	60

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighHeapFreePercent("30 seconds","10 minutes",60)
```

This example smart rule:

1. Samples the value of the JVM free heap percentage from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average JVM free heap value, over the last 10 minutes, is greater than or equal to 60 per cent on this server instance.

ServerHighStuckThreads

The `ServerHighStuckThreads` smart rule determines whether an increase exists on server stress based on the average number of stuck threads within the local server in which the rule is running, as indicated by the value of the `ThreadPoolRuntimeMBean.StuckThreadCount` attribute.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.StuckThreadCount` attribute over a specific time interval is greater than or equal to the specified threshold value.

To use this smart rule, specify:

- The **sampling rate** and **retention window** of the `ThreadPoolRuntimeMBean.StuckThreadCount` attribute.
- High `StuckThreadCount` **threshold value**

Syntax

```
wls:ServerHighStuckThreads("period", "duration", stuckThreadsLimit)
```

Parameter	Description
<i>period</i>	<p>Sampling rate for <code>StuckThreadCount</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<i>duration</i>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<i>stuckThreadsLimit</i>	Value established as the high threshold value of the <code>StuckThreadCount</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	10
<i>stuckThreadsLimit</i>	5

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighStuckThreads("30 seconds","10 minutes",5)
```

This example smart rule:

1. Samples the value of the `StuckThreadCount` metric from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `StuckThreadCount` value, over the last 10 minutes, is greater than or equal to 5 on this server instance.

ServerHighProcessCpuLoadAverage

The `ServerHighProcessCpuLoadAverage` smart rule determines whether an decrease exists in the average system load within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average `ProcessCpuLoad` value of the `java.lang:type=OperatingSystem` MBean over the specified interval is greater than or equal to the specified threshold.

Note:

The value of the `ProcessCpuLoad` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <https://docs.oracle.com/javase/8/docs/jre/api/management/extension/com/sun/management/OperatingSystemMBean.html#getProcessCpuLoad-->.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `ProcessCpuLoad` attribute.
- High `ProcessCpuLoad` [threshold value](#)

Syntax

```
wls:ServerHighProcessCpuLoadAverage("period", "duration", processCpuLoadLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for <code>ProcessCpuLoad</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>processCpuLoadLimit</i>	Value established as the high threshold value of the <code>ProcessCpuLoad</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	5
<i>processCpuLoadLimit</i>	0.8

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighProcessCpuLoadAverage("30 seconds", "5 minutes", 0.8)
```

This example smart rule:

1. Samples the value of the `ProcessCpuLoad` metric from the local server instance every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `ProcessCpuLoad` value, over the last 5 minutes, is greater than or equal to 0.8 on this server instance.

ServerLowSystemLoadAverage

The `ServerLowSystemLoadAverage` smart rule determines whether a reduction exists in the average system load within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the value of the `SystemLoadAverage` metric of the `java.lang:type=OperatingSystem` MBean over a specified interval is less than the specified low threshold value.



Note:

The value of the `SystemLoadAverage` metric is platform-specific and is not available on all platforms. The MBean attribute from which this metric originates is described at <http://docs.oracle.com/javase/8/docs/api/java/lang/management/OperatingSystemMBean.html#getSystemLoadAverage-->.

To use this smart rule, specify:

- The **sampling rate** and **retention window** of the `SystemLoadAverage` attribute.
- Low `SystemLoadAverage` **threshold value**

Syntax

```
wls:ServerLowSystemLoadAverage("period", "duration", loadLimit)
```

Parameter	Description
<code>period</code>	<p>Sampling rate for <code>SystemLoadAverage</code> values, expressed as a <code>String</code>. For example, <code>30s</code> specifies that this metric is sampled every 30 seconds.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>30s</code>. <p>See sampling rate for more information about specifying this parameter.</p>
<code>duration</code>	<p>Period of time for which collected samples are retained, expressed as a <code>String</code>.</p> <ul style="list-style-type: none"> The default time unit is seconds. The default value is <code>10m</code>. <p>See retention window for more information about specifying this parameter.</p>
<code>loadLimit</code>	<p>Value established as the low threshold value of the <code>SystemLoadAverage</code> attribute.</p>

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<code>period</code>	30
<code>duration</code>	15
<code>loadLimit</code>	0.2

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowSystemLoadAverage("30 seconds","15 minutes",0.2)
```

This example smart rule:

1. Samples the value of the `SystemLoadAverage` metric from the local server instance every 30 seconds over a retention window of 15 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `SystemLoadAverage` value, over the last 15 minutes, is less than 0.2 on this server instance.

ServerLowHeapFreePercent

The `ServerLowHeapFreePercent` smart rule determines whether an increase exists in heap stress within the local server in which the rule is running.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average `JVMRuntimeMBean.HeapFreePercent` value over the specified time interval is less than the specified low threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `JVMRuntimeMBean.HeapFreePercent` attribute.
- Low Java free heap percentage [threshold value](#)

Syntax

```
wls:ServerLowHeapFreePercent("period", "duration", percentFreeLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for Java free heap percentage values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>percentFreeLimit</i>	Value established as the low threshold value of the Java free heap percentage.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	5
<i>percentFreeLimit</i>	20

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerLowHeapFreePercent("30 seconds","5 minutes",20)
```

This example smart rule:

1. Samples the value of the Java free heap percentage from the local server instance every 30 seconds over a retention window of 5 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average Java free heap percentage value, over the last 5 minutes, is less than 20 per cent on this server instance.

ServerHighIdleThreads

The `ServerHighIdleThreads` smart rule determines whether a reduction in average system load exists within the local server in which the rule is running, by measuring an increase in idle threads as indicated by the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute.

You can target this smart rule on either a Managed Server or an Administration Server.

Group: Server

Description

This rule returns `true` if the average value of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute over the specified retention window is greater than or equal to the specified threshold value.

To use this smart rule, specify:

- The [sampling rate](#) and [retention window](#) of the `ThreadPoolRuntimeMBean.ExecuteThreadIdleCount` attribute.
- High `ExecuteThreadIdleCount` [threshold value](#)

Syntax

```
wls:ServerHighIdleThreads("period", "duration", idleThreadsLimit)
```

Parameter	Description
<i>period</i>	Sampling rate for <code>ExecuteThreadIdleCount</code> values, expressed as a <code>String</code> . For example, <code>30s</code> specifies that this metric is sampled every 30 seconds. <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>30s</code>. See sampling rate for more information about specifying this parameter.
<i>duration</i>	Period of time for which collected samples are retained, expressed as a <code>String</code> . <ul style="list-style-type: none"> • The default time unit is seconds. • The default value is <code>10m</code>. See retention window for more information about specifying this parameter.
<i>idleThreadsLimit</i>	Value established as the high threshold value of the <code>ExecuteThreadIdleCount</code> attribute.

Example

The smart rule shown in this example uses the following input parameters:

Parameter	Value
<i>period</i>	30
<i>duration</i>	10
<i>idleThreadsLimit</i>	20

The smart rule that uses the preceding parameters is expressed as follows:

