Diagnostics

Version 9.51, Released November 2018

Java Agent Guide

Published November 2018

Legal Notices

Disclaimer

Certain versions of software and/or documents (“Material”) accessible here may contain branding from Hewlett-Packard Company (now HP Inc.) and Hewlett Packard Enterprise Company. As of September 1, 2017, the Material is now offered by Micro Focus, a separately owned and operated company. Any reference to the HP and Hewlett Packard Enterprise/HPE marks is historical in nature, and the HP and Hewlett Packard Enterprise/HPE marks are the property of their respective owners.

Warranty

The only warranties for products and services of Micro Focus and its affiliates and licensors (“Micro Focus”) are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Micro Focus shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Restricted Rights Legend

Contains Confidential Information. Except as specifically indicated otherwise, a valid license is required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

© Copyright 2005 - 2018 Micro Focus or one of its affiliates

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Java is a registered trademark of Oracle and/or its affiliates.

Oracle® is a registered trademark of Oracle and/or its affiliates.

Acknowledgements

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes software developed by the Spice Group (http://spice.codehaus.org).

For information about open source and third-party license agreements, see the *Open Source and Third-Party Software License Agreements* document.

Contents

Welcome to This Guide 8
How This Guide Is Organized 8
Diagnostics Documentation 8
Part 1: Introduction 10
Chapter 1: Diagnostics Java Agent Overview 11
About the Diagnostics Java Agent 11
Introducing the Diagnostics Profiler for Java 11
Features and Benefits of the Diagnostics Profiler for Java 11
Part 2: Installation and Configuration of the Java Agent 13
Chapter 2: Preparing to Install the Diagnostics Java Agent 14
Java Agent Installation Overview 15
System Requirements for the Diagnostics Java Agent 15
Chapter 3: Installing Java Agents 16
Pre-installation Checklist for the Java Agent 16
Installing and Configuring Java Agents 17
Silent Installation of the Java Agent 27
Setting File Permissions 28
Determining the Version of the Java Agent 28
Configuring for Firewalls, HTTPS, and Proxies 28
Uninstalling the Java Agent 28
Chapter 4: Preparing Application Servers for Monitoring with the Java Agent 30
About Preparing Application Servers for Monitoring 30
Examples for Configuring Application Servers 33
Example 1: Configuring GlassFish Application Server for Monitoring 34
Example 2: Configuring JBoss Application Server and JBoss EAP for Monitoring 36
Configuring a JBoss EAP Application 38
Example 3: Configuring Oracle Application Server for Monitoring 39
Using the Diagnostics JRE Instrumenter in Manual Mode 41
Example 4: Configuring SAP NetWeaver Application Server for Monitoring 42
Example 5: Configuring TIBCO ActiveMatrix BusinessWorks and Service Bus for Monitoring 44
Example 6: Configuring Tomcat Application Server for Monitoring 46
Example 7: Configuring WebLogic Application Server for Monitoring 48
Example 8: Configuring webMethods Server for Monitoring 49
Example 9: Configuring WebSphere Application Server for Monitoring 53
Example 10: Configuration for WebSphere Application Server Liberty 56
Verify the Application Server is Running the Java Agent 58
About the JRE Instrumenter and Different Options to Invoke 58

Other Configuration Options 65
Probe Registration Auto-Assigment 65
Configure Monitoring of Multiple Java Processes on an Application Server 66
Adjusting the Heap Size for the Java Agent in the Application Server 69
Configuring the SOAP Message Handler 69
Configuring the Discovery of a New J2EE Server for CI Population 71
Special Considerations for Applications Based on the OSGi Framework 72
Chapter 5: Configuring for Azul or Cloud Environments 74
Java Agents on Azul 74
Java Agents in Cloud Environments 75
Chapter 6: Preparing Application Servers for Client Monitoring with the Java Agent 78
About Client Monitoring 78
Enabling Client Monitoring 79
Configuring and Disabling Client Monitoring 80
Manually Instrumenting HTML/JSP Pages for Client Monitoring 81
Chapter 7: Upgrading the Diagnostics Java Agent 82
Upgrade Java Agents 82
Upgrade Notes and Limitations 84
Part 3: Advanced Java Agent Configuration and Instrumentation 85
Chapter 8: Monitoring Profiles 86
About Monitoring Profiles 87
Understanding Types of Diagnostics Deployments 87
The Predefined Monitoring Profiles 89
Custom Monitoring Profiles 89
Applying a Specific Monitoring Profile to a Probe 90
Overriding Settings in the Monitoring Profiles 91
Mapping Instrumentation Points to a Monitoring Profile 92
Mapping Metrics to a Monitoring Profile 92
Mapping Property Values to a Monitoring Profile 92
Chapter 9: Automatically Assigning a Probe to an Application 94
About Automatic Probe Assignment 94
Configuring a Probe to Automatically Assign Applications 94
Configuring an Agent to Automatically Assign Applications 94
General Configuration 95
Chapter 10: Custom Instrumentation for Java Applications 96
About Instrumentation and Capture Points Files 96
Using Regular Expressions in Points Files 97
Coding Points in the Capture Points File 98
Defining Points With Code Snippets 103
Controlling Class Map Capture 113
Instrumentation Examples 114

Understanding the Overhead of Custom Instrumentation 126
Instrumentation Control on a Per Layer Basis 126
Instrumented Location Throughput Throttling 127
Advanced Instrumentation Examples 128
Capturing HTTP Server Requests Based on Query Parameters 129
Configuring Cross VM Correlations for New or CustomTechnologies 137
Tutorial for Configuring Cross VM Correlation for Custom Technologies 140
Maintaining Instrumentation from the Java Profiler UI 147
Default Layers Defined for Typical Java Classes and Methods 156
Chapter 11: Advanced Java Agent and Application Server Configuration 158
Advanced Configuration Overview 158
About Dynamic Configuration 159
Disabling the Java Diagnostics Profiler 160
Controlling Probe Logging 160
Setting the Probe’s Host Machine Name 161
Specifying a Different Probe IP Address 161
Setting the Active Products Mode 162
Controlling Automatic Method Trimming on the Agent 163
Configuring URI and Parameter Capture 164
Capturing Non-Sequential Server Requests 167
Configuring an Agent for a Proxy Server 167
Time Synchronization for Probes Running on VMware 168
Limiting Exception Tree Data 168
Diagnostics Probe Administration Page 170
Authentication and Authorization for Diagnostics Java Profilers 172
Configuring Collection of CPU Time Metrics 174
Configuring Consumer IDs 175
A Value in the SOAP Body 179
Configuring SOAP Fault Payload Data 182
Configuring REST Services 183
Customizing Grouping JMS Temporary Queue/Topics 183
Configuring SQL Query Parsing 183
Capturing SQL Parameters 184
Configuring Display of Application Name for Server Requests 185
Maintaining Probe Settings from the Java Profiler UI 186
Generating Performance Reports for JUnit Tests 189
Chapter 12: Java Agent Metrics Collectors 191
About Metrics Capture 191
What Metrics are Being Collected by the Java Agent 192
Understanding Metric Collector Entries 192
About Collecting Additional Probe Metrics 194

Modifying Probe Metrics Already Being Captured 194
Stopping Capture of a Metric 194
Using Customized metrics.config Files for Multiple JVM Applications on a System 194
Chapter 13: Java Agent - System Metrics Capture 196
About System Metrics 196
System Metrics Captured by Default 196
Configuring the System Metrics Collector 197
Capturing Additional Custom System Metrics 198
Capturing Custom System Metrics on Windows Hosts 198
Capturing Custom System Metrics on Solaris Hosts 200
Capturing Custom System Metrics on Linux Hosts 200
Chapter 14: Java Agent - JMX Metrics Capture 203
About JMX Metrics 203
About Configuring JMX Metric Collectors 204
Additional Custom JMX Metrics 204
Getting a List of Available JMX or WebSphere PMI Metrics 204
Creating New JMX or WebSphere PMI Metrics Entries 206
Part 4: Using the Diagnostics Profiler for Java 210
Chapter 15: Diagnostics Profiler for Java 211
About the Java Diagnostics Profiler 211
How the Java Agent Provides Data for the Java Profiler 212
Java Diagnostics Profiler UI Navigation and Display Controls 213
Analyzing Performance Using the Call Profile Window 215
Thread Call Stack Trace Sampling 219
Comparison of Collection Leak Pinpointing and LWMD 221
Object Lifecycle Monitoring 222
Heap Walker Memory Analysis Execution Steps 224
Heap Walker Performance Characteristics 227
How to Access the Java Diagnostics Profiler 227
How to Enable LWMD for Collections Displays 228
How to Enable Allocation Capture 228
How to Enable Object Lifecycle Monitoring 229
How to Analyze Object Allocation 230
How to Enable Memory Analysis 230
Summary Tab Description 232
Hotspots Tab Description 234
Metrics Tab Description 236
Threads Tab Description 238
All Methods Tab Description 242
All SQL Tab Description 244
Collection Leaks Tab Description 245

Collections Tab Description 247
Exceptions Tab Description 250
Server Requests Tab Description 252
Web Services Tab Description 254
Allocation/LifeCycle Analysis Tab Description 256
Memory Analysis Tab Description 258
Configuration Tab Description 260
Send Documentation Feedback 263

Welcome to This Guide

Welcome to the Diagnostics Java Agent Guide. This guide describes how to install, configure and use the Diagnostics Java Agent and the Diagnostics Profiler for Java.

The Diagnostics Java Agent captures events such as method invocations, collection sites, and the beginning and end of business and server transactions.

The Diagnostics Java Agent works with other Software products such as LoadRunner, Application Performance Management, and Performance Center, and is an integrated part of Software's application lifecycle solution which includes load testing, production monitoring, and trouble diagnosis.

The Diagnostics Profiler for Java is installed as part of the Diagnostics Java Agent. The Diagnostics Profiler for Java provides a way for Java development teams to monitor the performance and diagnose issues with applications in the development environment. Software makes this tool available at no cost, through an easy-to-install trial software download.

How This Guide Is Organized

This guide contains the following parts:

- Part 1: "Introduction" on page 10

  Provides a high level overview of the features, components, architecture, and outputs of the Diagnostics Java Agent and the Diagnostics Profiler for Java.

- Part 2: "Installation and Configuration of the Java Agent" on page 13

  Describes how to install and configure the Diagnostics Java Agent.

- Part 3: "Advanced Java Agent Configuration and Instrumentation " on page 85

  Describes advanced configuration and instrumentation of the Java Agent and application server.

- Part 4: "Using the Diagnostics Profiler for Java" on page 210

  Describes the UI of the Diagnostics Java Profiler, and how to use it.

Diagnostics Documentation

Diagnostics includes the following documentation. Unless specified otherwise, the guides are in PDF format only and are available from the Software Support web site (https://softwaresupport.softwaregrp.com/) .

- **Diagnostics User Guide and Online Help:** Explains how to choose and interpret the Diagnostics views in the Diagnostics Enterprise UI to analyze your monitored applications. To access the online help for Diagnostics, choose **Help** > **Help** in the Diagnostics Enterprise UI. If Diagnostics is integrated with another Micro Focus Software product the online help is also available through that product's Help menu. The User Guide is a PDF version of the online help and their content is identical. The User Guide is available from the Diagnostics online help Home page, from the Windows Start menu (open **User Guide**), or from the Diagnostics Server installation directory.
- **Diagnostics Server Installation and Administration Guide:** Explains how to plan a Diagnostics deployment, and how to install and maintain a Diagnostics Server.

The following Agent guides contain content that supports agent installation, setup and configuration.

- **Diagnostics Java Agent Guide:** Describes how to install, configure, and use the Diagnostics Java Agent and the Diagnostics Profiler for Java.
- **Diagnostics .NET Agent Guide:** Describes how to install, configure, and use the Diagnostics .NET Agent and Diagnostics Profiler for .NET.

- **Diagnostics Collector Guide:** Explains how to install and configure a Diagnostics Collector.
- **Diagnostics System Requirements and Support Matrixes Guide:** Describes the system requirements for the various Diagnostics components.
- **Release Notes:** Provides last-minute new information and known issues about each version of Diagnostics. The PDF file is also located in the Diagnostics installation disk root directory.
- **Diagnostics Data Model and Query API:** Describes the Diagnostics data model and the query API you can use to access the data. The guide is also available from the Diagnostics online help Home page.
- **Diagnostics Frequently Asked Questions (FAQ):** Gives answers to frequently asked questions. The FAQ is also available from the Diagnostics online help Home page.

Part 1: Introduction

Chapter 1: Diagnostics Java Agent Overview

This chapter introduces the Diagnostics Java Agent and the Diagnostics Java Profiler by providing a high-level overview of features and components.

This chapter includes:

- "About the Diagnostics Java Agent" below
- "Introducing the Diagnostics Profiler for Java" below
- "Features and Benefits of the Diagnostics Profiler for Java" below

About the Diagnostics Java Agent

The Diagnostics Java Agent is installed on the machine that hosts the application that you want to monitor.

The agent captures events such as method invocations, collection sites, and the beginning and end of business and server transactions.

The Java Agent works with many of Software's Diagnostics products such as BSM/APM, LoadRunner, and Performance Center.

The Java Agent and the application environment must be configured to enable monitoring of your application. Instructions for configuring the Java Agent and the application environment can be found in:

- "Preparing Application Servers for Monitoring with the Java Agent" on page 30
- "Preparing Application Servers for Client Monitoring with the Java Agent" on page 78
- "Custom Instrumentation for Java Applications" on page 96
- "Advanced Java Agent and Application Server Configuration" on page 158

Introducing the Diagnostics Profiler for Java

The Diagnostics Java Profiler is installed as part of the Java Agent.

The Diagnostics Profiler for Java provides a way for JAVA and SAP development teams to monitor and diagnose issues with the performance of applications in the development environment. Software makes this tool available at no cost, through an easy-to-install trial software download.

The Diagnostics Profiler for Java provides a strong foundation for collaborative diagnostics because it has been built using the same Diagnostics probe technology that is used in Software's load testing and production monitoring products. When you use the Diagnostics Java Profiler in the development environment to profile applications and solve problems, you get a glimpse of the features that are included in the Diagnostics Lifecycle Solution that enable you to solve the toughest performance problems throughout the application's lifecycle.

Features and Benefits of the Diagnostics Profiler for Java

The following table describes some of the features and benefits of the Diagnostics Java Agent and the Diagnostics Profiler for Java:

Feature Description	Enables you to
Summary and Hotspots	Identify the top performance hotspots in your applications.
Server Request Breakdown	Identify where time is spent in an application.
Layer Breakdown	Identify the slowest J2EE layer.
Slowest Roots	Identify the slowest server request or application entry points for non-Web-fronted applications.
Top 3 Slowest Instances	Identify outliers to help diagnose intermittent problems.
VM Heap Usage	Identify memory problems and garbage collection issues.
Collection Memory Leak Diagnostics	Identify the fastest growing and largest size JAVA collections, including the caller, and the exact line number where collection was allocated.
Heap Breakdown including Class and Size Information	Identify leaking objects, object growth trends, object instance counts, and the byte size for objects.
SQL Diagnostics (Slowest SQL)	Identify the slowest SQL query and report query information.
Synchronization Diagnostics	Identify locks including hold times.
Exception Diagnostics (including exception traces and counts)	Identify exception counts and trace information (which often go undetected)
Layered view of Portal Transaction data	Identify the layer in the J2EE stack that consumes the most time for Portal transactions, along with the business context for the transaction, so that end-user impact can be assessed. The monitored layers include iVews, portal server requests, WebDynPro and JSP DynPro applications.
Transaction breakdown of portal server requests and methods	Identify the worst performing server requests or methods, and the applications and services that are being impacted
Cross Tier Transaction Breakdown	Detect problems originating from NetWeaver or ABAP platforms.

Part 2: Installation and Configuration of the Java Agent

Chapter 2: Preparing to Install the Diagnostics Java Agent

This chapter presents the information that you need as you prepare for the installation and configuration of the Diagnostics Java Agent.

Note: The procedures in this chapter do not apply when installing the Java Agent in an AppPulse environment. For information about AppPulse agent installation, see the *Java Agent Quick Start* guides. These guides are available on the Diagnostics Agent Download and Setup page in AppPulse.

This chapter includes:

- "Java Agent Installation Overview" on the next page
- "System Requirements for the Diagnostics Java Agent" on the next page

Java Agent Installation Overview

The following is an overview of the steps involved in installing and configuring the Java Agent. Understanding this workflow will help you plan your Java Agent installation.

Agents can optionally be auto-deployed. In that case some steps are performed automatically for you as described below.

1. **Prepare the host where the Java Agent is to be installed.**

   The host must contain the application server installation for the application to be monitored.The host also must meet the system requirements listed in the next section.

2. **Obtain the Java Agent installation package and install (unpack) the Java Agent.**

3. **Run the Java Agent Setup program.**

   When running the setup, you can choose to auto-deploy an agent.

   For more information, see "Installing and Configuring Java Agents" on page 17.

4. **Instrument the JRE used by the application server.**

   Diagnostics' JRE instrumentation does not modify the installed JRE, but rather places copies of instrumented classes under the Java Agent installation directory. Then with the proper JVM parameters these instrumented classes will be loaded into the JVM that runs the application server.

   If you chose to auto-deploy an agent, this step is performed automatically.

   This procedure varies for each type of application server. For more information, see "Preparing Application Servers for Monitoring with the Java Agent" on page 30.

5. **Configure the application server startup script.**

   Configure your application server JVM parameters to invoke the agent and use the instrumented JRE when the application starts.

   If you chose to auto-deploy an agent, this step is performed automatically.

   This procedure varies for each type of application server. For more information, see "Preparing Application Servers for Monitoring with the Java Agent" on page 30.

6. **Restart the application server to pick up the changes to the startup script.**

7. **Validate the agent installation and configuration.**

   For more information, see "Verify the Application Server is Running the Java Agent" on page 58.

System Requirements for the Diagnostics Java Agent

For details on the system configurations that are recommended for hosting the Diagnostics Java Agent, refer to the relevant version of the **Diagnostics System Requirements and Support Matrices Guide** on the Software Support site (https://softwaresupport.softwaregrp.com/group/softwaresupport/).

Chapter 3: Installing Java Agents

This chapter describes how to install a Java Agent and give you information about the setup and configuration of the Java Agent

Note: The procedures in this chapter do not apply when installing the Java Agent in an AppPulse environment. For information about AppPulse agent installation, see the *Java Agent Quick Start* guides on the Diagnostics Agent Download and Setup page in AppPulse.

This chapter includes:

- "Pre-installation Checklist for the Java Agent" below
- "Installing and Configuring Java Agents" on the next page
- "Silent Installation of the Java Agent" on page 27
- "Setting File Permissions" on page 28
- "Determining the Version of the Java Agent" on page 28
- "Configuring for Firewalls, HTTPS, and Proxies" on page 28
- "Uninstalling the Java Agent" on page 28

Pre-installation Checklist for the Java Agent

The following list is provided to help you gather the information that you will need during the installation of the Java Agent.

- Determine which mode the agent needs to operate in–it can only operate in one mode at a time. The deployment scenario of your Diagnostics installation determines the mode that you specify. The mode affects the licensing impact of the agent as well as the default configuration of the agent. The modes are as follows:
  - **Diagnostics Profiler mode**. Provides access to raw metric data on the agent host directly, without it being processed. The agent instance does not connect to a Diagnostics Server.
  - **Diagnostics Mode for LoadRunner/Performance Center**. The agent is used with a Diagnostics Server in a load testing (or pre-production) environment where probes are used only in LoadRunner or Performance Center runs.
  - **Enterprise Mode**. Agent sends collected metrics to an on-premise Diagnostics Server and/or an Software-as-a-Service (SaaS) Diagnostics Server.

  You can rerun the Agent setup to change the mode of an existing agent installation.
- For all modes, the agent must be installed on the machine hosting the application that you want to monitor. The Agent cannot monitor an application remotely.
- For all modes, determine the location of the application server startup script.
- For all modes, make sure the host meets the recommended system requirements. For details, refer to the relevant version of the **Diagnostics System Requirements and Support Matrices Guide** on the Software Support site (https://softwaresupport.softwaregrp.com/group/softwaresupport/).

- For agents installed in Enterprise Mode, you need the server connection details. For Diagnostics Servers, this is the fully-qualified host name (FQDN) or IP address of the host of the mediator server to which the probe sends the collected data. Your deployment may require that multiple probes send data to the same mediator. Your deployment may have no mediator servers in which case the collected data is sent to the commander server. If the server is configured to use a port other than the default port, you need the port number.

  You can obtain the server host FQDN and port from the Diagnostics System Administrator.

  For Software-as-a-Service (SaaS)-hosted servers, obtain the server connection details from your SaaS administrator.
- For agents installed in Enterprise Mode or Diagnostics Mode for Load Runner/Performance Center, you need an agent naming strategy. Each agent instance in the deployment environment is represented in the same, shared Diagnostics Enterprise UI. Agent names must be unique and clear so that users can distinguish between the different applications and types of probes among all in the deployment environment.
- For agents installed in Enterprise Mode or Diagnostics Mode for Load Runner/Performance Center, determine which agents belong in which agent groups. Probe groups are optional, logical groupings of probes.
- For all modes, if there is a pre-existing installation of the Java Agent on the host machine and you want to retain its configuration, follow the procedure in "Upgrading the Diagnostics Java Agent" on page 82.

Installing and Configuring Java Agents

The installation and configuration of the Java Agent includes the following steps:

"Step 1: Obtain the Installation Package" below

"Step 2: Start the Agent Setup" on the next page

"Step 3: Specify the Agent Mode" on page 19

"Step 4: Specify Agent Name, Group, and Auto-deployment" on page 20

"Step 5: Specify Diagnostics Server Information" on page 23

"Step 6: Specify RUM Integration Settings" on page 24

"Step 7: Review Post Setup Summary" on page 26

"Step 8: Verify Connectivity from the Agent to the Diagnostics Server" on page 27

Step 1: Obtain the Installation Package

1. Copy the Java Agent installation package to the target host. You typically obtain the package from one of the following locations:
   - The Diagnostics release media.
   - The Software Support site.
   - The Downloads page in BSM/APM; select **Admin > Platform Administration > Setup and Maintenance > Category > Diagnostics.**

The package name indicates the platform on which it can be run:

On this platform:	Use this package:
Windows	DiagJavaAgent_<release number>.zip
All other platforms, including AIX, Linux, or Solaris	DiagJavaAgent_<release number>.zip or DiagJavaAgent_<release number>.tgz

2. Extract all contents of the installation package to a directory on the target host.

   If you are extracting the **.tgz** package for Linux/Unix systems, use the following command to extract the files with their permissions: `tar -pxvzf DiagJavaAgent_<release number>.tgz`.

   Note: On AIX systems, you must use the GNU version of tar, or install from the zip version of the package.

   Caution: Do not extract the zip contents to a temp directory.

   Within the extracted files you see the **JavaAgent/DiagnosticsAgent/** directory. This location is hereafter referred to as <agent_install_directory>.

Step 2: Start the Agent Setup

Running the Agent Setup does not require root or administrative privileges.

If you plan to auto-deploy the agent, the user running the Agent Setup must have permission to modify the application server startup script and permission to write files in the application server **bin** directory.

On AIX, Linux, or Solaris, the user that installs the Java Agent ideally is the same user that installed the application server. The reason is that write access to the <agent_install_directory>/log directory is required by application server. See "Setting File Permissions" on page 28.

Run the setup command appropriate for your platform. You can run the Agent Setup in graphical or console mode.

Graphical mode on Windows:

```
<agent_install_directory>\setup.cmd
```

Graphical mode on AIX, Linux, or Solaris:

```
export DISPLAY=<hostname>:0.0
<agent_install_directory>/setup.sh
```

The "xhost +" command must have been executed on the host where the installation is to be displayed (the <hostname> used in the export command).

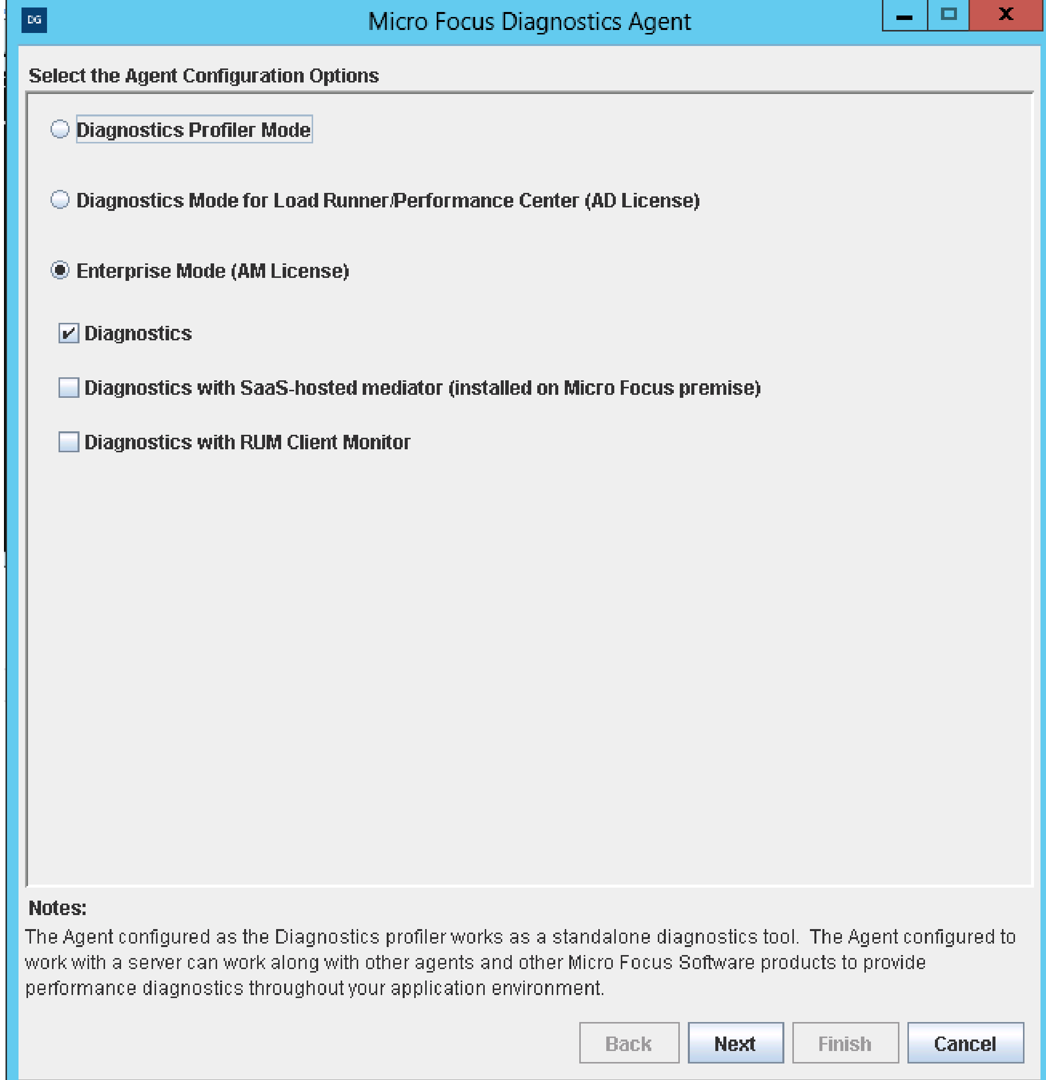
Console mode on Windows:

```
<agent_install_directory>\setup.cmd -console
```

Console mode on AIX, Linux, or Solaris:

```
<agent_install_directory>/setup.sh -console
```

Step 3: Specify the Agent Mode



Select the mode appropriate for the agent:

- **Diagnostics Profiler Mode:** Configure the agent as a Diagnostics Java Profiler. The Diagnostics Java Profiler does not connect to a Diagnostics server and is accessed through its own user interface. Diagnostics Profiler mode is typically used when installing the Diagnostics Java Profiler trial software prior to purchasing the Diagnostics product.

  When you select Diagnostics Profiler Mode there are no other configuration options. Select **Finish** to complete the configuration and skip to "Step 7: Review Post Setup Summary" on page 26.

- **Diagnostics Mode for LoadRunner/Performance Center (AD License):** Configure the agent for use with a Diagnostics Server in a load testing (or pre-production) environment where probes are used only in LoadRunner or Performance Center runs.

The agent will be configured in AD license mode which means the agent will only be counted against your Diagnostics AD license capacity when the agent is in a LoadRunner or Performance Center testing run. See "Licensing Diagnostics" in Diagnostics Server Installation and Administration Guide for more information on AD license capacity.

- **Enterprise Mode (AM License):** Configure the agent to send collected data to one of the following:
  - **Diagnostics**. The agent will connect to a Diagnostics Server that is installed locally, in your deployment environment.
  - **Diagnostics with SaaS-hosted mediator.** The agent will connect to a Diagnostics Server that is hosted on an SaaS system on-premise at Micro Focus.
  - **Diagnostics with RUM Client Monitor.** The agent will connect to a Diagnostics Server according to the selected mode (**Diagnostics** or **Diagnostics with SaaS-hosted mediator**) and enables the integration between Diagnostics and Real User Monitor (RUM). For details on the integration, refer to the RUM Client Monitor-Diagnostics Integration Guide located on the Software Support web site (https://softwaresupport.softwaregrp.com/).

    Note: This option is only available when installing the Java Agent on Windows, using setup.cmd in the graphical mode.

  For those agents with Enterprise mode set, the agent will be counted against your Diagnostics AM license capacity.