```
wls:ServerHighIdleThreads("30 seconds","10 minutes",20)
```

This example smart rule:

1. Samples the value of the `ExecuteThreadIdleCount` metric from the local server instance every 30 seconds over a retention window of 10 minutes.
2. At the end of the retention window, fires the associated action if the following condition evaluates to `true`:

The average `ExecuteThreadIdleCount` value, over the last 10 minutes, is greater than or equal to 20 on this server instance.

B

WLDF Beans and Functions Reference

The WebLogic Diagnostics Framework (WLDF) provides a set of beans and functions that can be used in collected metrics policy expressions to obtain access to common WebLogic Server JMX data sources.

- [WLDF Beans Reference](#)
WLDF includes several beans that can be used in collected metrics policy expressions to access statistics that provide information about active cluster objects, MBeans, instrument event fields, and more.
- [Functions Reference](#)
WLDF includes a set of functions that can be used in policy expressions to simplify the extraction or querying of data.

WLDF Beans Reference

WLDF includes several beans that can be used in collected metrics policy expressions to access statistics that provide information about active cluster objects, MBeans, instrument event fields, and more.



Note:

WebLogic Server Multitenant domain partitions, resource groups, resource group templates, and virtual targets are deprecated in WebLogic Server 12.2.1.4.0 and will be removed in the next release.

- [clusterRuntime](#)
The `clusterRuntime` bean provides cluster-wide access to statistics for active clusters in the domain.
- [domainRuntime](#)
The `domainRuntime` bean provides access to MBeans registered in the Domain Runtime MBean Server.
- [instrumentationEvent](#)
The `instrumentationEvent` bean provides access to instrumentation event fields in instrumentation policy expressions.
- [log](#)
Used in log policy expressions, the `log` bean provides access to log message fields.
- [partition](#)
The `partition` bean is a convenience mechanism for obtaining partition scope metrics. This bean is available only to policies that are configured in a diagnostic system module that is deployed in the same partition to which this bean is scoped.
- [platform](#)
- [resource](#)

- [runtime](#)
The `runtime` bean provides access to MBeans registered in the WebLogic Server Runtime MBean Server.

clusterRuntime

The `clusterRuntime` bean provides cluster-wide access to statistics for active clusters in the domain.

Attributes

Name	Description
<code>clusters</code>	Provides a map of beans that represent active cluster objects within the domain, keyed by cluster name. Type: <code>interface java.util.Map</code>
<code>name</code>	The name of the cluster. Type: <code>class java.lang.String</code>

Methods

Name	Description
<code>query</code>	<p>Performs a query for a set of MBean attribute values based on an Object Name pattern and an attribute expression.</p> <p>Parameters:</p> <ul style="list-style-type: none"> • <code>onPattern</code> A valid JMX Object Name, or Object Name pattern. • <code>attributePattern</code> A EL expression that is used to retrieve a value from each matching MBean instance, where the expression is an attribute expression relative to each MBean. <p>For example, if the MBean is the <code>ServerRuntimeMBean</code>, the expression <code>'OpenSocketsCurrentCount'</code> obtains the value of the <code>OpenSocketsCurrentCount</code> attribute. By contrast, <code>'HealthState.State'</code> obtains the <code>State</code> value of the <code>HealthState</code> child object.</p> <p>Return values: Returns a set of values matching the specified ObjectName pattern and attribute expression. These results can be fed to the <code>wls:extract</code> function for maintaining an in-memory history of values.</p>
<code>getClusters</code>	Provides a map of beans that represent active cluster objects within the domain.
<code>getAttribute</code>	<p>Obtains a single attribute value from an MBean source.</p> <p>Parameters:</p> <ul style="list-style-type: none"> • <code>objectNamePattern</code> A JMX ObjectName or ObjectName pattern that must resolve to a single MBean instance. • <code>attribute</code> The MBean attribute value to obtain. <p>Returns Values: Returns the attribute value matching the specified JMX ObjectName.</p>

domainRuntime

The `domainRuntime` bean provides access to MBeans registered in the Domain Runtime MBean Server.

Attributes

Name	Description
<code>domain</code>	The root DomainRuntimeMBean in the Domain Runtime MBean Server.
<code>name</code>	The bean name. Type: <code>class java.lang.String</code>
<code>serverRuntimes</code>	Returns the array of active ServerRuntimeMBean instances in the domain. Type: <code>class weblogic.management.runtime.ServerRuntimeMBean[]</code>

Methods

Name	Description
<code>query</code>	<p>Performs a query for a set of MBean attribute values based on an Object Name pattern and an attribute expression.</p> <p>Parameters:</p> <ul style="list-style-type: none"> <code>onPattern</code> A valid JMX Object Name, or Object Name pattern) <code>attributePattern</code> A EL expression that is used to retrieve a value from each matching MBean instance, where the expression is an attribute expression relative to each MBean. For example, if the MBean is the ServerRuntimeMBean, the expression <code>'OpenSocketsCurrentCount'</code> obtains the value of the <code>OpenSocketsCurrentCount</code> attribute. By contrast, <code>'HealthState.State'</code> obtains the <code>State</code> value of the <code>HealthState</code> child object. <p>Return values: Returns a set of values matching the specified Object Name pattern and attribute expression. These results can be fed to the wls:extract function for maintaining an in-memory history of values.</p>
<code>query</code>	<p>Executes a JMX query against a set of targets within the Domain Runtime MBean Server.</p> <p>Parameters:</p> <ul style="list-style-type: none"> <code>targets</code> A list of server or cluster targets specified as a comma-delimited <code>String</code> <code>onPattern</code> A valid JMX Object Name or Object Name pattern <code>expression</code> A EL expression that is used to retrieve a value on each matching MBean instance <p>Return values: Returns a set of values matching the specified Object Name pattern and attribute expression, across the specified target names. The target names can be a valid WebLogic Server instance or cluster in the domain. These results can be fed to the wls:extract function for maintaining an in-memory history of values.</p>

Name	Description
lookupServerRuntime	<p>Returns the ServerRuntimeMBean for the named server instance, or null if not specified.</p> <p>Parameter:</p> <ul style="list-style-type: none"> serverName The name of the ServerRuntimeMBean to look up <p>Return values: Returns a value matching the specified Object Name pattern and attribute expression.</p>
getAttribute	<p>Obtains a single attribute value from an MBean source.</p> <p>Parameters:</p> <ul style="list-style-type: none"> objectNamePattern A JMX ObjectName or ObjectName pattern that must resolve to a single MBean instance. attribute The MBean attribute value to obtain. <p>Returns Values: Returns the attribute value matching the specified JMX ObjectName.</p>

instrumentationEvent

The `instrumentationEvent` bean provides access to instrumentation event fields in instrumentation policy expressions.

Attributes

Name	Description
timeStamp	The timestamp value associated with the event creation. Type: class java.lang.Long
contextId	The diagnostic context ID associated with the instrumentation event. Type: class java.lang.String
txId	The JTA transaction ID associated with the instrumentation event. Type: class java.lang.String
userId	The user name associated with the request for which the instrumentation event is generated. Type: class java.lang.String
eventType	The instrumentation event type. Type: class java.lang.String
domain	The name of the current domain. Type: class java.lang.String
server	The name of the server on which the instrumentation event occurred. Type: class java.lang.String
scope	The instrumentation scope for this event. Type: class java.lang.String
module	The name of the module in which the instrumentation event rule is defined. Type: class java.lang.String

Name	Description
monitor	The instrumentation monitor that generated the instrumentation event. Type: class java.lang.String
fileName	The source file name containing the code that generated the instrumentation event. Type: class java.lang.String
lineNumber	The line number in the source file where the instrumentation event originated. Type: class java.lang.Integer
className	The class name where the instrumentation event originated. Type: class java.lang.String
methodName	The method name where the instrumentation event originated. Type: class java.lang.String
methodDesc	The description of the method that generated the instrumentation event. Type: class java.lang.String
arguments	The arguments passed into the method that generated the instrumentation event. Type: class java.lang.String
returnValue	The return value for the method that generated the instrumentation event. Type: class java.lang.String
payload	The payload associated with the instrumentation event. Type: class java.lang.Object
contextPayload	The context payload associated with the instrumentation event. Type: class java.lang.String
dyeVector	The dye vector associated with the instrumentation event. Type: class java.lang.Long
threadName	The name of the thread that generated the instrumentation event. Type: class java.lang.String
partitionId	The partition ID associated with the instrumentation event. Type: class java.lang.String
partitionName	The partition name associated with the instrumentation event. Type: class java.lang.String

Example

The following are examples of using the `instrumentationEvent` bean in an EL policy expression to access instrumentation event fields:

```
instrumentationEvent.monitor == 'Servlet_Around_Service'  
  
instrumentationEvent.getMonitor() == 'Servlet_Around_Service'  
  
instrumentationEvent.monitor.contains('Servlet_')
```

log

Used in log policy expressions, the `log` bean provides access to log message fields.

Attributes

Name	Description
<code>timestamp</code>	The timestamp indicating when the log message was created. Type: <code>class java.lang.Long</code>
<code>formattedDate</code>	The formatted date string. Type: <code>class java.lang.String</code>
<code>messageId</code>	The message ID of the log entry. Type: <code>class java.lang.String</code>
<code>machineName</code>	The machine name on which the log entry was created. Type: <code>class java.lang.String</code>
<code>serverName</code>	The server name on which the log entry was created. Type: <code>class java.lang.String</code>
<code>threadName</code>	The thread name in which the logged event was created. Type: <code>class java.lang.String</code>
<code>userId</code>	The ID of the user who generated the logged event. Type: <code>class java.lang.String</code>
<code>transactionId</code>	The JTA transaction ID associated with the logged event. Type: <code>class java.lang.String</code>
<code>severity</code>	The severity level for the log message. Type: <code>class java.lang.Integer</code>
<code>severityString</code>	The severity string for the log message. Type: <code>class java.lang.String</code>
<code>subsystem</code>	The name of the subsystem that generated the log message. Type: <code>class java.lang.String</code>
<code>logMessage</code>	The message content of the log entry. Type: <code>class java.lang.String</code>
<code>diagnosticContextId</code>	The diagnostic context ID associated with the logged event. Type: <code>class java.lang.String</code>
<code>supplementalAttributes</code>	The name-value pairs of supplemental attributes that are included in the log entries. Type: <code>class java.util.Properties</code>
<code>partitionId</code>	The partition ID associated with the logged event. Type: <code>class java.lang.String</code>
<code>partitionName</code>	The partition ID associated with the logged event. Type: <code>class java.lang.String</code>

Example

The following are examples of using the `log` bean in an EL policy expression to access log message fields:

```
log.logMessage.contains("Part of a message")

log.getLogMessage().contains("Part of a message")

log.messageId == "BEA-000365"

log.messageId.endsWith('000365')
```

partition

The `partition` bean is a convenience mechanism for obtaining partition scope metrics. This bean is available only to policies that are configured in a diagnostic system module that is deployed in the same partition to which this bean is scoped.

Attributes

Name	Description
ID	The globally unique identifier associated with the named partition.
root	The PartitionMBean configuration root of the partition. Note that this is the read-only PartitionMBean that is registered in the WebLogic Server Runtime MBean Server. Type: <code>interface weblogic.management.configuration.PartitionMBean</code>
runtime	The PartitionRuntimeMBean root of the partition in which the diagnostic system module is deployed. Type: <code>interface weblogic.management.runtime.PartitionRuntimeMBean</code>

Methods

Name	Description
query	Issues partition scoped JMX queries, filtered by the partition name. Parameters: <ul style="list-style-type: none"> <code>onPattern</code> A JMX Object Name or Object Name pattern <code>attributePattern</code> EL expression for accessing a value within the set of MBeans returned in the query set Return values: Returns a set of values matching the specified object name pattern and attribute expression. These results can be fed to the wls:extract function for maintaining an in-memory history of values.

Name	Description
getAttribute	<p>Obtains a single attribute value from an MBean source.</p> <p>Parameters:</p> <ul style="list-style-type: none"> objectNamePattern A JMX ObjectName or ObjectName pattern that must resolve to a single MBean instance. attribute The MBean attribute value to obtain. <p>Returns Values: Returns the attribute value matching the specified JMX ObjectName.</p>

platform

The `platform` bean obtain values from MBeans that are exposed through the JVM's platform MBean server. (Note that WebLogic Server uses the JVM's platform MBean server to contain the WebLogic run-time MBeans by default. As such, the platform MBean server provides access to platform MXBeans, WebLogic run-time MBeans, and WebLogic configuration MBeans that are on a single server instance.)

Attributes

Name	Description
name	The name of the platform bean ("platform")

Methods

Name	Description
query	<p>Performs a query for a set of MBean attribute values based on an Object Name pattern and an attribute expression.</p> <p>Parameters:</p> <ul style="list-style-type: none"> onPattern A valid JMX Object Name, or Object Name pattern) attributePattern A EL expression that is used to retrieve a value from each matching MBean instance, where the expression is an attribute expression relative to each MBean. For example, if the MBean is the ServerRuntimeMBean, the expression 'OpenSocketsCurrentCount' obtains the value of the OpenSocketsCurrentCount attribute. By contrast, 'HealthState.State' obtains the State value of the HealthState child object. <p>Return values: A set of values matching the specified Object Name pattern and attribute expression. These results can be fed to the wls:extract function for maintaining an in-memory history of values.</p>

Name	Description
getAttribute	<p>Obtains a single attribute value from an MBean source.</p> <p>Parameters:</p> <ul style="list-style-type: none"> objectNamePattern A JMX ObjectName or ObjectName pattern that must resolve to a single MBean instance. attribute The MBean attribute value to obtain. <p>Returns Values: Returns the attribute value matching the specified JMX ObjectName.</p>

resource

The `resource` bean provides access to beans and state information within a diagnostic system module deployment. Access is restricted to policies that are configured within the same diagnostic system module. That is, this bean cannot obtain access to beans and state information from policies that are configured in other diagnostic system modules. This bean is used for policy-chaining.

Attributes

Name	Description
watches	<p>A map of currently configured policies within the same diagnostic system module deployment.</p> <p>Type: <code>interface java.util.Map</code></p>

runtime

The `runtime` bean provides access to MBeans registered in the WebLogic Server Runtime MBean Server.

Attributes

Name	Description
domain	<p>The root <code>DomainMBean</code> in the local WebLogic Server Runtime MBean Server.</p> <p>Type: <code>interface weblogic.management.configuration.DomainMBean</code></p>
name	<p>The bean name.</p> <p>Type: <code>class java.lang.String</code></p>
serverRuntime	<p>The root <code>ServerRuntimeMBean</code> in the local WebLogic Server Runtime MBean Server.</p> <p>Type: <code>interface weblogic.management.runtime.ServerRuntimeMBean</code></p>

Methods

Name	Description
query	<p>Performs a query for a set of MBean attribute values based on an Object Name pattern and an attribute expression.</p> <p>Parameters:</p> <ul style="list-style-type: none"> onPattern A valid JMX Object Name, or Object Name pattern) attributePattern A EL expression that is used to retrieve a value from each matching MBean instance, where the expression is an attribute expression relative to each MBean. For example, if the MBean is the ServerRuntimeMBean, the expression 'OpenSocketsCurrentCount' obtains the value of the OpenSocketsCurrentCount attribute. By contrast, 'HealthState.State' obtains the State value of the HealthState child object. <p>Return values: A set of values matching the specified Object Name pattern and attribute expression. These results can be fed to the wls:extract function for maintaining an in-memory history of values.</p>
getAttribute	<p>Obtains a single attribute value from an MBean source.</p> <p>Parameters:</p> <ul style="list-style-type: none"> objectNamePattern A JMX ObjectName or ObjectName pattern that must resolve to a single MBean instance. attribute The MBean attribute value to obtain. <p>Returns Values: Returns the attribute value matching the specified JMX ObjectName.</p>

Functions Reference

WLDF includes a set of functions that can be used in policy expressions to simplify the extraction or querying of data.

- [wls:tableChanges](#)
The `wls:tableChanges` function takes a table of input values and generates an output table of difference vectors, one for each input vector.
- [wls:tableAverages](#)
- [wls:extract](#)
- [wls:average](#)
- [wls:changes](#)
- [wls:aliveServersCount](#)

wls:tableChanges

The `wls:tableChanges` function takes a table of input values and generates an output table of difference vectors, one for each input vector.

This function throws an `IllegalArgumentException` if the input either:

- Is not a two-dimensional table
- Contains non-numeric values

Parameters

Name	Description
<code>inputTable</code>	The input table of numeric values, where each row is typically a time series of values from the same metric instance.

wls:tableAverages

The `wls:tableAverages` function performs a matrix reduction on an input table of values, computing the average of each row in the table and producing a vector of averages, one for each row in the table. Typically each row in the table represents a time series of values from a particular metric instance.

This function throws an `IllegalArgumentException` if the input either:

- Is not a two-dimensional table
- Contains non-numeric values

Parameters

Name	Description
<code>valuesTable</code>	The input table of numeric values, where each row is typically a time series of values from the same metric instance.

wls:extract

The `wls:extract` function extracts a table of time series from a specified set of input sources, based on a specified sampling rate schedule and time window. The input source can be one of the following:

- The output from a `query()` operation from a JMX bean. For example:

```
wls.runtime.query('com.bea:Type=ServletRuntime,*', 'ExecutionTimeAverage')
```
- An EL expression, as a `String`. For example:

```
wls.runtime.JVMRuntime.heapFreePercent
```

Parameters

Name	Description
<code>inputExpression</code>	The bean metric to be sampled.

Name	Description
<code>schedule</code>	The sampling rate of the metric, specified as a string, in hours, minutes, or seconds (the default).
<code>duration</code>	The required sampling window of the metric, specified as a string, in hours, minutes, or seconds (the default)

The `schedule` and `duration` parameters can be specified in seconds, minutes, or hours, and are specified as strings using the following syntax:

```
amount[unit]
```

In the preceding syntax:

- `amount` represents an integer.
- `[unit]` represents `seconds`, `minutes`, or `hours`. Each can be abbreviated to the first letter. For example: `seconds` can be abbreviated to `s`.
- You may include a space character between `amount` and `unit`.

For example, any of the following can be used to specify five seconds:

- `5seconds`
- `5 sec`
- `5s`
- `5snds`

wls:average

The `wls:average` function computes an average value based on set of numeric input values. This function returns the scalar average of the input vector, or `Double.NaN` if the input is empty. If the input contains any non-numeric values, an `IllegalArgumentException` is thrown.



Note:

The `wls:average` function is different from the EL-provided `average()` operation.

Parameters

Name	Description
<code>inputValues</code>	A vector of numeric input values

wls:changes

The `wls:changes` method takes a vector of input values of size n and produces a vector of (at most) $n-1$ differences between successive values. For example, if the input vector is { 3, 2, 5, 3, 7 }, the resulting vector is { 1, -1, 3, -2, 4 }.

Note the following:

- It is possible for a sequence to contain `Double.NaN`, which are skipped in subsequent computations.
- If an input value is non-numeric, an `IllegalArgumentException` is thrown.

Parameters

Name	Description
<code>inputValues</code>	A input vector of numeric values

wls:aliveServersCount

The `wls:aliveServersCount` function is a helper function that counts the number of Managed Server instances that are in the `RUNNING` state in a given cluster.

Parameters

Name	Description
<code>clusterName</code>	The name of the cluster containing the running server instances to be counted.

C

WLDF Query Language

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

- [Components of a Query Expression](#)
- [Supported Operators](#)
- [Operator Precedence](#)
The WLDF query language has six levels of precedence among its operators.
- [Numeric Relational Operations Supported on String Column Types](#)
- [Supported Numeric Constants and String Literals](#)
The WLDF query language has two sets of rules: one set for numeric constants, and another for string literals.
- [About Variables in Expressions](#)
- [Creating Policy Expressions](#)
- [Creating Data Accessor Queries](#)
- [Creating Log Filter Expressions](#)
- [Building Complex Expressions](#)

Components of a Query Expression

A query expression may include operators, literals, and variables. The supported variables differ for each type of expression.

- [Supported Operators](#)
- [Supported Numeric Constants and String Literals](#)
- [About Variables in Expressions](#)

The query language is case-sensitive.

Supported Operators

The WLDF query language supports a set of operators and, for each operator, corresponding operator and operand types. These operators, and corresponding types and operands, are listed and described in [Table C-1](#).

Table C-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 C-1 (Cont.) WLDF 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 1010 0001 = 1011 (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 A period (.) matches any single character</p>
MATCHES	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>

Operator Precedence

The WLDF query language has six levels of precedence among its operators.

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

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, the query evaluator attempts to convert the string value to appropriate numeric value before comparison:

```
STATUS = 100
```

```
STATUS != 100
```

```
STATUS < 100
```

```
STATUS <= 100
```

```
STATUS > 100
```

```
STATUS >= 100
```

When the string value cannot be converted to a numeric value, the query fails.

Supported Numeric Constants and String Literals

The WLDF query language has two sets of rules: one set for numeric constants, and another for string literals.

The 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.

The 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. See the documentation for the `java.lang.String.compareTo(String str)` method.

About Variables in Expressions

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

- [Creating Policy Expressions](#)
- [Creating Data Accessor Queries](#)
- [Creating Log Filter Expressions](#)

Note:

When specifying a wildcard pattern in a variable for a policy expression that matches custom MBean ObjectName instances, make sure the pattern is sufficiently explicit. If you exclude an MBean type name and use an ambiguous instance pattern, the following may result:

- Only WebLogic Server runtime MBean instances are matched to the pattern.
- The desired custom MBean instances are ignored.

For example, the following ObjectName pattern does not explicitly declare a type and uses an ambiguous ObjectName pattern that can match a WebLogic Server runtime MBean instance:

```
${ServerRuntime//com.b*:Type=Server*,*}
```

The preceding pattern matches the WebLogic Server runtime MBean instances, and causes any custom MBeans matching the same pattern to be ignored.

Creating Policy Expressions

You can create policies based on log events, instrumentation events, and harvested attributes.

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

- [Configuring Policies and Actions](#)
- [Configuring Policies](#)

The variables supported for creating the expressions are different for each type of policy, as described in the following sections:

- [Creating Log Event Policy Expressions](#)
- [Creating Instrumentation Event Policy Expressions](#)
- [Creating Harvester Policy Expressions](#)

Creating Log Event Policy Expressions

A *log event* policy 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 C-2](#):

Table C-2 Variable Names for Log Event Policy 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 Info, Notice, Warning, Error, Critical, Alert, and Emergency.	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 policy expression is:

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

Creating Instrumentation Event Policy Expressions

An *instrumentation event* policy 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 C-3](#):

Table C-3 Variable Names for Instrumentation Event Policy 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 policy expression is:

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

Creating Harvester Policy Expressions

A *Harvester* policy 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 monitored by the policy. It can be set to either Server Runtime or DomainRuntime. If omitted, it defaults to ServerRuntime.

If the namespace component is 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 policy in each Managed Server to monitor metrics related to that Managed Server instance.

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

The syntax for constructing a Harvester policy 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
```

Creating Data Accessor Queries

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

A Data Accessor query contains the following:

- The logical name of a data store, as described in [Data Store Logical Names](#).
- Optionally, the name(s) of one or more columns from which to retrieve data, as described in [Data Store Column Names](#).

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

- [Data Store Logical Names](#)
- [Data Store Column Names](#)

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 C-4](#).

Table C-4 Naming Conventions for Log Types

Log Type	Optional Identifiers	Example
ConnectorLog	The JNDI name of the connection factory	ConnectorLog/eis/ 900eisaBlackBoxXATxConnectorJNDINAME In this example, eis/ 900eisaBlackBoxXATxConnectorJNDINAME is the JNDI name of the connection factory specified in the weblogic-ra.xml deployment descriptor.
DataSourceLog	None	DataSourceLog
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
JMSSAFMessageLog	The name of the SAF agent.	JMSSAFMessageLog/MySAFAgent
ServerLog	None	ServerLog
WebAppLog	Web server name + Root servlet context name	WebAppLog/MyWebServer/ MyRootServletContext

Data Store Column Names

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

All column names from all WebLogic Server log types are listed in [Table C-5](#).



Note:

WebLogic Server Multitenant domain partitions and Resource Consumption Management are deprecated in WebLogic Server 12.2.1.4.0 and will be removed in the next release.

Table C-5 Column Names for Log Types

Log Type	Column Names
ConnectorLog	LINE, RECORDID
DataSourceLog	RECORDID, DATASOURCE, PROFILETYPE, TIMESTAMP, USER, PROFILEINFORMATION, SUPP_ATTRS, PARTITION_ID, PARTITION_NAME
DomainLog	CONTEXTID, DATE, MACHINE, MESSAGE, MSGID, RECORDID, SERVER, SEVERITY, SUBSYSTEM, THREAD, TIMESTAMP, TXID, USERID, SUPP_ATTRS, SEVERITY_VALUE, PARTITION_ID, PARTITION_NAME
EventsDataArchive	ARGUMENTS, CLASSNAME, CONTEXTID, CTXPAYLOAD, DOMAIN, DYES, FILENAME, LINENUM, METHODNAME, METHODDSC, MODULE, MONITOR, PAYLOAD, RECORDID, RETVAL, SCOPE, SERVER, THREADNAME, TIMESTAMP, TXID, TYPE, USERID, PARTITION_ID, PARTITION_NAME
HarvestedDataArchive	ATTRNAME, ATTRTYPE, ATTRVALUE, DOMAIN, NAME, RECORDID, SERVER, TIMESTAMP, TYPE, WLDFMODULE, PARTITION_ID, PARTITION_NAME
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
JMSSAFMessageLog	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 [Accessing Diagnostic Data With the Data Accessor](#).

Creating Log Filter Expressions

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

- 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](#)*.

Building Complex Expressions

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

- Nest queries by surrounding subexpressions 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 consecutive forward slashes (`//`) in the policy variable name.

D

WLDF Instrumentation Library

The WebLogic Diagnostics Framework (WLDF) instrumentation library contains diagnostic monitors and diagnostic actions.

For information about using items from the instrumentation library, see [Configuring Instrumentation](#).

- [Diagnostic Monitor Library](#)
- [Diagnostic Action Library](#)

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 DyelInjection 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 D-1](#).

Table D-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.
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.