In AD mode the agent will ONLY capture data during a LoadRunner or Performance Center run and the results will be stored in a specific Diagnostics database for that run, for example, Default Client:21. When the agent is in AD mode it will NOT send any data to the server unless the probe is part of a LoadRunner/Performance Center run.

The advantage of running a probe in AD mode is that probes in AD mode are only counted against license capacity if they are in a LoadRunner or Performance Center test run. For example if 20 probes are installed in LoadRunner/Performance Center AD mode but only have 5 are in a run at any one time then you would only need an AD license capacity of 5 probes.

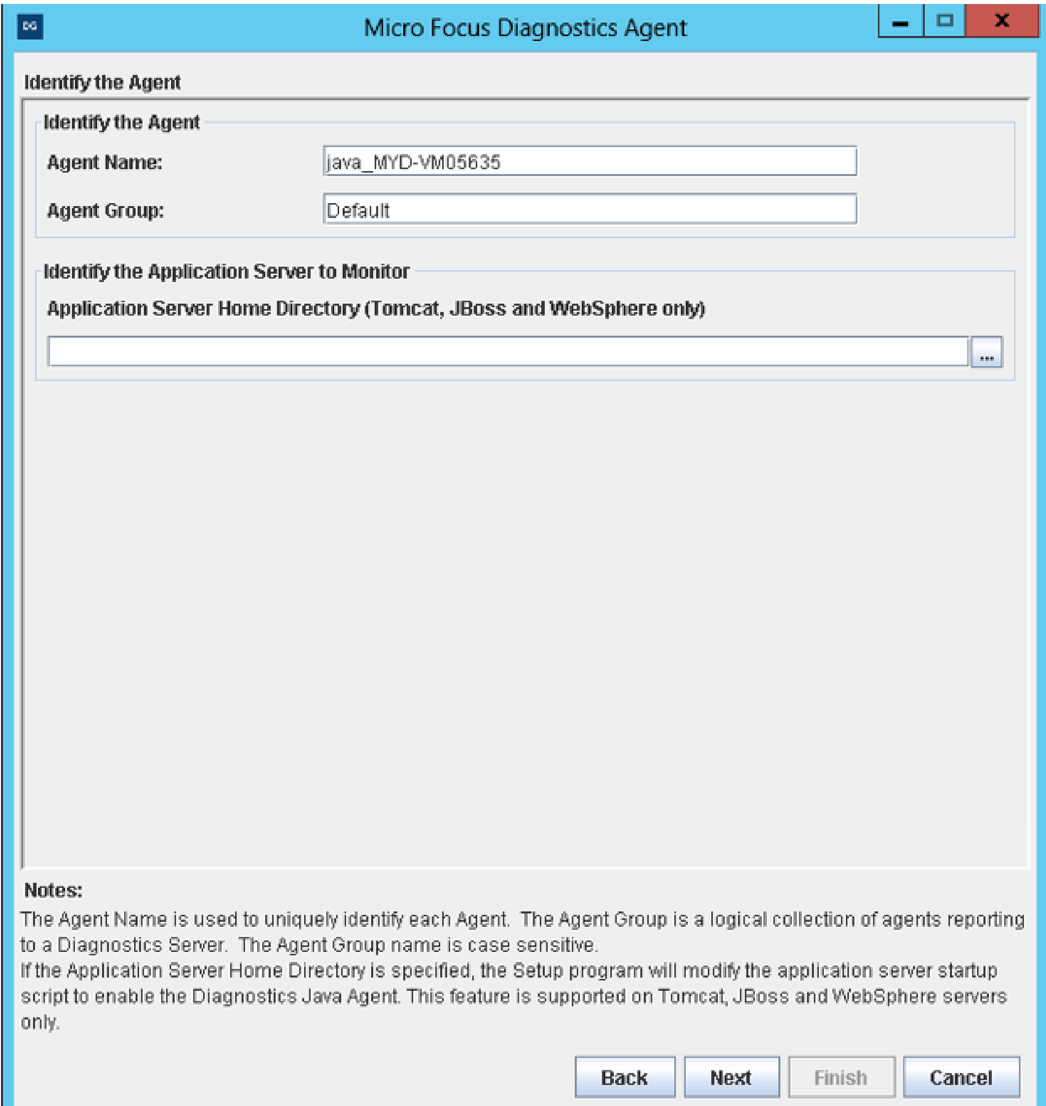
In console mode enter an X to select the mode for installation.

Click **Next** (in console mode **Enter**) to continue to the next step.

Step 4: Specify Agent Name, Group, and Auto-deployment

This step is skipped if the agent configuration specified in the previous step is **Diagnostics Profiler Mode**.

Assign a name to the Java Agent and specify the group to which it belongs. For agents that will monitor Tomcat, JBoss, or WebSphere application servers, you can optionally choose to auto-deploy the agent.



- **Agent Nam:e** Enter a name that uniquely identifies the agent within the Diagnostics Enterprise User Interface. You can use - , _ and all alphanumeric characters in the name. The agent name is assigned as the default probe entity name. When assigning a name to an agent, choose a name that will help you recognize the application being monitored and the system the agent is installed on (for example if installing on the system ovrserver130 with a WebLogic application server you could use the agent name WL10_MedRec_ovrserver130).

  Diagnostics does not support localization of agent names.

  If you have a single agent installed on a system and plan to monitor multiple application servers you specify unique probe names and parameters in the application server startup script. See "Configure Monitoring of Multiple Java Processes on an Application Server" on page 66.

- **Agent Group:** Enter a name for an existing group or a new group to be created. The agent group name is case-sensitive. The agent group name is used as the probe group name.

  Probe groups are logical groupings of probes. The performance metrics for a probe group are aggregated and can be displayed on many of the Diagnostics views. For example, you can assign all of the probes for a particular enterprise application to a probe group so that you can monitor both the performance at the group level and the performance based on individual probe entities.

- **Application Server Home Directory:** Enter or browse to select the home directory for the Tomcat, JBoss, or WebSphere application server to be monitored. For example, C:\JBossAll\jboss-as-web-7.0.2.Final for JBoss, or <C:\Program Files\IBM\WebSphere\AppServer> for WebSphere.

  Note:

  - For application servers that are not Tomcat, JBoss, or WebSphere, leave this field empty and refer to "Examples for Configuring Application Servers " on page 33.
  - You can auto-deploy the agent for Tomcat application servers that have a startup script (that is, applications that run as a process) as well as for Tomcat applications that run as a Windows service. For Tomcat applications that run as a Windows service, note:
    - The startup script is also changed.
    - Only those services whose catalina.home property points to the location of the relevant startup script are changed.
    - Auto-deploying Tomcat as a Windows service causes the JRE Instrumenter to run in Automatic Explicit Mode. For details, see "Using the JRE Instrumenter in Automatic Explicit Mode" on page 60.
    - For details on how to manually configure a Tomcat application as a Windows service, refer to "To configure a Tomcat server without a startup script" in "Example 6: Configuring Tomcat Application Server for Monitoring" on page 46.

  The Setup program modifies the startup script (for Tomcat and JBoss), or the xml file (for WebSphere), for the application server so that the application server runs enabled for monitoring by the Java agent the next time it is started. The original, initial version of the modified file is saved as a backup in the same location. The file is named as follows: Agentbackup_year_month_day_originalFileName. For example, **Agentbackup_2018_05_15_domain.bat**.

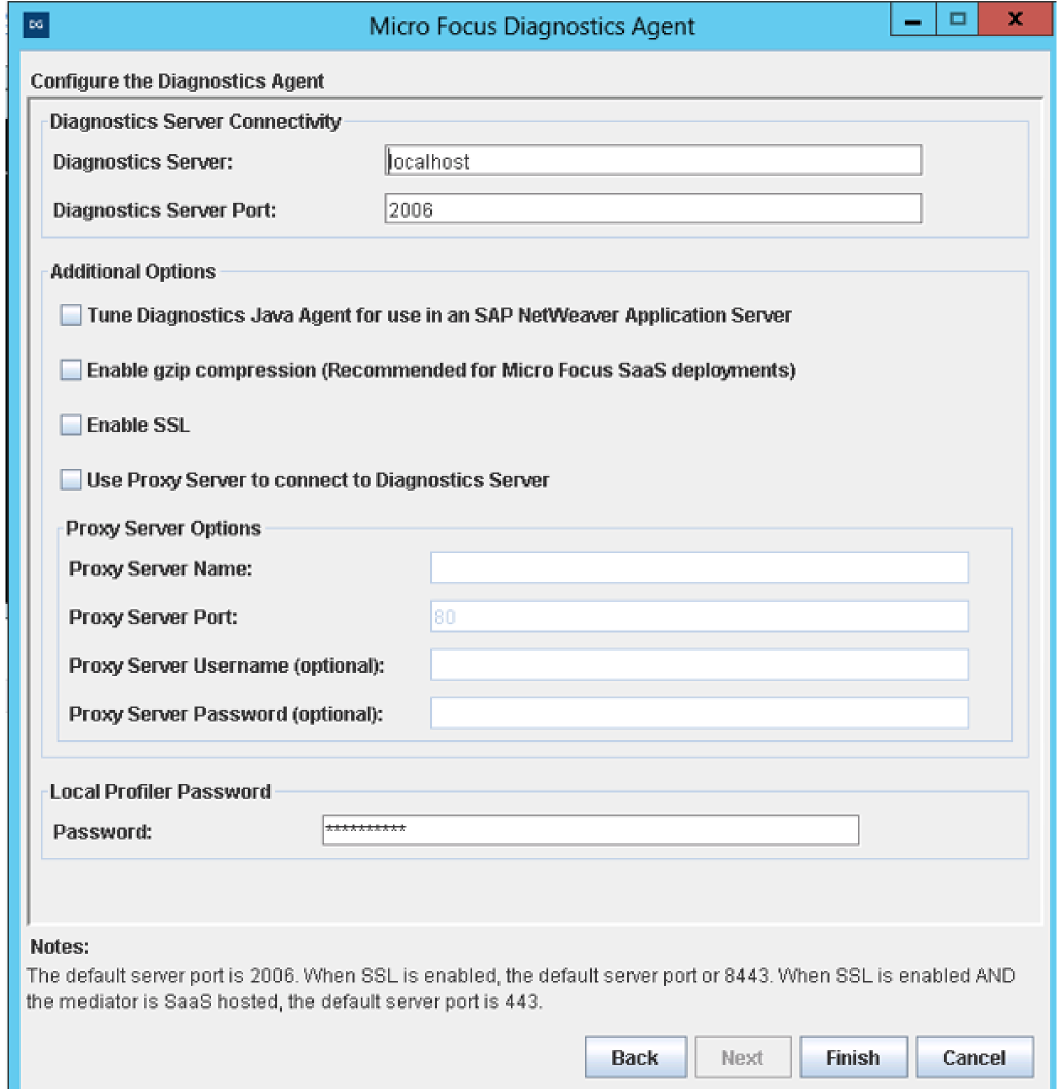
File Modified by the Setup	Backup File
For Tomcat: <TOMCAT_HOME>/bin/catalina.[bat|sh]	<TOMCAT_HOME>/bin//Agentbackup_<date>_catalinacatalina.[bat|sh]
For JBoss Version 7.x, Wildfly 8: <JBOSS_HOME>/bin/domain.[bat|sh]	<JBOSS_HOME>\bin\Agentbackup_<date>_domain.[bat|sh]
For JBoss Version 7.x, Wildfly 8: <JBOSS_HOME>/bin/standalone.[bat|sh]	<JBOSS_HOME>\bin\Agentbackup_<date>_standalone.[bat|sh]
For JBoss Version 6.x: <JBOSS_HOME>/bin/run.[bat|sh]	<JBOSS_HOME>/bin/Agentbackup_<date>_run.bat
For WebSphere: <WAS_HOME>/profiles/<profile_name>/config/cells/<cell_name>/nodes/<node_name>/servers/<server_name>/server.xml	<WAS_HOME>/profiles/<profile_name>/config/cells/<cell_name>/nodes/<node_name>/servers/<server_name>/Agentbackup_<date>_server.xml

  The Post Setup Summary dialog indicates whether the startup script has been modified successfully.

Select **Next** (in console mode **Enter**) to continue with the next step.

Step 5: Specify Diagnostics Server Information

Enter the configuration information for the Diagnostics Server and additional options.



- **Diagnostics Server:** Enter the host name or IP address of the host of the Diagnostics Server to which this agent will connect. Specify the **fully qualified host name** rather than just the simple host name. In a mixed OS environment, where UNIX is one of the systems, this is essential for proper network routing.

  Typically this is the Diagnostics mediator server. In environments with no Diagnostics mediator servers, specify the Diagnostics Commander Server details here.

  If this agent is being deployed for **Software-as-a-Service (SaaS)** then an Micro Focus SaaS administrator will provide you with the information on the host name and port to use. Also note that for an Micro Focus SaaS environment the Enable gzip option will be checked automatically for you and you will not see the Enable SSL option because it is configured on the Diagnostics Commander/Mediator on premises.
- **Diagnostics Server Port:** Enter the port number of the Diagnostics Server.

The default port for the Diagnostics Server is **2006**. For SSL communications with the server the port is typically set to **8443** for a locally installed server. If the port was changed since the Diagnostics Server was installed, specify the new port number here instead of the default.

The default port if you are installing the agent for a SaaS environment is **443** (the SaaS administrator will provide you with details).

- **Tune Diagnostics Java Agent for use in an SAP NetWeaver Application Server:** Set to allow this agent to support a SAP NetWeaver Application Server.
- **Enable gzip compression:** Set to compress the data between the Java Agent and the mediator. This is a tradeoff between bandwidth and probe performance overhead.

  If you are using **Software-as-a-Service (SaaS)** you typically enable gzip compression. See your SaaS administrator for more information.
- **Enable SSL:** Check to instruct the agent to connect to the Diagnostics Server in SSL mode and to attempt to download the required certificate chain from the server. As a result the **server.properties** trusted certificate will then include the certificate. For more information on secure communications see "Enabling HTTPS Between Components" in the Diagnostics Server Installation and Administration Guide.

  If you are using Software-as-a-Service (SaaS) this option is required.
- **Use Proxy Server to connect to Diagnostics Server:** Set if a proxy server is used to communicate with the Diagnostics Mediator Server. Enter the appropriate options.

  If you are using Software-as-a-Service (SaaS), specify this option if your company requires a proxy to communicate to outside servers.

  Proxy Server Options:
  - **Proxy Server Name:** Host name of the proxy server.
  - **Proxy Server Port:** Port of the proxy server.
  - **Proxy Server Username (optional):** The user used to authenticate the proxy server.
  - **Proxy Server Password (optional):** The password used to authenticate the proxy server.

  These options can be set or modified after the setup is run by modifying the **dispatcher.properties** file on the agent system. For more information on proxy configuration see "Configuring Diagnostics Servers and Agents for HTTP Proxy" in the Diagnostics Server Installation and Administration Guide.
- **Local Profiler Password:** This password is used to authenticate logins (username: admin) to the local Diagnostics Profiler, which is installed along with the agent. Enter a password and provide it to the users that will run the Diagnostics Profiler.

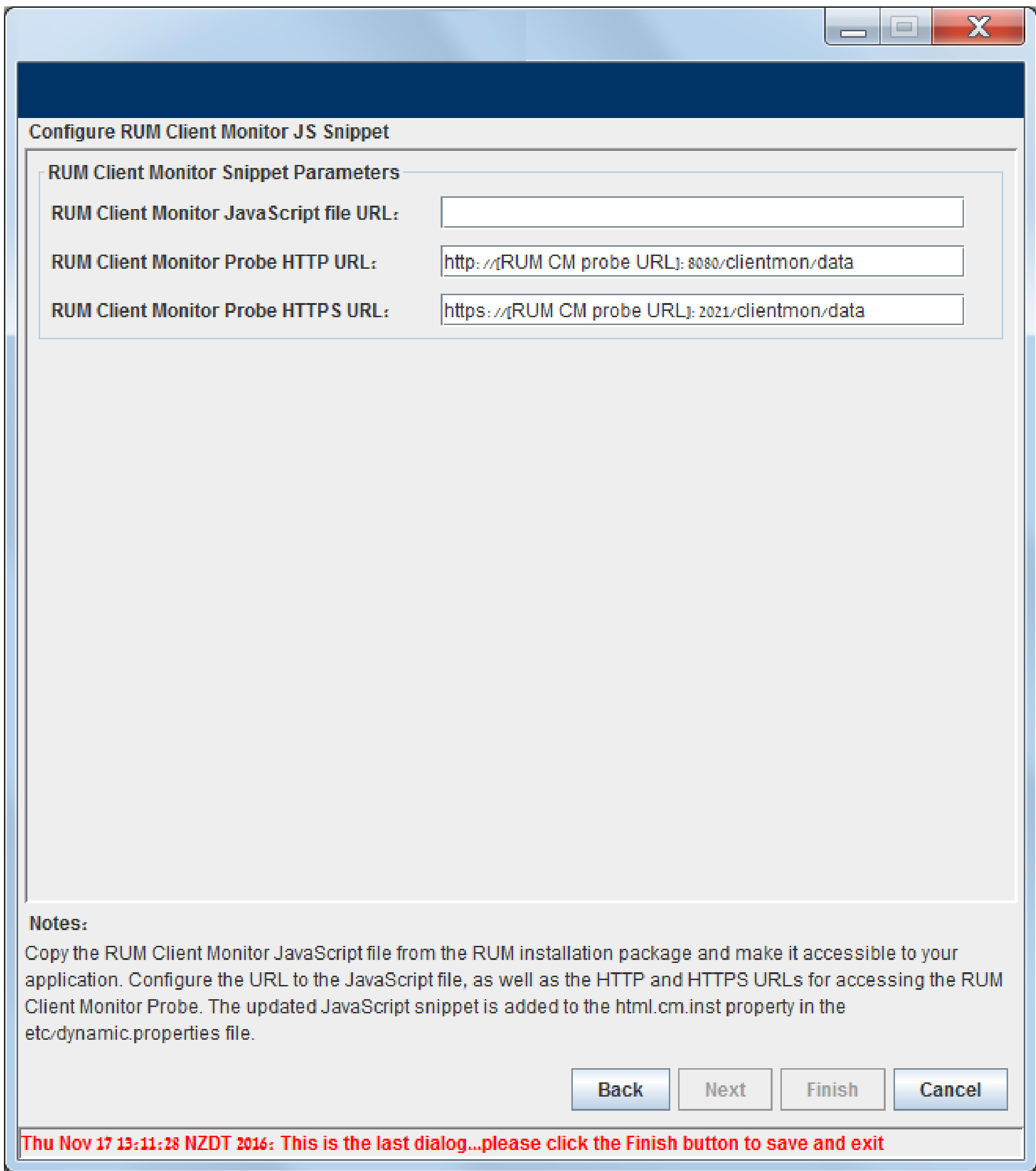
In console mode interface for each option enter an **X** for Yes and **O** for No.

Select **Next** (in console mode **Enter**) to continue with the next step.

Step 6: Specify RUM Integration Settings

This step is skipped if the **Diagnostics with RUM Client Monitor** check box is not selected in "Step 3: Specify the Agent Mode" on page 19

Enter the configuration information for the RUM Client Monitor (Browser Probe) JavaScript snippet.



- **RUM Client Monitor JavaScript file URL:** Enter the full URL path to the source file containing the RUM Client Monitor JavaScript. The default file name is **clientmon.js**.

  > **Note:** Copy the RUM JavaScript (clientmon.js) from the RUM installation package. Save it on the Web server, in the **webApps** directory and in the same domain as the application server. The following is an example of the path for an application called **cyclos**:
  >
  > ```
  > C:\tomcat7\webapps\cyclos\clientmon.js
  > ```

- **RUM Client Monitor Probe HTTP URL:** Enter the URL of the RUM Browser Probe to which the monitored client data is sent. The format for the URL is: <protocol>://<host>:<port>/clientmon/data
- **RUM Client Monitor Probe HTTPS URL:** Enter the URL of the RUM Browser Probe to which the monitored client data is sent, if using https. The format for the URL is: <protocol>://<host>:<port>/clientmon/data

Select **Next** or **Finish**. Only one of these options is enabled, depending on the selections made in "Step 3: Specify the Agent Mode" on page 19

> **Note:** For details on the RUM Client Monitor-Diagnostics integration, including how to configure these settings manually, refer to the RUM Client Monitor-Diagnostics Integration Guide located on the Micro Focus Software Support web site (https://softwaresupport.softwaregrp.com/).

Setup Process Begins

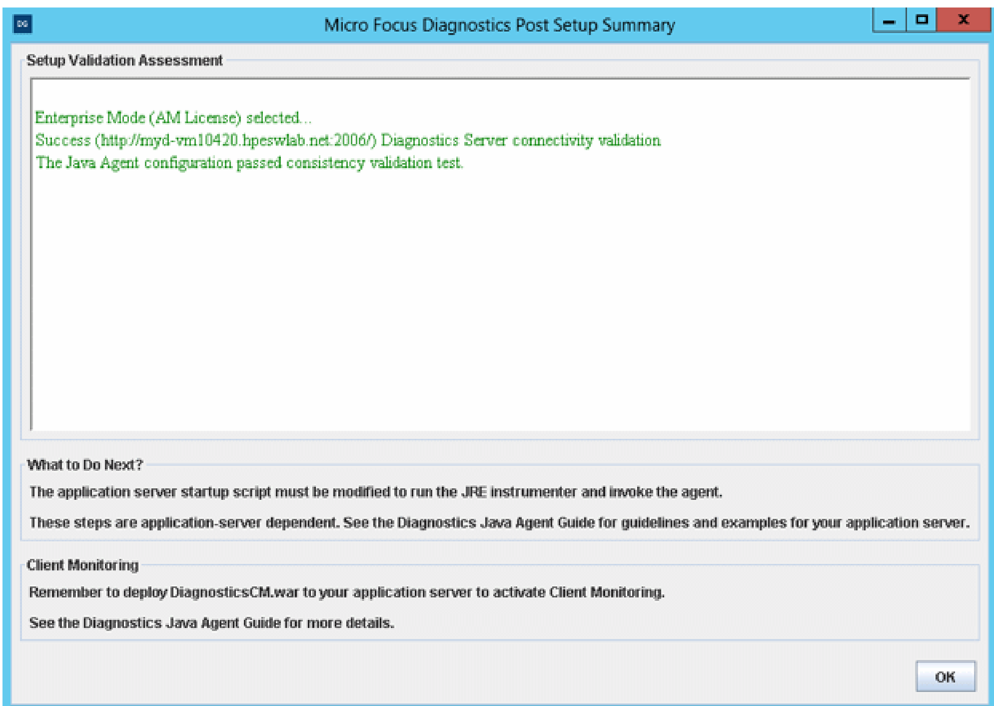
The Java Agent Setup process begins. In graphical mode a progress bar indicates how the configuration is proceeding.

If applicable, the connectivity to the Diagnostics Server is tested. If any connectivity problems are encountered, the Set Up Program displays the results of the connectivity check.

Continue with he next step

Step 7: Review Post Setup Summary

Review the Post Setup Summary.



If you chose to auto-deploy the agent, the summary includes the name of the modified application server startup script:

If no errors are reported, the agent has been configured successfully. If errors are reported, check the following:

- Whether the specified Diagnostics Server host name and port are correct. If a proxy server was specified, verify that the proxy host name and port are correct
- Whether the Diagnostics Server is started. See "Starting and Stopping Diagnostics Servers" in the Diagnostics Server Installation and Administration Guide.

- Whether network problems are affecting the general connectivity between the server host and the agent host, or between the proxy host and the agent host. For example, use the ping utility.
- If errors are related to the auto-deployment of the agent on JBoss,Tomcat, or WebSphere, make sure the user running the Agent Setup has permission to modify the application startup script or xml file and write files in that directory. Also check the following log file: **<agent_install_directory>/bin/setupModule.log**.
- If errors are related to monitoring profiles, check the relevant file or property and correct as necessary.

  Note: You can run the monitoring profile checking tool manually at any time from a command line, using the command : `<agent_install_directory>\bin\validator.cmd all` for Windows, or `<agent_install_directory>/bin/validator.sh all` for Linux and Unix.

Click **OK.**

Step 8: Verify Connectivity from the Agent to the Diagnostics Server

Optionally, to verify the Java Agent configuration and connectivity with the Diagnostics Server, you can run the following test scrip at any time:

- **<agent_install_dir>\bin\runTestProbe.cmd** on Windows
- **<agent_install_dir>/bin/runTestProbe.sh** on UNIX and Linux

This script runs a test probe that attempts to connect to the Diagnostics Server. The script displays log messages that indicate success or why the test probe is failing to connect. The failure messages can help you identify why the probe for your monitored application is not connecting to the Diagnostics Server.

Press **CTRL-C** to stop the test script.

The next step is to instrument the JRE and configure the application startup script to run the agent with the application server to be monitored. The way that you do this depends on whether the agent is being auto-deployed, as follows:

- If you specified the auto-deployment of the agent on JBoss, Tomcat, or WebSphere, the startup scripts or xml file have been modified as described in "Step 4: Specify Agent Name, Group, and Auto-deployment" on page 20. Simply restart the application server to pick up the changes.
- Otherwise, you need to instrument the JRE and modify the application server startup scripts to configure the application server to run with the agent. Follow the instructions in "Preparing Application Servers for Monitoring with the Java Agent" on page 30.

For more information on client monitoring see "Preparing Application Servers for Client Monitoring with the Java Agent" on page 78.

Silent Installation of the Java Agent

This section describes how to install the Java Agent in multiple locations using the same configuration files.

To install multiple Java Agents using a single set of configuration files:

1. Install the Java Agent temporarily, as described in "Installing and Configuring Java Agents" on page 17.
2. For each location in which you want to install the Java Agent:
   a. Extract all contents of the Java Agent installation package to a directory on the target host. For details, see "Step 1: Obtain the Installation Package" on page 17.
   b. Overwrite the contents of the Java Agent **etc** folder with the contents of the **etc** folder of the

temporary installation created in step **1** above.

c. Update the **id** property in the **etc\probe.properties** file with the id of Java probe you are configuring.

d. Instrument the JRE and configure the application startup script to run the agent with the application server to be monitored as described in "Step 8: Verify Connectivity from the Agent to the Diagnostics Server" on the previous page.

Setting File Permissions

This procedure is relevant for AIX, Linux, or Solaris installations only.

After installing the Java Agent, make the agent's 'group' the same as the application server's 'group'. Then assign the following permissions to files in the <probe install directory> for the group:

- Read access to the **<agent_install_directory>** directory and files.
- Execute access to the **<agent_install_directory>/bin** directory.
- Read/Write access to the **<agent_install_directory>/log** directory.

Determining the Version of the Java Agent

When you request support, it is useful to know the version of the Diagnostics component you have a question about.

You can find the version of the Java Agent in one of the following ways:

- In the file **<agent_install_directory>\version.txt**. The file contains the version number, as well as the build number.
- In the probe log file **<agent_install_directory>/log/<probe_id>/probe.log**.
- For agents in Enterprise mode, in the System Health view of the Diagnostics UI.

Configuring for Firewalls, HTTPS, and Proxies

The Java Agent requires additional configuration if it is being deployed into an Enterprise Diagnostics environment that includes firewalls, SSL-enabled communications, and proxies. This configuration is described in the Diagnostics Server Installation and Administration Guide. See the following sections in that guide:

- “Configuring Diagnostics Servers and Agents for HTTP Proxy”
- "Configuring Diagnostics to Work in a Firewall Environment"
- "Enabling HTTPS Between Components"

Uninstalling the Java Agent

To uninstall the Java Agent:

1. Stop the application server that is being monitored by the Java probe.
2. Restore the original application server startup script or remove any modifications that were made to the script to enable monitoring, for example on JBoss you would remove the following:

```
# Configuring Diagnostics Java Agent
AGENT_HOME=<agent_install_dir>
PROBE_ID=<probe_id>
...
PROBE_OPTS="$PROBE_OPTS -
Djboss.modules.system.pkgs=org.jboss.byteman,com.mercury.opal"
JAVA_OPTS="$JAVA_OPTS $PROBE_OPTS"
```

If the agent was auto-deployed, restore the backup copy of the script. See "Step 4: Specify Agent Name, Group, and Auto-deployment" on page 20.

3. Delete the entire <agent_install_directory> directory.

Chapter 4: Preparing Application Servers for Monitoring with the Java Agent

This chapter describes how to prepare your application servers to allow the Diagnostics Java Agent to monitor your applications.

This chapter includes:

- "About Preparing Application Servers for Monitoring" below
- "Examples for Configuring Application Servers " on page 33
- "Verify the Application Server is Running the Java Agent" on page 58
- "About the JRE Instrumenter and Different Options to Invoke" on page 58
- "Other Configuration Options" on page 65

About Preparing Application Servers for Monitoring

After the Diagnostics Java Agent is installed and configured, the application server must be prepared (instrumented) to allow the Java Agent to monitor the applications. This preparation usually involves **instrumenting the JRE** used by the application servers and **configuring the application server** JVM parameters to invoke the Java Agent.

Diagnostics' JRE instrumentation does not modify the installed JRE, but rather places copies of instrumented classes under the Java Agent installation directory. Then with the proper JVM parameters these instrumented classes will be loaded into the JVM that runs your application server. The instrumentation is done using the Diagnostics JRE Instrumenter utility which can be invoked automatically using various options or manually.

There are two-levels of instrumentation:

- Basic instrumentation.

  By adding the Java Agent to your application server start up, your application server will be instrumented and monitored. This is done by adding the -javaagent option to your application server JVM parameters.

- Recommended instrumentation.