Table D-1 (Cont.) Diagnostic Monitors for Use Within Server Scope

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
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 D-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 D-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: EntityBean.setEntityContext EntityBean.unsetEntityContext EntityBean.ejbRemove EntityBean.ejbActivate EntityBean.ejbPassivate EntityBean.ejbLoad EntityBean.ejbStore
EJB_Around_EntityEjbMethods	Around	Around	At exits of methods: EntityBean.setEntityContext EntityBean.unsetEntityContext EntityBean.ejbRemove EntityBean.ejbActivate EntityBean.ejbPassivate EntityBean.ejbLoad EntityBean.ejbStore
EJB_After_EntityEjbSemantic Methods	After	Stateless	At exits of methods: EntityBean.set* EntityBean.get* EntityBean.ejbFind* EntityBean.ejbHome* EntityBean.ejbSelect* EntityBean.ejbCreate* EntityBean.ejbPostCreate*
EJB_Around_EntityEjbSemanticMethods	Around	Around	At entry and exits of methods: EntityBean.set* EntityBean.get* EntityBean.ejbFind* EntityBean.ejbHome* EntityBean.ejbSelect* EntityBean.ejbCreate* EntityBean.ejbPostCreate*
EJB_After_SessionEjbMethods	After	Stateless	At exits of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate

Table D-2 (Cont.) Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
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.
EJB_Before_SessionEjbMethods	Before	Stateless	At entry of methods: SessionBean.setSessionContext SessionBean.ejbRemove SessionBean.ejbActivate SessionBean.ejbPassivate

Table D-2 (Cont.) Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
EJB_Before_SessionEjb SemanticMethods	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*
JDBC_Around_Execute	Around	Around	Before and after calls to methods: Statement.execute* PreparedStatement.execute*

Table D-2 (Cont.) Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
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 RowSet.setCommand
JDBC_After_Statement	After	Stateless	After calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JDBC_Around_Statement	Around	Around	Before and after calls to methods: Connection.prepareStatement Connection.prepareCall Statement.addBatch RowSet.setCommand
JMS_Before_AsyncMessage Received	Before	Stateless	At entry of methods: MessageListener.onMessage
JMS_After_AsyncMessage Received	After	Stateless	At exits of methods: MessageListener.onMessage
JMS_Around_AsyncMessage Received	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
JMS_Before_SyncMessage Received	Before	Stateless	Before calls to methods: MessageConsumer.receive*
JMS_After_SyncMessage Received	After	Stateless	After calls to methods: MessageConsumer.receive*

Table D-2 (Cont.) Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
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
MDB_Before_MessageReceived	Before	Stateless	At entry of methods: MessageDrivenBean.onMessage
MDB_After_MessageReceived	After	Stateless	At exits of methods: MessageDrivenBean.onMessage

Table D-2 (Cont.) Diagnostic Monitors for Use Within Application Scopes

Monitor Name	Monitor Type	Compatible Action Type	Pointcuts
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_SetMessageDrivenContext	Before	Stateless	At entry of methods: MessageDrivenBean.setMessageDrivenContext
MDB_After_SetMessageDrivenContext	After	Stateless	At exits of methods: MessageDrivenBean.setMessageDrivenContext
MDB_Around_SetMessageDrivenContext	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 D-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

Diagnostic Action Library

WLDF includes a library of diagnostic actions that you can use with delegating monitors. You can also use these diagnostic actions with custom monitors that you can define and use

within applications. Each diagnostic action can be used only with monitors with which they are compatible, as indicated by the Compatible Monitor Type column. Some actions (for example, `TraceElapsedTimeAction`) generate an event payload. The diagnostic action library includes the following actions:

- [TraceAction](#)
- [DisplayArgumentsAction](#)
- [TraceElapsedTimeAction](#)
- [TraceMemoryAllocationAction](#)
- [StackDumpAction](#)
- [ThreadDumpAction](#)
- [MethodInvocationStatisticsAction](#)
- [MemoryAllocationStatisticsAction](#)

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

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

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

TraceMemoryAllocationAction

`TraceMemoryAllocationAction` uses the HotSpot `ThreadMXBean` 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.

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

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 8 and later (Oracle HotSpot) supports this action.

This action generates an instrumentation event that is dispatched to the events archive. This action may be used only with HotSpot. 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

MethodInvocationStatisticsAction

MethodInvocationStatisticsAction is an Around action that is compatible with Around monitor types.

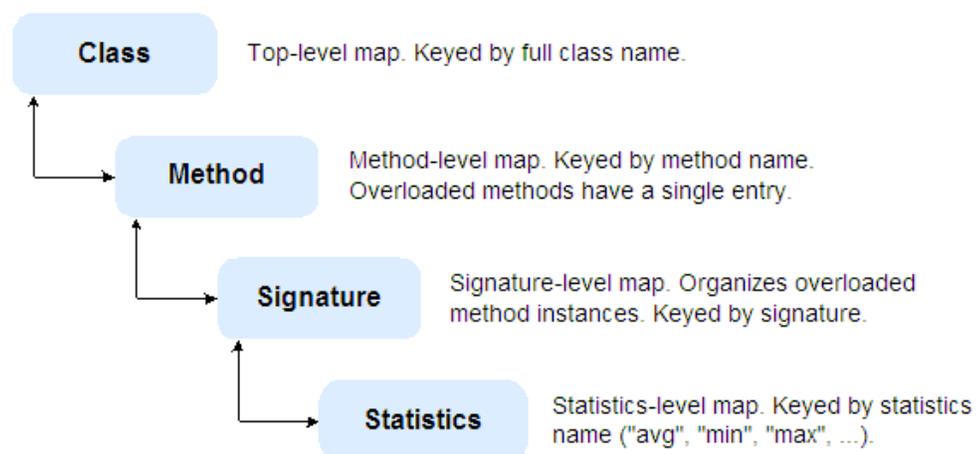
MethodInvocationStatisticsAction captures performance metrics around a joinpoint in memory without persisting an event in the Archive for each invocation. The statistics are collected and made available through the WLDInstrumentationRuntimeMBean. The collected statistics are also consumable by the Harvester and the Policies and Actions 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 WLDInstrumentationRuntimeMBean 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 D-1](#).

Figure D-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.

- [Instrumenting an Application with `MethodInvocationStatisticsAction` and Querying the Results](#)
- [Configuring the Harvester to Collect `MethodInvocationStatisticsAction` Data](#)
- [Configuring Policies Based on `MethodInvocationStatistics` Metrics](#)
- [Using JMX to Collect Data](#)

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 `WLDInstrumentationRuntimeMBean` 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. See *Navigating and Interrogating MBeans* in *Understanding the 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. See *Code Examples and Sample Applications* in *Understanding Oracle WebLogic Server*.

- [Configuring the Custom Monitor to Use `MethodInvocationStatisticsAction`](#)
- [Using WLST to Query Method Performance Statistics](#)

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 WebLogic Server Administration Console, as described in the following steps for instrumenting the MedRec sample application:

1. In the Domain Structure pane of the WebLogic Server 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_`

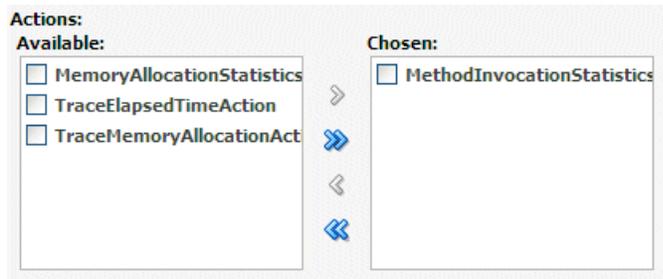
This pointcut expression encompasses a wide variety of public methods and classes in MedRec, but ignores all getter and setter methods, as well as code generated by WebLogic Server.

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 D-2](#):

Figure D-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 D-3](#):

Figure D-3 Selecting the MedRec Deployment



- 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. See [Using Deployment Plans to Dynamically Control Instrumentation Configuration](#).