  In addition to the basic instrumentation, we recommend that you also instrument the JRE (Java Runtime Environment) used by your application server using the JRE Instrumenter utility provided by the Java Agent. This extra instrumentation will enable the Java Agent to provide advanced features such as the patent-pending Collection Leak Pinpointing (CLP). CLP automatically detects leaking collections and provides a stack trace of where the leak occurs. This helps identify issues early, while there is time to mitigate the issue (such as an eventual out of memory error/server crash), as well as saves developers time by avoiding the tedious task of analyzing heap dumps (see "Configuring Collection Leak Pinpointing" on page 123). And this extra instrumentation also has performance benefits on certain application servers such as WebSphere 6.1.

Note: If you chose to auto-deploy the application server during agent setup, you do not need to perform this procedure. Restart the application server to pick up the changes.

For general instructions on using the different JRE instrumentation modes see "About the JRE Instrumenter and Different Options to Invoke" on page 58.

To continue, find your application server in the next section and follow the instructions for instrumenting and configuring.

Specifying Probe Properties as Java System Properties

The configuration of the Java Agent is managed by property settings in several property and configuration files. You can view and modify these files in **<agent_install_directory>/etc/**. Property settings can also be specified as Java system properties on the startup command line for the application server, where they configure only that instance of the probe. These system properties can be specified in the following ways:

- "Specified individually on the command line" below
- "Grouped in a file that is specified on the command line" on the next page
- "Macros for probe and host naming" on page 33

Specified individually on the command line

Except for those defined in the **dynamic.properties** property file, all probe properties can be specified as Java System properties on the startup command line for the application server.

When the application starts, properties specified in the startup command line override properties with the same name in the corresponding property file. If you make a change to the dynamic property settings while an application is running, these changes will override the command-line specification.

Specifying probe properties on the application startup command line is useful when there is more than one JVM being monitored by a single agent installation. Each probe can specify its own configuration as a delta to the shared agent configuration and property files.

To specify a property as a Java System property, add **-D** to the first part of the module name or properties file name, for example **-Dprobe** or **-Ddispatcher**. See the following examples.

- For the property **webserver.jetty.port**, from the startup command, add **-D** before the module name (**probewebserver**) as follows:

```
-Dprobewebserver.jetty.port=SomePortNumber
```

Note: The webserver property is different from other properties as you need to use the module name (probewebserver), not the property file name.

- To set the **id** property in **probe.properties** from the startup command, add **-D** before **probe** in the property file name, and add the name of the property you are specifying (**id**) as follows:

```
-Dprobe.id=SomeId
```

- To set the **active.products** property in **probe.properties** from the startup command, add **-D** before **probe** in the property file name, and add the name of the property you are specifying (**active.products**), as follows:

```
-Dprobe.active.products=Enterprise
```

- To set the **registrar.url** property in **dispatcher.properties** from the startup command, add **-D** before **dispatcher** in the property file name, and add the name of the property you are specifying (**registrar.url**), as follows:

  ```
  -Ddispatcher.registrar.url=http:/host.company.com:2006/registrar
  ```

- To set the **minimum.sql.latency** property in **dispatcher.properties** from the startup command, specify a value as follows:

  ```
  -Ddispatcher.minimum.sql.latency=3s
  ```

  Because this property is dynamic, you can override the above specification by modify the setting in the following file:
  <agent_install_directory>/etc/dispatcher.properties

  Example

  ```
  # If an SQL statement takes less than this amount of time, it will
  # not be trended, until it does exceed this time.
  # (This property can be dynamically changed)
  minimum.sql.latency = 1s
  ```

  In this case, the setting is restored to its default value of 1 second.

Grouped in a file that is specified on the command line

As an alternative to specifying individual probe properties on the startup command line for the application server, you can group several property settings together in a file and specify the file as a Java System property on the startup command line for the application server.

Just as with the command-line specification, any properties specified in the file override those of the same names in the corresponding property files when the application starts. However when using the file method, you can include properties from the **dynamic.properties** property file. However, unlike when specifying individual probe properties on the command line, all properties are overridden unconditionally. Any changes to dynamic settings that occur once the application is running **do not** override their specification in the file.

Using a file to specify a number of probe properties is helpful when you have many properties to specify, or the property settings require unusual syntax which is easier to maintain in a file.

To specify a file that contains property settings on the application server startup command line, specify **-Ddiag.config.override=<my_prop_settings>**
where <my_property_settings> specifies the file with your settings that you have created and placed in **<agent_install_ directory>/etc/overrides**. The file must contain the **.settings** suffix.

For example:

```
-Ddiag.config.override=WebSphereProbe24
```

This directs the probe to read the file: **<agent_install_directory>/etc/overrides/ WebSphereProbe24.settings** file. This file contains any settings that you want to override at startup, for example:

```
probe.id=SomeId
probe.active.products=Enterprise
dispatcher.registrar.url=http:/host.company.com:2006/registrar
dispatcher.minimum.sql.latency=3s

dynamic.stack.trace.sampling.rate=30ms
```

Macros for probe and host naming

Probe name, host name and IP address can be specified by using macros. The macros pull values from system properties or environment variables and use the values to build the name or IP address at runtime.

Macros for probe and host naming are useful in cloud environments.

<table>
<tr><td>Where macros can be specified</td><td>Macros can be specified for any of the following properties:
probe.id
dispatcher.probe.host.ip_address.override
dispatcher.probe.host.name.override</td></tr>
<tr><td>Macro format</td><td>You specify a macro in either of the following formats:
<code>${key}</code>
or
<code>${key:subkey}</code>
where:
<code>key</code> is a system property or environment variable. The value of the system property or the environment variable is used as the macro value.
<code>subkey</code> is specific field of the <code>key</code> value. The key value must be in a JSON map form.</td></tr>
<tr><td>Examples</td><td>For example, assume <agent_install_directorydir>etc/probe.properties contains the following entry:
<code>id = ${PARAMETERS:username}-${PARAMETERS:port}_foo</code>
If the PARAMETERS environment variable has a value of:
{"username":"joe","user_id":1003,"port":3003}
Then the id property a evaluates to:
<code>id = joe-3003_foo</code></td></tr>
</table>

Examples for Configuring Application Servers

This section provides examples of how to configure various commonly used application servers for monitoring. See the section "About the JRE Instrumenter and Different Options to Invoke" on page 58 for a description of the different ways you can invoke the JRE Instrumenter.

Note:

- Make sure that you understand the structure of the startup scripts, how the property values are set, and the use of environment variables before you make any application server configuration changes. Always create a backup copy of any file that you plan to update before making the changes.
- For JBoss, Tomcat, and WebSphere application servers, we recommend that you use the auto-deploy option. For details, see "Step 4: Specify Agent Name, Group, and Auto-deployment" on page 20.

"Example 1: Configuring GlassFish Application Server for Monitoring" below

"Example 2: Configuring JBoss Application Server and JBoss EAP for Monitoring" on page 36

"Example 3: Configuring Oracle Application Server for Monitoring" on page 39

"Example 4: Configuring SAP NetWeaver Application Server for Monitoring" on page 42

"Example 5: Configuring TIBCO ActiveMatrix BusinessWorks and Service Bus for Monitoring" on page 44

"Example 6: Configuring Tomcat Application Server for Monitoring" on page 46

"Example 7: Configuring WebLogic Application Server for Monitoring" on page 48

"Example 8: Configuring webMethods Server for Monitoring" on page 49

"Example 9: Configuring WebSphere Application Server for Monitoring" on page 53

"Example 10: Configuration for WebSphere Application Server Liberty " on page 56

The long lines in the script examples shown in this chapter do not have line breaks, which makes them hard to read. However this allows you to copy and paste the text directly from the manual (when viewing online) and into your editor without extraneous formatting characters.

Use quotes if there are spaces in the files paths that you specify.

Example 1: Configuring GlassFish Application Server for Monitoring

The following are the instructions for a generic GlassFish 3.x or 4.x application server implementation. Your site administrator should be able to use these instructions to guide you in making the changes that are appropriate to your specific environment.

Note: 1.GlassFish requires additional, special settings to work properly with the agent.

Locate the property **org.osgi.framework.bootdelegation** in the GlassFish configuration files and append the text **",com.mercury.opal.capture.proxy"** to the end of the property value (do not include the quotes).

In GlassFish 3.1.2 and later, this property is located in **<GlassFish_install_dir>/glassfish/config/osgi.properties**.

Locate the property **extra-system-packages** and append the text **,com.mercury.opal.capture.proxy;version=<Java probe version>**. An example version numbers is 9.50.1.153

In an earlier versions of GlassFish, these properties may reside in the following two files:

< GlassFish_install_dir >/osgi/equinox/configuration/config.ini

< GlassFish_install_dir >/osgi/felix/conf/config.properties

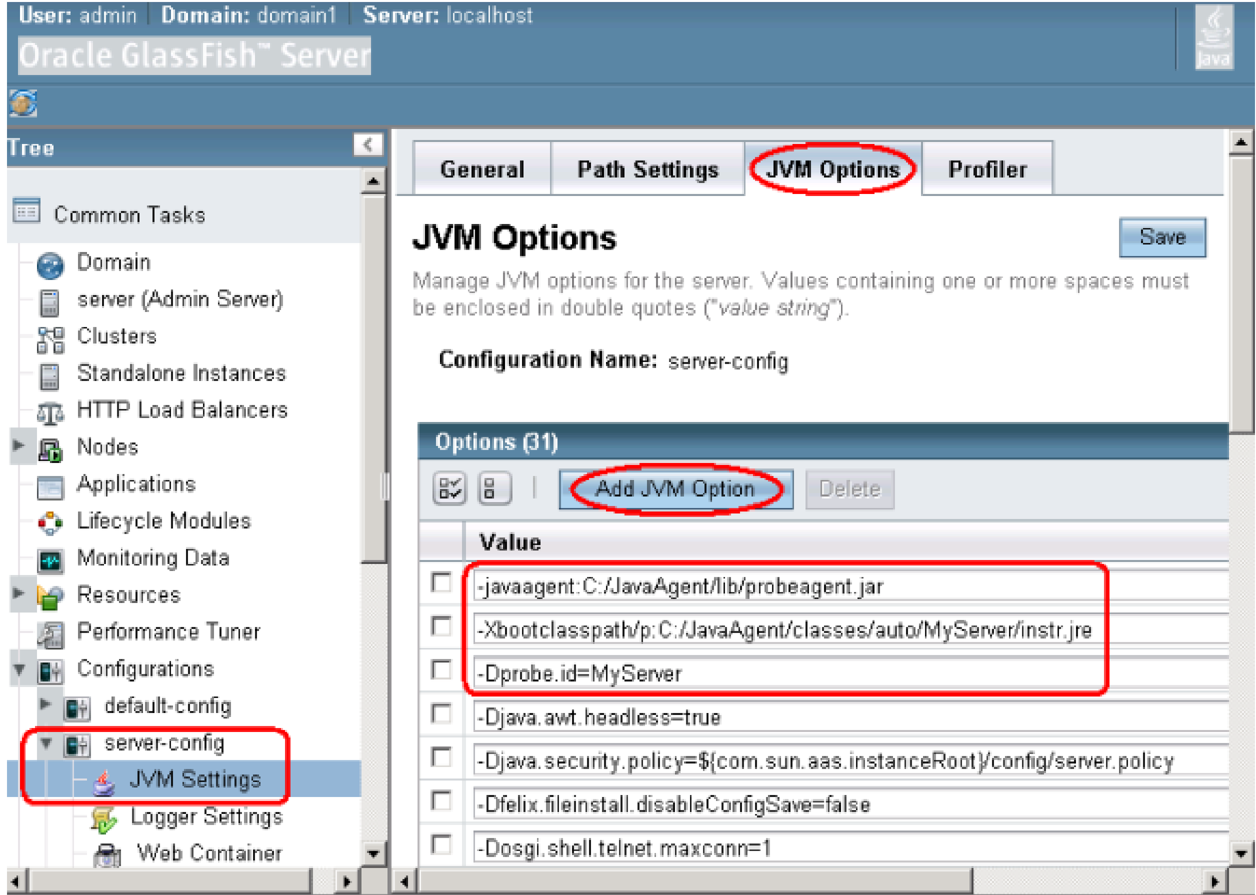
You may also need to disable the Monitoring Service on GlassFish to avoid a conflict with the Diagnostics monitoring. Go to **Configurations > {config_name} > Monitoring** and deselect the **Enabled** check box of the Monitoring Service option.

1. Locate the GlassFish JVM configuration settings by logging in to the GlassFish Administration Console and navigating to the **JVM Options** page.

   For GlassFish 3.1.2 and later, in the left-hand tree go to **Configurations > {config_name} > JVM Settings**, where {config_name} is the name of your server configuration (such as, **server-config**).

   If you are working with an earlier version of GlassFish, click **Application Server** in the left-hand tree and then select the JVM Settings tab at the top.

   Then select the **JVM Options** tab. See the screenshot below as a reference.



2. Using the **Add JVM Option** button, add the following JVM parameters, one at a time. For <agent_install_dir> use the full path to where you installed the agent. On Windows, use forward slashes (/) instead of backward slashes (\). For <probe_id> use a name you've chosen for the probe, such as MyServer.

```
-javaagent:<agent_install_dir>/lib/probeagent.jar
-Xbootclasspath/p:<agent_install_dir>/classes/auto/<probe_id>/instr.jre
-Dprobe.id=<probe_id>
```

Note: In case of cluster setup, remove **-Dprobe.id=MyServer** in JVM options tab and add different

probe ids in the probe.properties file located in **<Diag_Java_Agent_Path>\etc** for both servers, so that you can see both cluster instance in Diagnostics.

3. Restart the GlassFish application server.

   If the GlassFish application server does not start, you can check and change the JVM parameters in the **<GlassFish_install_dir>/glassfish/domains/<domain_name>/config/domain.xml** file to resolve the issue, where <domain_name> is the name of your domain (such as, **domain1**).

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Note: If you update the JRE used by your application server in the future, you must delete the **<agent_install_dir>/classes/auto/<probe_id>** directory so that the new JRE will be instrumented. Otherwise, your application server may not start. . For general information on the instrumentation mode used see "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.

Example 2: Configuring JBoss Application Server and JBoss EAP for Monitoring

The following sections provide instructions with specific examples for the JBoss Application Server and JBoss EAP (Enterprise Application Platform) for a generic implementation. Your site administrator should be able to use these instructions to guide you to make these changes in your customized environment.

Note: For JBoss 6.x, add the following JVM parameter:
-Djava.util.logging.manager=org.jboss.logmanager.LogManager

For JBoss Application Server, if you chose to auto-deploy the application server during agent setup, you do not need to perform this procedure. Restart the application server to pick up the changes.

To configure a JBoss application server:

1. Locate the startup script that is used to start JBoss for the application and locate a convenient point in the file after all options are set but before the java command line (or code block) that starts the application server is executed.

   - On JBoss versions earlier than 7.0:

     The startup script file is typically located in a path similar to the following:

     <JBOSS_HOME>\bin\run.[bat|sh]

     where **<JBOSS_HOME>** is the path to your JBoss installation directory, such as **C:\ jboss-4.2.3.GA or C:\jboss-6.0.0.Final**.

   - On JBoss 7.0 or higher:

     The startup script file is typically located in a path similar to one of the following:

     <JBOSS_HOME>\bin\domain.[bat|sh]

     <JBOSS_HOME>\bin\standalone.[bat|sh]

     where **<JBOSS_HOME>** is the path to your JBoss installation directory, such as **C:\jboss-as-7.1.0.Final**.

2. Insert additional configuration lines as illustrated by the examples. In the example you should replace <agent_install_dir> and <probe_id> with values for your environment.

   Below is an example showing the modified .bat file for JBoss 6.x:

```
rem Setup JBoss specific properties
rem Setup the java endorsed dirs
set JBOSS_ENDORSED_DIRS=%JBOSS_HOME%\lib\endorsed

rem Configuring Diagnostics Java Agent
set AGENT_HOME=<agent_install_dir>
set PROBE_ID=<probe_id>
"%JAVA%" -jar %AGENT_HOME%\lib\jreinstrumenter.jar -f %PROBE_ID%
set PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\%PROBE_ID%\instr.jre
set PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
rem Use the line below ONLY for JBoss 6
set PROBE_OPTS=%PROBE_OPTS% -
Djava.util.logging.manager=org.jboss.logmanager.LogManager
rem Use the line below ONLY for JBoss 7
rem set PROBE_OPTS=%PROBE_OPTS% -
Djboss.modules.system.pkgs=org.jboss.byteman,com.mercury.opal
set JAVA_OPTS=%JAVA_OPTS% %PROBE_OPTS%
```

Below is an example showing the modified .sh file for JBoss 7.x, 8.x (Wildfly):

```
if $cygwin; then
 JBOSS_HOME=`cygpath --path --windows "$JBOSS_HOME"`
 JAVA_LOC=`cygpath --path --windows "$JAVA_LOC"`
 JBOSS_CLASSPATH=`cygpath --path --windows "$JBOSS_CLASSPATH"`
 JBOSS_ENDORSED_DIRS=`cygpath --path --windows "$JBOSS_ENDORSED_DIRS"`
fi

# Configuring Diagnostics Java Agent
AGENT_HOME=<agent_install_dir>
PROBE_ID=<probe_id>
"$JAVA" -jar $AGENT_HOME/lib/jreinstrumenter.jar -f $PROBE_ID
PROBE_OPTS="-Xbootclasspath/p:$AGENT_HOME/classes/$PROBE_ID/instr.jre"
PROBE_OPTS="$PROBE_OPTS -javaagent:$AGENT_HOME/lib/probeagent.jar"
PROBE_OPTS="$PROBE_OPTS -Dprobe.id=$PROBE_ID"
# Use the line below ONLY for JBoss 6
# PROBE_OPTS="$PROBE_OPTS -
Djava.util.logging.manager=org.jboss.logmanager.LogManager"
# Use the line below ONLY for JBoss 7
PROBE_OPTS="$PROBE_OPTS -
Djboss.modules.system.pkgs=org.jboss.byteman,com.mercury.opal"
JAVA_OPTS="$JAVA_OPTS $PROBE_OPTS"

# Display our environment
echo "========================================================================="
echo ""
echo " JBoss Bootstrap Environment"
echo ""
```

```
echo " JBOSS_HOME: $JBOSS_HOME"
```

Note: If your java command line does not use the JAVA_OPTS variable to define the JVM parameters, you need to change the variable name JAVA_OPTS shown in these examples to the correct name.

3. Save the changes to the startup script and restart the application server using the modified script.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Configuring a JBoss EAP Application

Following is an example for JBoss EAP (Enterprise Application Platform) for a generic implementation.

To Configure a JBoss EAP Application:

1. Locate and edit **domain.xml** for the domain.
   By default, this is located in the following folder:
   $JBOSS_HOME/domain/configuration/.
2. In the **system-properties** element, add a property named **jboss.modules.system.pkgs** with a value of **com.mercury.opal** to the existing system properties.

   Example:

   ```
   <system-properties>
      <property name="jboss.modules.system.pkgs"
       value="org.jboss.byteman,com.mercury.opal"/>
   </system-properties>
   ```

   This property tells the JBoss class loader to load the Diagnostics packages. This is required for the Java Agent to run.

3. Under the server group name where you want to enable the Java Agents, add the JVM options using the required values for the agent location, JBoss application name, and tier name.

   Example:

   ```
   <server-group name="main-server-group" profile="full-ha">
     <jvm name="default">
       <jvm-options>
         <option value="-Xbootclasspath/p:/home/x001059a/JavaAgent/DiagnosticsAgent
           /classes/Oracle/1.7.0_40/instr.jre"/>
         <option value="-javaagent:/home/x001059a/JavaAgent/DiagnosticsAgent
           /lib/probeagent.jar"/>
         <option value="-Djava.util.logging.manager=org.jboss.logmanager.
           LogManager"/>
         <option value="-Dprobe.id=CARSCA_STGM_AppSrv"/>
       </jvm-options>
     </jvm>
   </server-group>
   ```

4. If you are using JBoss EAP 6.x, set **mercury.enable.jboss6eap=true** for **detials.conditional.properties** in **inst.properties.**
5. Restart the JBoss application server after the changes.

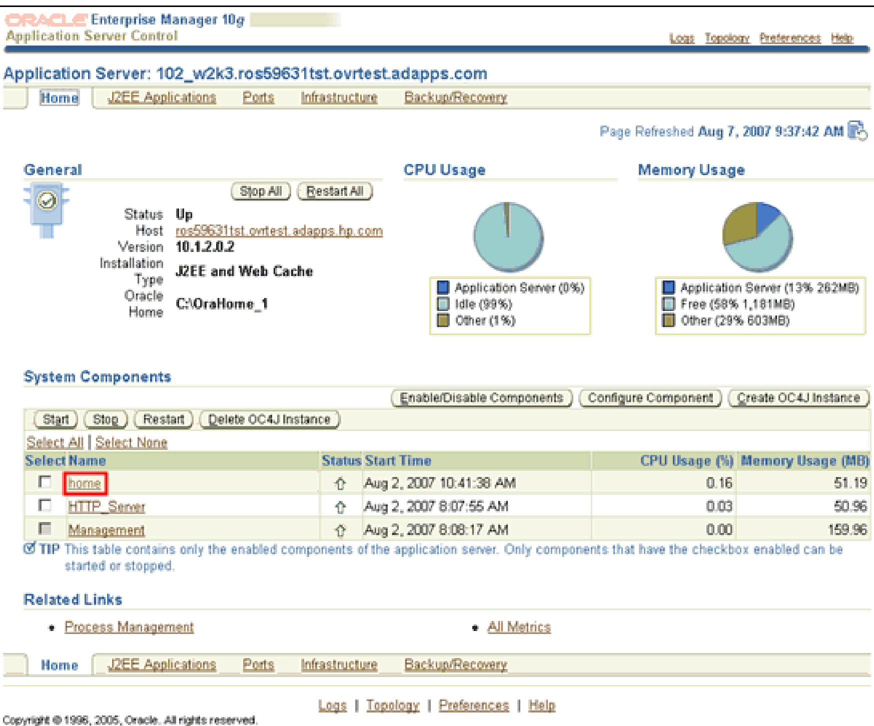
Example 3: Configuring Oracle Application Server for Monitoring

This section provides instructions for configuring an Oracle 10g application server.

> **Note:** Some of the Web Services deployed on Oracle OC4J application server, due to their non-compliance to the JAX-WS standard, may not be recognized by Diagnostics agent. To work around this issue you can try setting **annotation.inheritance.allow=true** in **etc/inst.properties** on the agent system.

To configure an Oracle 10g application server:

1. Locate the Oracle Application Server JVM configuration settings by opening Oracle's **Application Server Control Console**, select **home** (or MyOC4J) System Component, and then **Administration**.

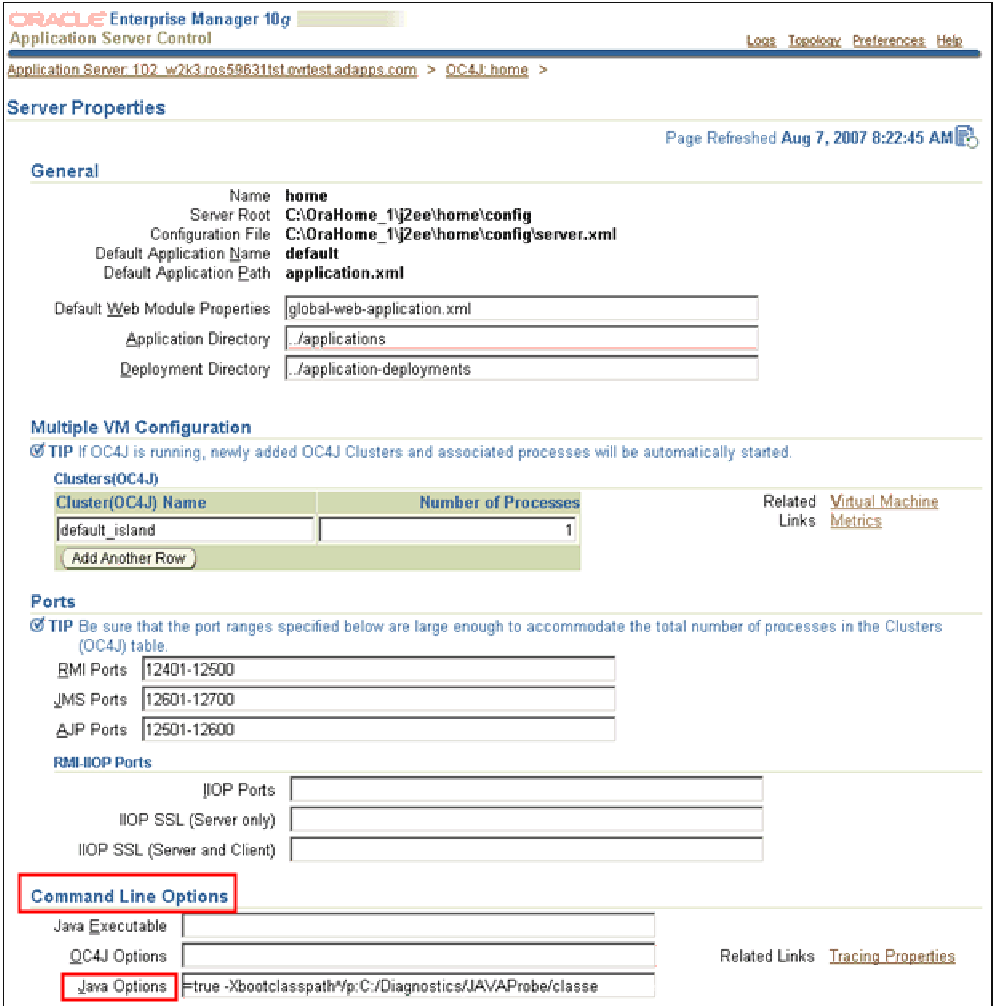


On the Administration page, select **Server Properties**. You'll input in JVM parameters under **Command Line Options**.

2. Run the Diagnostics JRE Instrumenter to instrument the JRE used by your Oracle application server. See "Using the JRE Instrumenter in Manual Mode" below.

   Copy the JVM parameters provided by this tool and paste them in the Command Line Options "Java Options" text field found in the previous step and shown in the following figure.

   > **Note:** It is required to add a (**^**) prior to the /p switch or Oracle will change the (**/**) switch option to a (\).



3. Apply the changes and restart the Oracle server.

See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Note: If you update the JRE used by your application server in the future (such as applying an application server patch), you must run the JRE Instrumenter again to instrument the new JRE and change the JVM parameters accordingly. Otherwise, your application server may not start.

Using the Diagnostics JRE Instrumenter in Manual Mode

Manually invoke the JRE Instrumenter and copy the provided JVM parameters into your application server startup settings.

Note: If you update the JRE used by your application server in the future (such as applying an application server patch), you must run the JRE Instrumenter again to instrument the new JRE and change the JVM parameters accordingly. Otherwise, your application server may not start.

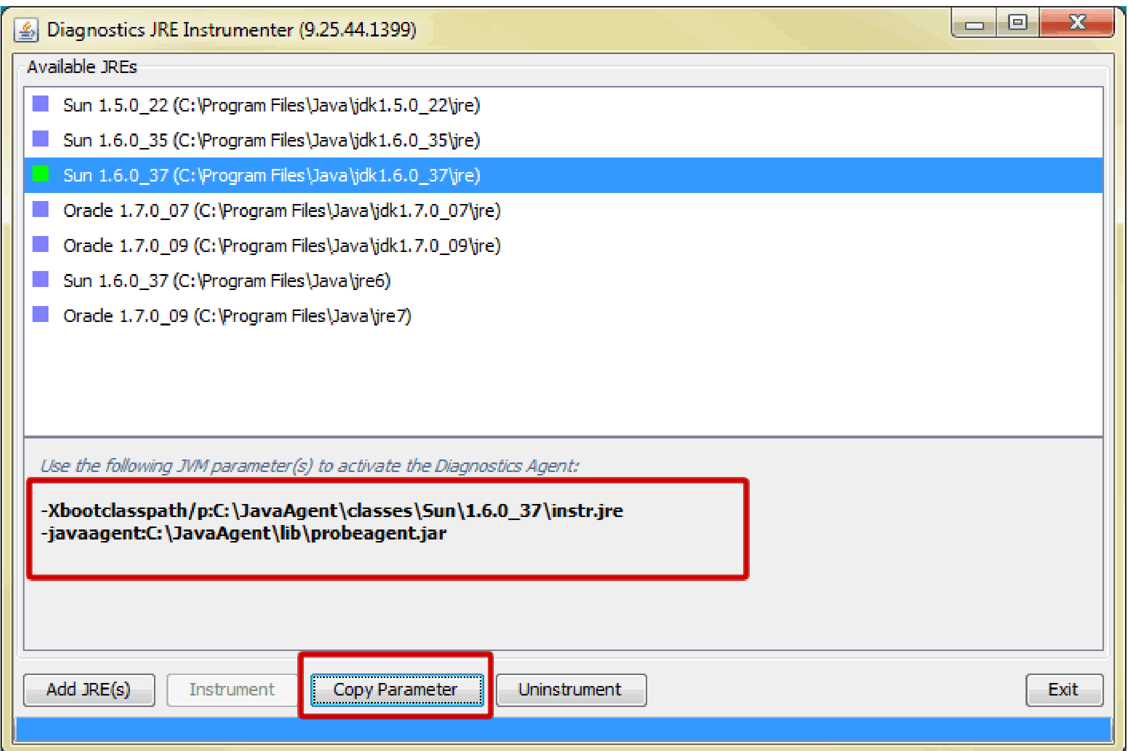
By default, the JRE Instrumenter uses a graphical user interface (UI Mode). Directions to run the JRE Instrumenter from a console window (Console Mode) follow below.