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 [Sample Applications and Code Examples in *Understanding 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','password','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/
medrec')
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 D-4](#):

Figure D-4 Displaying All Data Values Collected by MethodInvocationStatisticsAction

```
wls:/medrec/serverRuntime/WLDFRuntime/WLDFRuntime/WLDFInstrumentationRuntimes/medrec>
print cmo.getMethodInvocationStatistics()
{com.bea.medrec.model.Patient$$Status={values={count=1, sum=4749.0, std_deviation=0.0,
avg=4749.0, sum_of_squares=2.2553001E7, max=4749, min=4749}}, com.bea.medrec.model.Record={
persistence_new={org.eclipse.persistence.internal.descriptors.Persiste
n
ceInternalDescriptor={count=1, sum=3353.0, std_deviation=0.0, avg=3353.0, sum_of_squares=1.1242609E7,
max=3353, min=3353}}, com.bea.medrec.repository.impl.RecordRepositoryImpl={findRec
ordsByPatientId={java.lang.Long={count=1, sum=5.7901772E7, std_deviation=0.0, avg=5.7901772E7,
sum_of_squares=3.352615200739984E15, max=57901772, min=57901772}}, com.bea.medrec.web.LocalizationDispatchingFilter={init={javax.servlet.FilterConfig={count=1,
sum=4191.0, std_deviation=0.0, avg=4191.0, sum_of_squares=1.7564481E7, max=4191, min=4191}}, com.bea.medrec.model.Administrator={persistence_new={org.eclipse.persisten
ce.internal.descriptors.Persiste
n
ceInternalDescriptor={count=1, sum=2794.0, std_deviation=0.0, avg=2794.0, sum_of_squares=7806436.0, max=2794, min=2794}}, com.bea.medrec.common.web.CacheControlPhaseListener={afterPhase={javax.faces.event.PhaseEvent={count=9, sum=60062.0,
std_deviation=2037.9947176417818, avg=6673.555555555556, sum_of_squares=4.38207896E8, max=9778, min=3911}}, beforePhase={javax.faces.event.PhaseEvent={count=9, sum=2.0024612E7,
std_deviation=4543091.564340776, avg=2224956.888888889, sum_of_squares=2.30311027074586E14, max=15026491, min=207847}}, com.bea.medrec.common.web.PageContextImpl={addGlobalOnlyMessage={java.lang.String={count=3, sum=4906209.0,
std_deviation=2293450.68553232, avg=1635403.0, sum_of_squares=2.3803377058133E13, max=4878832, min=13130}}, invalidateSession={count=1, sum=719365.0, std_deviation=0.0, avg=719365.0,
sum_of_squares=5.17486003225E11, max=719365, min=719365}}, com.bea.medrec.repository.impl.PatientRepositoryImpl={findByUsernameAndPasswordAndStatus={java.lang.String, java.lang.String, com.bea.medrec.model.Patient$$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 D-1](#):

Example D-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, "password")
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 D-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 |

```

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 [MethodInvocationStatisticsAction Examples](#).

- *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 D-3](#) provides examples that show to use `MethodInvocationStatisticsAction` to harvest various statistics.

Table D-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 ([]) following the type name. Space characters are insignificant for the Harvester.

Table D-3 (Cont.) MethodInvocationStatisticsAction Examples

The following MethodInvocationStatisticsAction instance configuration causes the following to be harvested
<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 <code>min</code> and <code>max</code> 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.

Configuring Policies Based on MethodInvocationStatistics Metrics

You can use the same syntax described in the previous sections to use `MethodInvocationStatistics` metrics in a policy expression. 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.

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.

MemoryAllocationStatisticsAction

The `MemoryAllocationStatisticsAction` uses the `HotSpot ThreadMXBean` 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 `MemoryAllocationStatisticsAction` is very similar to the existing `MethodInvocationStatisticsAction`. However, statistics tracked by

MemoryAllocationStatisticsAction are related to the memory allocated within a method call.

The MemoryAllocationStatisticsAction does not create an instrumentation event. When HotSpot is available, the statistics are available through the WLDInstrumentationRuntimeMBean.

The following statistics for each method are kept:

- count
- min
- max
- avg
- sum
- sum_of_squares
- std_deviation

E

Using Wildcards in Expressions

The WebLogic Diagnostics Framework (WLDF) supports the ability to use 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 policies, as well as specifying complex attributes in Harvester policies.

This appendix includes the following sections:

- [Using Wildcards in Harvester Instance Names](#)
- [Specifying Complex and Nested Harvester Attributes](#)
- [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 Policy and Actions configuration, the actual metrics are persisted in terms of each individually gathered metric.
- [Using Wildcards in Policy Instance Names](#)
Within Harvester policy expressions, you can use the asterisk (*) wildcard character to specify portions of an ObjectName. This gives you the ability to watch for multiple instances of a type.
- [Specifying Complex Attributes in Harvester Policies](#)

Using Wildcards in Harvester Instance Names

When specifying instance names within the <harvested-instance> element, you have some flexibility with regards to the property list order. Specifically, 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.
- [Examples](#)

Examples

The instance specification in [Example E-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 E-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 E-2](#) shows a JMX ObjectName pattern as the <harvested-instance> value:

Example E-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 E-3](#), some of the values in the ObjectName property list contain wildcard characters:

Example E-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 E-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 E-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>
```

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.

- [Examples](#)

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 E-5](#).

Example E-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 E-6](#) shows referencing all elements of URLPatterns using a wildcard character.

Example E-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 Foo.
<code><harvested-attribute>MyMap (Foo,Bar)</harvested-attribute></code>	Harvesting the value from the map with keys Foo and Bar.
<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 Foo and end with Bar.
<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 MyBean, which has a nested map type attribute MyMap. This example harvests this value from the map that has the key Foo.

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 Policy and Actions 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=MyMBean" ATTRNAME LIKE "mymap(%)a"
NAME="fooName=MyMBean" ATTRNAME MATCHES "mymap\((.*?)\)a"
```

Using Wildcards in Policy Instance Names

Within Harvester policy expressions, you can use the asterisk (*) wildcard character to specify portions of an `ObjectName`. This gives you the ability to watch for multiple instances of a type.

For example, to specify the `OpenSocketsCurrentCount` attribute for all instances of the `ServerRuntimeMBean` that begin with the name `managed`:

- The instance-name pattern can be a valid JMX `ObjectName` pattern, in which case the property list order is not important. For example:

```
${com.bea:Name=managed*,Type=ServerRuntime,*//OpenSocketCurrentCount}
```

This example is a valid JMX `ObjectName` pattern that can match:

- Any `ObjectName` that contains a `Name` key with a value that starts with `managed`
- A `Type` key that exactly matches the value `ServerRuntime`
- Any other property pairs

For more examples of valid JMX `ObjectName` patterns, see the `ObjectName` API documentation at <http://docs.oracle.com/javase/8/docs/api/javax/management/ObjectName.html>.

- If the name is a pattern but is not a JMX `ObjectName` pattern, WebLogic Server does pattern-matching using the pattern as-is. For example:

```
${com.bea:*Name=managed*,Type=ServerRuntime,*//OpenSocketCurrentCount}
```

This example is not a valid JMX `ObjectName` pattern. This pattern is matched using straight string substitution, where the pattern is matched as-is against the canonical form of the `ObjectName` for any target MBean instance.

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.