Running the JRE Instrumenter Utility in UI Mode

1. Start the JRE Instumenter utility.

   On Windows run the **<agent_install_dir>\bin\jreinstrumenter.cmd** command.

   On UNIX or Linux run the **<agent_install_dir>/bin/jreinstrumenter.sh** command.
2. Click the **Add JRE(s)** button, navigate to a parent directory where the JRE used by your application is stored and click **Search from here**. The JRE Instrumenter lists the JREs found in the Available JREs list.
3. Select the JRE that is used by your application and then click **Instrument**. The JRE Instrumenter instruments some of the classes for the selected JRE and places the instrumented classes in a folder under the **<agent_install_dir>/classes** directory.
4. Click **Copy Parameter** to copy the JVM parameters in the box below the Available JREs list, to the clipboard.



5. Click **Exit** to close the JRE Instrumenter window and continue with configuring your application server JVM parameters.

Example 4: Configuring SAP NetWeaver Application Server for Monitoring

The following are the instructions for a generic NetWeaver application server implementation. Your site administrator should be able to use these instructions to guide you in making the changes that are appropriate to your specific environment.

> **Note:** SAP NetWeaver requires additional, special settings to work properly with the agent.

Edit the <agent_install_dir>\etc\capture.properties file and assign the following values to these properties:

event_buffer.size = `10000`

event_buffer.flush.level = `1000`

To configure a SAP NetWeaver application server:

1. Locate the NetWeaver JVM configuration settings by running the NetWeaver J2EE Engine configuration tool. The script to run this tool is called **configtool.bat** and is located in the **usr\sap\CR2\JC00\j2ee\configtool** directory, where CR2 is an example of the name of your SAP system.

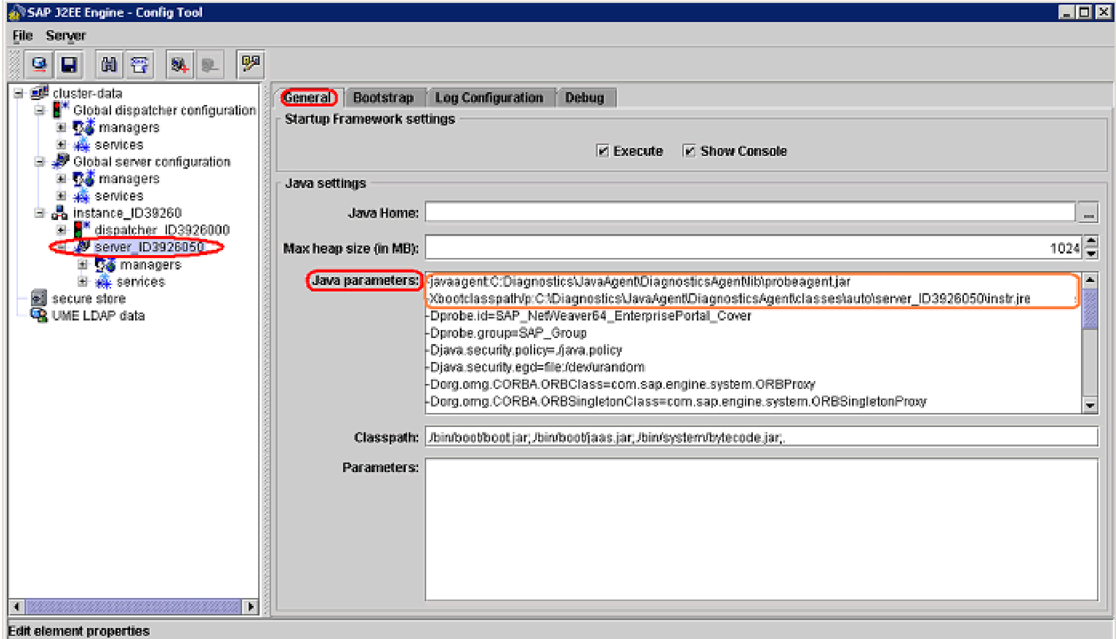
   In the configuration tool UI, in the left-hand tree, select the server that you want to monitor. For example in the screenshot below, select **cluster-data > instance_ID39260 > server_ID3926050**. Then, at the right-hand side select the **General** tab where you'll find the **Java parameters** text window.

2. Add the following JVM options to the Java parameters text window. In the example you should replace <agent_install_dir> and <probe_id> with values for your environment.

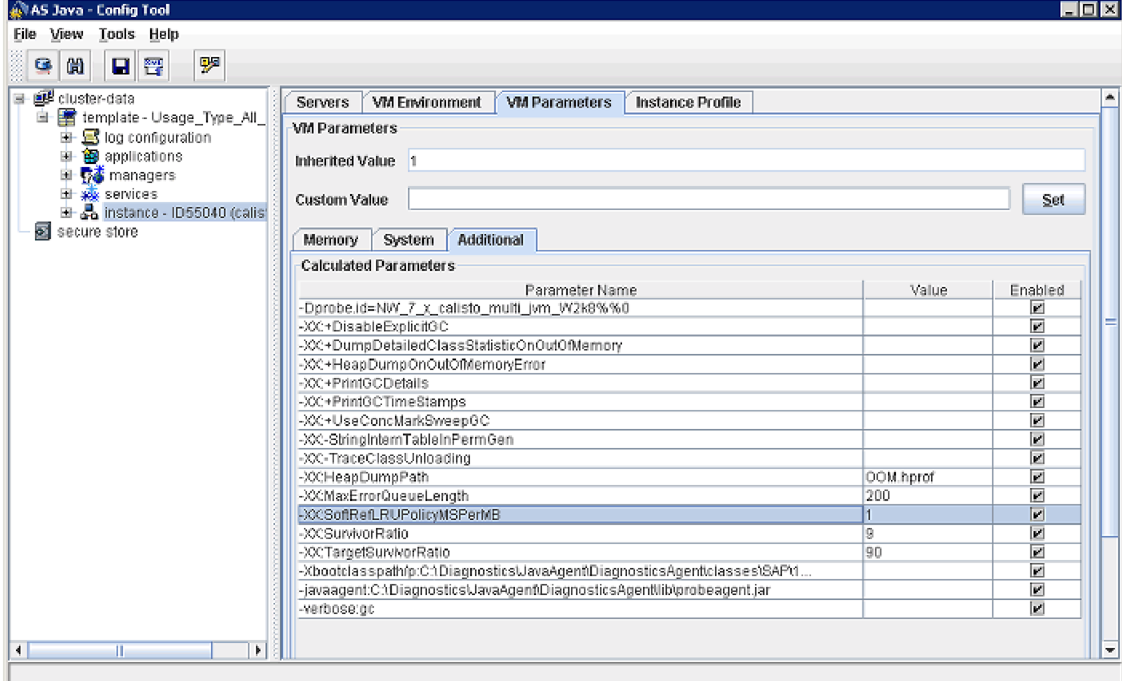
```
-javaagent:<agent_install_dir>\lib\probeagent.jar
-Xbootclasspath/p:<agent_install_dir>\classes\auto\<probe_id>\instr.jre
-Dprobe.id=<probe_id>
```

   Note: In a clustered environment where a single startup script is used to start multiple probed application server instances you need to add a suffix (%0) to the parameter -Dprobe.id=<probeName>%0. This will generate a custom probe identifier for each probe. On Windows, use %%0 (the first % is used to escape the second %).

   The following is an example screen for SAP NetWeaver versions 7.1 or earlier with the JVM parameters highlighted.



The following is an example screen for SAP NetWeaver verison 7.3. You enter the JVM parameters in the Custom parameters box and you must enter each parameter separately (-javaagent, -Xbootclasspath and -Dprobe.id).



3. Save your changes and exit the configuration tool and restart the application server.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

> **Note:** If you update the JRE used by your application server in the future (such as applying an application server patch), you must delete the **<agent_install_dir>/classes/auto/<probe_id>** directory so that the new JRE will be instrumented. Otherwise, your application server may not start For general information on the instrumentation mode used see "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.

Example 5: Configuring TIBCO ActiveMatrix BusinessWorks and Service Bus for Monitoring

The following sections describe the steps to configure TIBCO ActiveMatrix BusinessWorks and Service Bus so that the applications can be monitored.

To configure TIBCO ActiveMatrix BusinessWorks:

Configuring a TIBCO BusinessWorks application server involves modifying its configuration files to add JVM parameters. Below are the instructions for a generic server implementation. Your site administrator should be able to use these instructions to guide you in making the changes that are appropritate to your specific environment.

1. Locate the TIBCO BusinessWorks **.tra** configuration files. These files are typically located in:

   <tibco_home>\tra\domain\<Domain_Name>\application\<Application_Name>\<Application_Name>.tra

2. Insert additional configuration lines as illustrated by this example. In the example you should replace

<agent_install_dir> and <probe_id> with values for your environment.

```
#
# Other arguments to application, JVM etc.
#
tibco.env.APP_ARGS=
tibco.env.HEAP_SIZE=256M

# Configuring Diagnostics Java Agent
tibco.env.AGENT_HOME=<agent_install_dir>
tibco.env.PROBE_ID=<probe_id>
JmxEnabled=true
tibco.env.PROBE_OPTIONS=-Xbootclasspath/p:%AGENT_HOME%/classes/auto/%PROBE_
ID%/instr.jre
tibco.env.PROBE_OPTIONS=%PROBE_OPTIONS% -javaagent:%AGENT_
HOME%/lib/probeagent.jar
tibco.env.PROBE_OPTIONS=%PROBE_OPTIONS% -Dprobe.id=%PROBE_ID%
java.extended.properties=%PROBE_OPTIONS%
```

> **Note:** If **java.extended.properties** already exists in the file, be sure to append the %PROBE_OPTIONS% to the existing definition. Also do not use backslashes (\) for any values. Instead replace them with forward slashes (/).

3. Save the changes to the startup script and restart the application using the modified script.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

To configure TIBCO ActiveMatrix Service Bus:

> **Note:** TIBCO ActiveMatrix Service Bus (AMSB) 3.1.2 requires additional, special settings to work properly with the agent.
>
> Locate the TIBCO ActiveMatrix Service Bus 3.1.2 machine.xmi file. This file is typically located in a path such as:
>
> ```
> <tibco_amx_configuration_dir>\data\tibcohost\<EnterpriseName_
> ServerName>\tools\machinemodel\machine.xmi
> ```
>
> Update the **runtimes** section of the file **for each node** you want to monitor. For example:
>
> `<runtimes xsi:type="machinemodel:OSGiRuntime"` **name="Node1"**`"`
>
> In the runtimes section for each node locate the **frameworkProperties** key **org.osgi.framework.bootdelegation** and append **com.mercury.*** to the value of the property.
>
> For example:
>
> `<frameworkProperties key="org.osgi.framework.bootdelegation" value="com.ibm.*, ....,sun.*,`**com.mercury.***`"/>`

1. Locate the TIBCO ActiveMatrix Service Bus .tra configuration files.

   On TIBCO ActiveMatrix Service Bus (AMSB) 2.0 and 2.3 these files are typically located in:

   <tibco_home>\amx\data\<Node>\<Application\bin

   On TIBCO ActiveMatrix Service Bus 3.1.2 these files are typically located in:

<tibco_amx_configuration_dir>\tibcohost\<EnterpriseName_ServerName>\nodes\<NodeName>\bin\tibamx_<NodeName>.tra

2. Insert additional configuration lines as illustrated by this example. In the example you should replace <agent_install_dir> and <probe_id> with values for your environment.

```
# NOTE:
# There must be only one java.extended.properties in the .tra file. Append remote
# debugging extended properties here to use remote debugging for this process.
#
# Configuring Diagnostics Java Agent
tibco.env.AGENT_HOME=<agent_install_dir>
tibco.env.PROBE_ID=<probe_id>
tibco.env.PROBE_OPTIONS=-Xbootclasspath/p:%AGENT_HOME%/classes/auto/%PROBE_
ID%/instr.jre
tibco.env.PROBE_OPTIONS=%PROBE_OPTIONS% -javaagent:%AGENT_
HOME%/lib/probeagent.jar
tibco.env.PROBE_OPTIONS=%PROBE_OPTIONS% -Dprobe.id=%PROBE_ID%
java.extended.properties=%PROBE_OPTIONS%
```

Note: If **java.extended.properties** already exists in the file, be sure to append the %PROBE_OPTIONS% to the existing definition. Also do not use backslashes (\) for any values. Instead replace them with forward slashes (/).

3. Save the changes to the startup script and restart the application using the modified script.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Note: If you update the JRE used by your application server in the future (such as applying an application server patch), you must delete the **<agent_install_dir>/classes/auto/<probe_id>** directory so that the new JRE will be instrumented. Otherwise, your application server may not start. For general information on the instrumentation mode used see "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.

Example 6: Configuring Tomcat Application Server for Monitoring

Apache Tomcat is frequently embedded into other applications or servers. As a result, the way to instrument it may vary. The following sections provide instructions on how to configure a Tomcat server in simple scenarios, but it is generic enough to guide you in your particular situation.

If you chose to auto-deploy the application server during agent setup, you do not need to perform this procedure. Restart the application server to pick up the changes.

If your Tomcat server is started by script, follow the instructions in "To configure a Tomcat server with a startup script:" on the next page.

If Tomcat is installed as a Windows service or has no scripts, follow the instructions in "To configure a Tomcat server without a startup script:" on the next page.

To configure a Tomcat server with a startup script:

1. Locate the startup script that is used to start Tomcat for the application and locate a convenient point in the file after all options are set but before the java command line (or code block) that starts the application server is executed.

   In some scenarios, the startup script will end up calling the following script to start Tomcat:

   <Tomcat_install_dir>/bin/catalina.[bat|sh]

   where **<Tomcat_install_dir>** is the path to your Tomcat installation directory, such as **C:\apache-tomcat-7.0.8**.

2. Insert additional configuration lines as illustrated by the examples below In both examples you should replace <agent_install_dir> and <probe_id> with values for your environment.

   The following is an example showing a modified **catalina.bat** file:

   ```
   :doStart

   rem Configuring Diagnostics Java Agent
   set AGENT_HOME=<agent_install_dir>
   set PROBE_ID=<probe_id>
   %_RUNJAVA% -jar %AGENT_HOME%\lib\jreinstrumenter.jar -f %PROBE_ID%
   set PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\%PROBE_ID%\instr.jre
   set PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
   set PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
   set CATALINA_OPTS=%CATALINA_OPTS% %PROBE_OPTS%
   ```

   The following is an example showing a modified **catalina.sh** file:

   ```
   # ----- Execute The Requested Command -----------------------

   # Configuring Diagnostics Java Agent
   AGENT_HOME=<agent_install_dir>
   PROBE_ID=<probe_id>
   "$_RUNJAVA" -jar $AGENT_HOME/lib/jreinstrumenter.jar -f $PROBE_ID
   PROBE_OPTS="-Xbootclasspath/p:$AGENT_HOME/classes/$PROBE_ID/instr.jre"
   PROBE_OPTS="$PROBE_OPTS -javaagent:$AGENT_HOME/lib/probeagent.jar"
   PROBE_OPTS="$PROBE_OPTS -Dprobe.id=$PROBE_ID"
   CATALINA_OPTS="$CATALINA_OPTS $PROBE_OPTS"
   ```

3. Save the changes to the startup script and restart the application server using the modified script.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

To configure a Tomcat server without a startup script:

Locate the Tomcat JVM configuration settings by right-clicking on the Apache Tomcat service icon from the Windows Task bar and then selecting **Configure**. Alternatively, you can navigate from the Start menu. For example, **Programs > Apache Tomcat 7.0 > Configure Tomcat**.