Specifying Complex Attributes in Harvester Policies

You can specify complex attributes (a collection, an array type or an Object with nested intrinsic attribute types) within Harvester policy 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
```

F

WebLogic Scripting Tool Examples

The WebLogic Diagnostics Framework (WLDF) includes examples that show using WLST and JMX to interact with WLDF components.



Note:

The following examples are also included with the WebLogic Server code examples:

- [Example: Configuring a Policy and a JMX Action](#)
- [Example: Writing a JMXWatchNotificationListener Class](#)
- [Example: Registering MBeans and Attributes For Harvesting](#)

These examples are bundled under the title "Configuring the Policies and Actions System and Harvesting Data Using WLST". For information about installing and configuring the WebLogic Server code examples, see *Sample Applications and Code Examples* in *Understanding Oracle WebLogic Server*.

For information about running WebLogic Scripting Tool (WLST) scripts, see *Running WLST from Ant* in *Understanding the WebLogic Scripting Tool*. For information about developing JMX applications, see *Understanding JMX* in *Developing Manageable Applications Using JMX for Oracle WebLogic Server*.

This appendix includes the following sections:

- [WLST Commands for Diagnostics](#)
- [Example: Dynamically Creating DyelInjection Monitors](#)
- [Example: Configuring a Policy and a JMX Action](#)
- [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](#)

WLST Commands for Diagnostics

WLST includes a set of commands that you can use to retrieve diagnostic data and manage diagnostic system resources. These commands are summarized in [Table F-1](#).

Table F-1 WLST Commands Used with WLDF

Command	Summary
<code>captureAndSaveDiagnosticImage</code>	Captures a diagnostics image and downloads it locally.
<code>createSystemResourceControl</code>	Creates a diagnostics system resource control using specified descriptor file that is not persisted in the domain configuration. See Using WLST to Activate and Deactivate Diagnostic System Modules .
<code>destroySystemResourceControl</code>	Destroys an external diagnostics system resource control; that is, one that is created in a server or cluster instance but that is not persisted in the domain configuration. See Using WLST to Activate and Deactivate Diagnostic System Modules .
<code>disableSystemResource</code>	Deactivates a diagnostic system resource control that is persisted in the domain configuration. See Using WLST to Activate and Deactivate Diagnostic System Modules .
<code>dumpDiagnosticData</code>	Dumps the diagnostics data from a Harvester to a local file.
<code>enableSystemResource</code>	Activates a diagnostic resource control. See Using WLST to Activate and Deactivate Diagnostic System Modules .
<code>exportDiagnosticData</code>	Execute a query against the specified log file.
<code>exportDiagnosticDataFromServer</code>	Executes a query on the server side and retrieves the exported WLDF data.
<code>getAvailableCapturedImages</code>	Returns a list of the previously captured diagnostic images.
<code>listSystemResourceControls</code>	Lists the diagnostic system modules that are currently configured in the domain. See Using WLST to Activate and Deactivate Diagnostic System Modules .
<code>mergeDiagnosticData</code>	Merges a set of data files that were previously generated by the <code>dumpDiagnosticData()</code> command.
<code>saveDiagnosticImageCaptureFile</code>	Downloads the specified diagnostic image capture.
<code>saveDiagnosticImageCaptureEntryFile</code>	Downloads a specific entry from the diagnostic image capture.

For complete details about each of these commands, including additional examples, see Diagnostics Commands in *WLST Command Reference for WebLogic Server*.

Example: Dynamically Creating DyelInjection Monitors

You can create a DyelInjection monitor dynamically using WLST. This demonstration script shown in [Example F-1](#) does the following:

- Connects to a server (boots the server first if necessary).
- Looks up or creates a WLDF System Resource.
- Creates the DyelInjection 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 F-1](#) only configures the dye monitor, which injects dye values into the diagnostic context. To fire events, you must implement downstream diagnostic monitors that use dye filtering to fire on the specified dye criteria. An example downstream monitor artifact is shown in [Example F-2](#). This must be placed in a file named `weblogic-diagnostics.xml` and placed into the `META-INF` directory of a application archive. It is also possible to create a monitor using a JSR-88 deployment plan. See *Deploying Applications to Oracle WebLogic Server*.

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

```
# Script name: demoDyeMonitorCreate.py
#####
# Demo script showing how to create a DyeInjectionMonitor dynamically
# via WLST. This script will:
# - Connect to a server, booting it first if necessary
# - Look up or create a WLDF System Resource
# - Create the DyeInjection Monitor (DIM)
# - Set the dye criteria
# - Enable the monitor
# - Save and activate
# - Enable the Diagnostic Context functionality via the
#   ServerDiagnosticConfig MBean
# Note: This will only configure the dye monitor, which will inject dye
# values into the Diagnostic Context. To fire events requires the
# existence of "downstream" monitors set to fire on the specified
# dye criteria.
#####
myDomainDirectory="domain"
url="t3://localhost:7001"
user="weblogic"
password="password"
myServerName="myserver"
myDomain="mydomain"
props="weblogic.GenerateDefaultConfig=true,weblogic.RootDirectory="\
+myDomainDirectory
try:
    connect(user,password,url)
except:
    startServer(adminServerName=myServerName,domainName=myDomain,
        username=user,password=password,systemProperties=props,
        domainDir=myDomainDirectory,block="true")
    connect(user,password,url)
# Start an edit session
edit()
startEdit()
cd ("/")
# Look up or create the WLDF System resource.
wldfResourceName = "mywldf"
myWldfVar = cmo.lookupSystemResource(wldfResourceName)
if myWldfVar==None:
    print "Unable to find named resource,\
        creating WLDF System Resource: " + wldfResourceName
    myWldfVar=cmo.createWLDFSystemResource(wldfResourceName)
# Target the System Resource to the demo server.
wldfServer=cmo.lookupServer(serverName)
myWldfVar.addTarget(wldfServer)
# create and set properties of the DyeInjection Monitor (DIM).
mywldfResource=myWldfVar.getWLDFResource()
mywldfInst=mywldfResource.getInstrumentation()
mywldfInst.setEnabled(1)
```

```

monitor=mywldfInst.createWLDfInstrumentationMonitor("DyeInjection")
monitor.setEnabled(1)
# Need to include newlines when setting properties
# on the DyeInjection monitor.
monitor.setProperties("\nUSER1=larry@celtics.com\
                    \nUSER2=brady@patriots.com\n")
monitor.setDyeFilteringEnabled(1)
# Enable the diagnostic context functionality via the
# ServerDiagnosticConfig.
cd("/Servers/"+serverName+"/ServerDiagnosticConfig/"+serverName)
cmo.setDiagnosticContextEnabled(1)
# save and disconnect
save()
activate()
disconnect()
exit()

```

Example F-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>

```

Example: Configuring a Policy and a JMX Action

You can use WLST to configure a policy and a JMX action using the WLDf Policies and Actions component. The demonstration script shown in [Example F-3](#) does the following:

- Connects to a server and boots the server first if necessary.
- Looks up/creates a diagnostic system module.
- Creates a policy expression on the ServerRuntimeMBean for the OpenSocketsCurrentCount attribute.

- Configures the actuaion to use a JMXNotification medium.



Note:

This example is also included with the WebLogic Server code examples. For information about installing and configuring these examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

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

- The `JMXWatchNotificationListener.java` class (see [Example: Writing a JMXWatchNotificationListener Class](#)).
- The `demoHarvester.py` script, which registers the `OpenSocketsCurrentCount` attribute with the Harvester for collection (see [Example: Registering MBeans and Attributes For Harvesting](#)).

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

1. To run the policy 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 executes the JMXNotification action for the policy configured in step 1.

Example F-3 Example: Policy and JMXNotification (demoWatch.py)

```
# Script name: demoWatch.py
#####
# Demo script showing how to configure a policy and a JMXNotification
# using the WLDF Policies and Action framework.
# The script will:
# - Connect to a server, booting it first if necessary
# - Look up or create a WLDF System Resource
# - Create a policy expression on the ServerRuntimeMBean for the
#   "OpenSocketsCurrentCount" attribute
# - Configure the policy 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 policy 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 fires the
# JMXNotification for the policy 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=password, 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.getObject ().getCanonicalName ()
watch.setRuleExpression ("${"+on+"} > 1")
watch.getRuleExpression ()
watch.setRuleExpression ("${"+on+"}/OpenSocketsCurrentCount} > 1")
watch.setAlarmResetPeriod (10000)
edit ()
save ()
activate ()
disconnect ()
exit ()

```

Example: Writing a JMXWatchNotificationListener Class

You can use the JMX API to write a JMXWatchNotificationListener. [Example F-4](#) shows an example.

Note:

This example is also included with the WebLogic Server code examples. For information about installing and configuring these examples, see [Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*](#).

Example F-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 policy
            action: " + notifName + "\n" + x);
        x.printStackTrace();
    }
    }
}
private void addNotificationHandler() throws Exception {
    /*
     * The JMX policy action listener registers with a Runtime MBean
     * that matches the name of the corresponding policy bean.
     * Each policy has its own Runtime MBean instance.
     */
    ObjectName oname =
        new ObjectName(
            "com.bea:ServerRuntime=" + serverName + ",Name=" +
            JMXWatchNotification.GLOBAL_JMX_NOTIFICATION_PRODUCER_NAME +
            ",Type=WLDWatchJMXNotificationRuntime," +
            "WLDWatchNotificationRuntime=WatchNotification," +
            "WLDRuntime=WLDRuntime"
        );
    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 = "password";

```

```

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()
}

```

Example: Registering MBeans and Attributes For Harvesting

You can use WLST to register MBeans and attributes for collection by the WLDF Harvester. The script shown in [Example F-5](#) does the following:

- 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.

Note:

This example is also included with the WebLogic Server code examples. For information about installing and configuring these examples, see [Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*](#).

Example F-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=password,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()

```

Example: Setting the WLDF Diagnostic Volume

You can use WLST to configure the volume of Java Flight Recorder data that is captured in a diagnostic image. By default, WLDF gathers 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`, WLDF, and potentially the JVM if flight recording is enabled, 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 F-6](#) shows changing the WLDF diagnostic volume to `Medium`:

Example F-6 Setting WLDF Diagnostic Volume

```
connect()
edit()
startEdit()
cd("Servers/myserver")
cd("ServerDiagnosticConfig")
cd("myserver")
cmo.setWLDFDiagnosticVolume("Medium")
save()
activate()
```

Example: Capturing a Diagnostic Image

You can use WLST to create a diagnostic image capture for a WebLogic Server instance. (Note that you can also create a diagnostic image capture using the WebLogic Server Administration Console or by executing an image action by means of the Policies and Actions 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 F-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 F-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, "password")
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)

```

Example: Retrieving a JFR File from a Diagnostic Image Capture

You can use WLST to retrieve the Java Flight Recorder (JFR) file from each diagnostic image capture that is located in the image destination directory on the server and copy it to a local directory. The script shown in [Example F-8](#) 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 F-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, "password")
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
```

G

WLDF Query Language-Based Policies

The WebLogic Diagnostics Framework (WLDF) provides the WLDF query language for creating policy expressions.



Note:

The WLDF query language is deprecated in WebLogic Server as of version 12.2.1. Oracle recommends using Java Expression Language (EL) instead. Diagnostic system modules containing policy expressions that use the WLDF query language are supported for backward compatibility. For information about using Java EL in policy expressions, see [Configuring Policies](#).

- [Types of Policies](#)
WLDF supports policies that you can configure within the context of using the WLDF query language.
- [Policy Configuration Options](#)
WLDF provides several tool options for configuring policies.
- [Configuring Harvester Policies Using the WLDF Query Language](#)
WLDF provides three main types of Harvester policies that can be configured with WLDF query language-based expressions. Each policy type is based on what the policy can monitor.
- [Configuring Log Policies Using the WLDF Query Language](#)
Use log policies to monitor the occurrence of specific messages or strings in the server or domain log. Policies of this type are triggered as a result of a log message containing the specified data being issued.
- [Configuring Instrumentation Policies Using the WLDF Query Language](#)
You use instrumentation policies to monitor the events from the WLDF Instrumentation component. Policies of this type are triggered as a result of the event being posted.

Types of Policies

WLDF supports policies that you can configure within the context of using the WLDF query language.

WLDF provides three main types of policies, based on what the policy can monitor:

- **Harvester** policies monitor the set of harvestable MBeans in the local runtime MBean server.
- **Log** policies monitor the set of messages generated into the server or domain logs.
- **Instrumentation** (or Event Data) policies 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 policy 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>
```

Policies 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 policies are triggered in real time, whereas Harvester policies are triggered only after the current harvest cycle completes.

Policy Configuration Options

WLDF provides several tool options for configuring policies.

For information about policy configuration options, see [How Policies Are Configured](#).

Configuring Harvester Policies Using the WLDF Query Language

WLDF provides three main types of Harvester policies that can be configured with WLDF query language-based expressions. Each policy type is based on what the policy can monitor.



Note:

If you define a policy to monitor an MBean (or MBean attributes) that the Harvester is not configured to harvest, the policy will work. The Harvester will implicitly harvest values to satisfy the requirements set in the defined policy expressions. However, data harvested in this way (that is, implicitly for a policy) is not archived. See [Configuring the Harvester for Metric Collection](#).

Harvester policies are triggered in response to a harvest cycle. So, for Harvester policies, the Harvester sample period defines a time interval between when a situation is identified and when it can be reported through an action. On average, the delay is $\text{SamplePeriod}/2$.

Example G-1 shows a configuration example of a Harvester policy that monitors several runtime MBeans. When the policy expression (defined in the `<rule-expression>` element) evaluates to true, six different actions are executed to generate the following: a JMX notification, an SMTP notification, an SNMP notification, an image action, and JMS notifications for both a topic and a queue.

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

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

Each variable is of the form:

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

In the preceding syntax, {entityName} is the JMX ObjectName as registered in the runtime MBean server or the type name as defined by the Harvester, and {attributeName} is the name of an attribute defined on that MBean type.



Note:

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

Example G-1 Sample Harvester Policy 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/2.0/
weblogic-diagnostics.xsd">
  <name>mywldf1</name>
  <harvester>
    <!-- Harvesting does not have to be configured and enabled for harvester
    policies. 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=WLDfHarvesterRuntime,ServerRuntime=myserver,Type=
WLDfHarvesterRuntime,WLDfRuntime=WLDfRuntime//TotalSamplingTime}
          &gt;= 100
          AND
          ${mydomain:Name=myserver,Type=
            ServerRuntime//OpenSocketsCurrentCount} &gt; 0)
          OR
          ${mydomain:Name=WLDfWatchNotificationRuntime,ServerRuntime=
            myserver,Type=WLDfWatchNotificationRuntime,
            WLDfRuntime=WLDfRuntime//Enabled} = true
          OR
          ${myCustomDomain:Name=myCustomMBean3//State} =
            'active')
      </rule-expression>
      <severity>Warning</severity>
      <alarm-type>AutomaticReset</alarm-type>
      <alarm-reset-period>10000</alarm-reset-period>
      <notification>myJMXNotif,myImageNotif,
        myJMSTopicNotif,myJMSQueueNotif,mySNMPNotif,
        mySMTPNotif</notification>
    </watch>
    <!-- Other policy-action configuration elements -->
  </watch-notification>
</wldf-resource>
```

This policy 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 policy, but will be passed on through the actions.

Configuring Log Policies Using the WLDF Query Language

Use log policies to monitor the occurrence of specific messages or strings in the server or domain log. Policies of this type are triggered as a result of a log message containing the specified data being issued.

The following example shows the configuration, in *DIAG_MODULE.xml*, for a server log policy:

```
<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/2.0/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>
```

In the preceding example, note how the `<rule-type>` of `Log` causes messages or strings entered in the server log to be monitored. A `<rule-type>` of `DomainLog` monitors messages or strings in the domain log.

Configuring Instrumentation Policies Using the WLDF Query Language

You use instrumentation policies to monitor the events from the WLDF Instrumentation component. Policies of this type are triggered as a result of the event being posted.

The following example shows the configuration, in *DIAG_MODULE.xml*, for an instrumentation policy:

```
<watch-notification>
  <watch>
    <name>myInstWatch</name>
```

```

<enabled>true</enabled>
<rule-type>EventData</rule-type>
<rule-expression>
  (PAYLOAD &gt; 100000000) AND (MONITOR = 'Servlet_Around_Service')
</rule-expression>
<alarm-type xsi:nil="true"></alarm-type>
<notification>mySMTPNotification</notification>
</watch>
<smtp-notification>
  <name>mySMTPNotification</name>
  <enabled>true</enabled>
  <mail-session-jndi-name>myMailSession</mail-session-jndi-name>
  <subject xsi:nil="true"></subject>
  <body xsi:nil="true"></body>
  <recipient>username@emailservice.com</recipient>
</smtp-notification>
</watch-notification>

```

Glossary

action

The operation that occurs as a result of the successful evaluation of a policy expression. The WebLogic Diagnostics Framework supports these types of diagnostic actions: Java Management Extensions (JMX), Java Message Service (JMS), Simple Mail Transfer Protocol (SMTP), Simple Network Management Protocol (SNMP), scaling, REST, script, and diagnostic image capture. See also [diagnostic image](#).



Note:

As of WebLogic Server 12.2.1, the term *notification* is replaced with *action*. The definition is unchanged.

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.

bean

See [WLDF bean](#).

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 policies that are to be evaluated; the actions, notifications, and alarms that are to be executed; 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.

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 *MBean Reference for Oracle WebLogic Server*.

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

harvested data

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

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.

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 runtime state. MBeans are registered with the MBean server that runs inside WebLogic Server. MBeans are implemented as standard MBeans which means that each class implements its own MBean interface.

MBean type discovery

For WebLogic Server entities, the set of harvestable types is known at system startup, but not the complete set of harvestable instances. 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 WebLogic Server 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.

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.

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](#).

smart rule

Out-of-the-box policy expression predicate that includes a number of configurable parameters. Smart rules greatly simplify the creation of scaling policies by providing

templates in which you specify the conditions for executing a scaling action on a dynamic cluster.

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.

policy

A policy encapsulates all of the information for a policy expression. This includes the expression, the alarm settings for the policy, and the various action handlers that are executed once a policy expression evaluates to true.



Note:

Note: As of WebLogic Server 12.2.1, the term *watch* is replaced with *policy*. The definition is unchanged.

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.

WLDF bean

An annotated HK2 service that can be discovered at run time and inserted into the Java EL objects that are used by policy expressions. Each bean provides a set of attributes and operations that can present a set of domain-specific data that can be used from policies.