1. In the Apache Tomcat Properties dialog box, select the **Java** tab and find the Java Options box.
2. In the **Java Options** box, add the following JVM parameters, each on its own line, replacing <agent_install_dir> and <probe_id> with the actual values.

```
-javaagent:<agent_install_dir>\lib\probeagent.jar
-Xbootclasspath/p:<agent_install_dir>\classes\auto\<probe_id>\instr.jre
-Dprobe.id=<probe_id>
```

3. Restart the Tomcat service.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Note: If you update the JRE used by your application server in the future (such as applying an application server patch), you must delete the **<agent_install_dir>/classes/auto/<probe_id>** directory so that the new JRE will be instrumented. Otherwise, your application server may not start. For general information on the instrumentation mode used see "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.

Example 7: Configuring WebLogic Application Server for Monitoring

The following section provides general instructions with specific examples for the WebLogic application server for a generic implementation. Your site administrator should be able to use these instructions to show you how to make these changes in your customized environment.

To configure a WebLogic application server:

1. Locate the startup script used to start WebLogic for your domain and locate a convenient point in the file after all options are set but before the java command line (or code block) that starts the application server is executed.

   The startup script file is typically located in a path similar to the following:

   <DOMAIN_HOME>\bin\startWebLogic.[cmd|sh]

   where **<DOMAIN_HOME>** is the path to your domain directory, such as **C:\bea\user_projects\domains\<Domain_Name>**; or **C:\bea\wlserver_10.0\samples\domains\<Domain_Name>** where **<Domain_Name>** is the name of your domain.

   For example, if your domain name is **MedRec**, the path would look like the following:

   C:\bea\wlserver_10.0\samples\domains\medrec\bin\startWebLogic.cmd

2. Insert additional configuration lines as illustrated by the examples. In both examples you should replace <agent_install_dir> and <probe_id> with values for your environment.

   Below is an example showing the added lines in a **.cmd** file:

```
echo starting weblogic with Java version:

%JAVA_HOME%\bin\java %JAVA_VM% -version

set AGENT_HOME=<agent_install_dir>
set PROBE_ID=<probe_id>
%JAVA_HOME%\bin\java -jar %AGENT_HOME%\lib\jreinstrumenter.jar -f %PROBE_ID%
set PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\%PROBE_ID%\instr.jre
set PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
```

```
set JAVA_OPTIONS=%JAVA_OPTIONS% %PROBE_OPTS%

if "%WLS_REDIRECT_LOG%"=="" (
 echo Starting WLS with line:
 echo %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% …%JAVA_
HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% …
) else (
 echo Redirecting output from WLS window to %WLS_REDIRECT_LOG%
 %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% …
)
```

Below is an example showing the added lines in a **.sh** file:

```
echo "starting weblogic with Java version:"

${JAVA_HOME}/bin/java ${JAVA_VM} -version

# Configuring Diagnostics Java Agent
AGENT_HOME=<agent_install_dir>
PROBE_ID=<probe_id>
${JAVA_HOME}/bin/java -jar $AGENT_HOME/lib/jreinstrumenter.jar -f $PROBE_ID
PROBE_OPTS="-Xbootclasspath/p:$AGENT_HOME/classes/$PROBE_ID/instr.jre"
PROBE_OPTS="$PROBE_OPTS -javaagent:$AGENT_HOME/lib/probeagent.jar"
PROBE_OPTS="$PROBE_OPTS -Dprobe.id=$PROBE_ID"
JAVA_OPTIONS="$JAVA_OPTIONS $PROBE_OPTS"

if [ "${WLS_REDIRECT_LOG}" = "" ] ; then
 echo "Starting WLS with line:"
 echo "${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} …
 ${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} …
else
 echo "Redirecting output from WLS window to ${WLS_REDIRECT_LOG}"
 ${JAVA_HOME}/bin/java ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS} …
fi
```

Note: If your java command line does not use the JAVA_OPTIONS variable to define the JVM parameters, you need to change the variable name JAVA_OPTIONS shown in these examples to the correct name.

3. Save the changes to the startup script and restart the application server using the modified script.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Example 8: Configuring webMethods Server for Monitoring

There are two types of webMethods servers discussed in this example:

- webMethods Integration Server
- My webMethods Server

The following sections provide general instructions with specific examples for webMethods Integration Server and My webMethods Server. Your site administrator should be able to use these instructions to show you how to make these changes in your customized environment.

- "To configure a webMethods Integration Server started without the configuration wrapper:" below
- "To configure a webMethods Integration Server started with the configuration wrapper:" on the next page
- "To configure the My webMethods Server started without the configuration wrapper:" on the next page
- "To configure the My webMethods Server started with the configuration wrapper:" on page 52

To configure a webMethods Integration Server started without the configuration wrapper:

1. Locate the startup script used to start the webMethods Integration Server and locate a convenient point in the file after all options are set but before the java command line (or code block) that starts the application server is executed. There are two possible scripts based on how the server is started:

   <software_ag_home>\IntegrationServer\bin\server.bat

   <software_ag_home>\profiles\IS\bin\runtime.bat

2. Insert additional configuration lines as illustrated by these examples. In both examples you should replace <agent_install_dir> and <probe_id> with values for your environment.

   Below is an example showing the modified **server.bat** file:

```
if exist "%JAVA_DIR%\bin\jre.exe" (
set JAVA_EXEC="%JAVA_DIR%\bin\jre.exe"
set JAVA_CP="%JAVA_DIR%\lib\classes.zip;%JAVA_DIR%\lib\i18n.jar"
) else (
set JAVA_EXEC="%JAVA_DIR%\bin\java.exe"
set JAVA_CP="%JAVA_DIR%\lib\rt.jar;%JAVA_DIR%\lib\i18n.jar"
)

rem Configuring Diagnostics Java Agent

set AGENT_HOME=<agent_install_dir>
set PROBE_ID=<probe_id>
"%JAVA_EXEC%" -jar %AGENT_HOME%\lib\jreinstrumenter.jar -f %PROBE_ID%
set PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\%PROBE_ID%\instr.jre
set PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
set JAVA_OPTS=%JAVA_OPTS% %PROBE_OPTS%
```

   Below is an example showing the modified **runtime.bat** file:

```
rem Configuring Diagnostics Java Agent
set AGENT_HOME=<agent_install_dir>
set PROBE_ID=<probe_id>
"%JAVA_RUN%" -jar %AGENT_HOME%\lib\jreinstrumenter.jar -f %PROBE_ID%
set PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\%PROBE_ID%\instr.jre
set PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
set JAVA_OPTS=%JAVA_OPTS% %PROBE_OPTS%
```

```
%JAVA_RUN% -Xbootclasspath/a:"%OSGI_CLASSPATH%" %JAVA_OPTS% …
```

3. Save the changes to the startup script and restart the application server using the modified script.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

To configure a webMethods Integration Server started with the configuration wrapper:

Use this method if the application server is started as a service using **<software_ag_ home>\profiles\IS\bin\service.bat**.

1. Locate the webMethods Integration Server **custom_wrapper.conf** file. This file is typically located in:

   <software_ag_home>\profiles\IS\configuration\custom_wrapper.conf.

2. Insert additional configuration lines as illustrated by this example. In the example you should replace <agent_install_dir> and <probe_id> with values for your environment.

   Add the wrapper.java.additional entry near the other wrapper.java.additional parameters, changing number 777 as needed for your configuration.

   Below is an example showing the modified **custom_wrapper.conf** file:

```
# Put here your custom properties.

# Configuring Diagnostics Java Agent
set.AGENT_HOME=<agent_install_dir>
set.PROBE_ID=<probe_id>
set.PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\auto\%PROBE_ID%\instr.jre
set.PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set.PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
wrapper.java.additional.777=%PROBE_OPTS%
```

3. Save the changes to the configuration wrapper and restart the application server using the modified wrapper.

   See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Note: If you update the JRE used by your application server when started with the configuration wrapper in the future (such as applying an application server patch), you must delete the **<agent_install_ dir>/classes/auto/<probe_id>** directory so that the new JRE will be instrumented. Otherwise, your application server may not start. For general information on the instrumentation mode used see "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.

To configure the My webMethods Server started without the configuration wrapper:

Use this method if you start the application server by using the **run** command.

1. Locate the startup script used to start the My webMethods Server and locate a convenient point in the file after all options are set but before the java command line (or code block) that starts the application server is executed.

   The script file is: **<ag_software_home>\MWS\bin\mws.bat**.

2. Insert additional configuration lines as illustrated by this example. In the example you should replace <agent_install_dir> and <probe_id> with values for your environment.

The following is an example of the modified **mws.bat** file:

```
rem Configuring Diagnostics Java Agent
set AGENT_HOME=<agent_install_dir>
set PROBE_ID=<probe_id>
"%JAVA%" -jar %AGENT_HOME%\lib\jreinstrumenter.jar -f %PROBE_ID%
set PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\%PROBE_ID%\instr.jre
set PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
set JAVA_OPTIONS=%JAVA_OPTIONS% %PROBE_OPTS%

set JAVA_OPTIONS=%JAVA_OPTIONS% -Dserver.name=%SERVER_NAME% ...
set PARAMS=
set MAIN_CLASS=com.webmethods.portal.system.PortalSystem
set RUN_CMD=%JAVA% -cp %CLASSPATH% %JAVA_ARGS% %JAVA_OPTIONS% ...
```

3. Save the changes to the startup script and restart the application server using the modified script.

See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

To configure the My webMethods Server started with the configuration wrapper:

Use this method if you start the application server as a service or by using the **start** command.

> **Note:** This method requires customizations to the **wrapper.conf** file, which which may be overridden when the application server is upgraded or patched.

1. Locate the My webMethods Server **wrapper.conf** file. This file is typically located in:

   <ag_software_home>\MWS\server\<server_name>\config\wrapper.conf.

2. Insert additional configuration lines as illustrated by this example. In the example you should replace <agent_install_dir> and <probe_id> with values for your environment.

   Add the wrapper.java.additional entry near the other wrapper.java.additional parameters, changing number 777 as needed for your configuration.

   Below is an example showing the modified **wrapper.conf** file:

```
# Java Additional Parameters
…
# Configuring Diagnostics Java Agent
set.AGENT_HOME= <agent_install_dir>
set.PROBE_ID=<probe_id>
set.PROBE_OPTS=-Xbootclasspath/p:%AGENT_HOME%\classes\auto\%PROBE_ID%\instr.jre
set.PROBE_OPTS=%PROBE_OPTS% -javaagent:%AGENT_HOME%\lib\probeagent.jar
set.PROBE_OPTS=%PROBE_OPTS% -Dprobe.id=%PROBE_ID%
wrapper.java.additional.777=%PROBE_OPTS%
#NOTE: wrapper.java.additional.300 to 310 is reserved for debug configuration !
```

3. Save the changes to the configuration wrapper and restart the application server using the modified wrapper.

See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

> **Note:** If you update the JRE used by your application server when started with the configuration wrapper in the future (such as applying an application server patch), you must delete the **<agent_install_dir>/classes/auto/<probe_id>** directory so that the new JRE will be instrumented. Otherwise, your application server may not start. For general information on the instrumentation mode used see "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.

Example 9: Configuring WebSphere Application Server for Monitoring

> **Note:** If you have auto-deployed the application server during the Java agent setup, further configuration is unnecessary.

The following section provides general instructions with specific examples for the WebSphere application server for a generic implementation. Your site administrator should be able to use these instructions to show you how to make these changes in your customized environment.

Procedures are provided for WebSphere 7.0 or higher.

> **Note:** Extra steps are required to enable metric collections in WebSphere. See "Configuring WebSphere for JMX Metric Collection" on page 55.

To configure WebSphere 7.0 or higher

1. Locate the application JVM configuration settings by logging in to the WebSphere Application Server Administrative Console. For example:

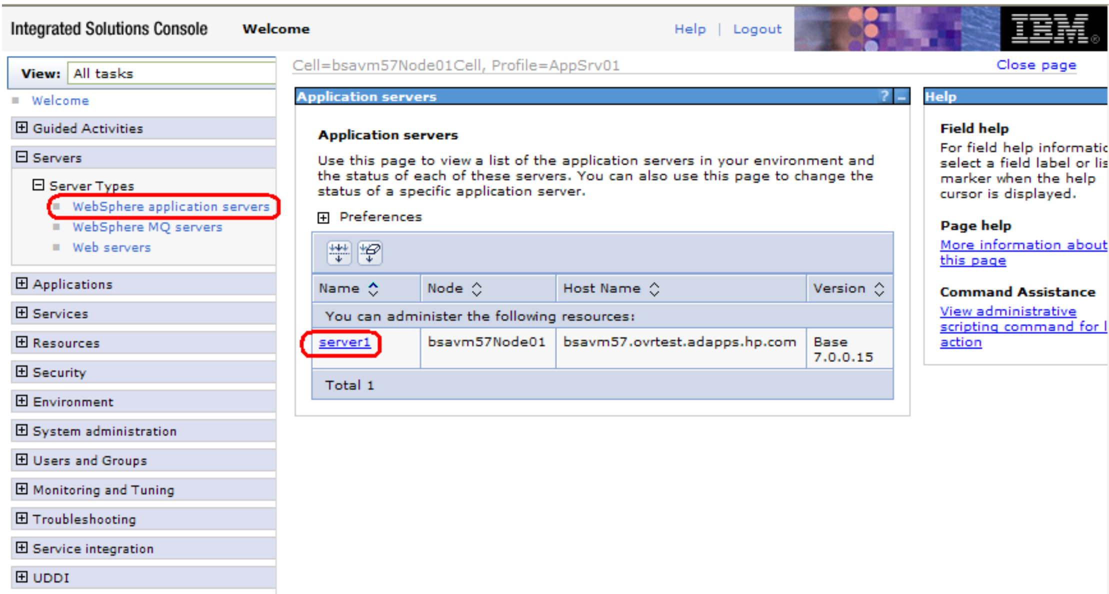
   ```
   http://<App_Server_Host>:9060/ibm/console
   ```

   Replace **<App_Server_Host>** with the machine name for the application server host and 9060 with the correct administrative port number (such as 9060, 9061, and so on).

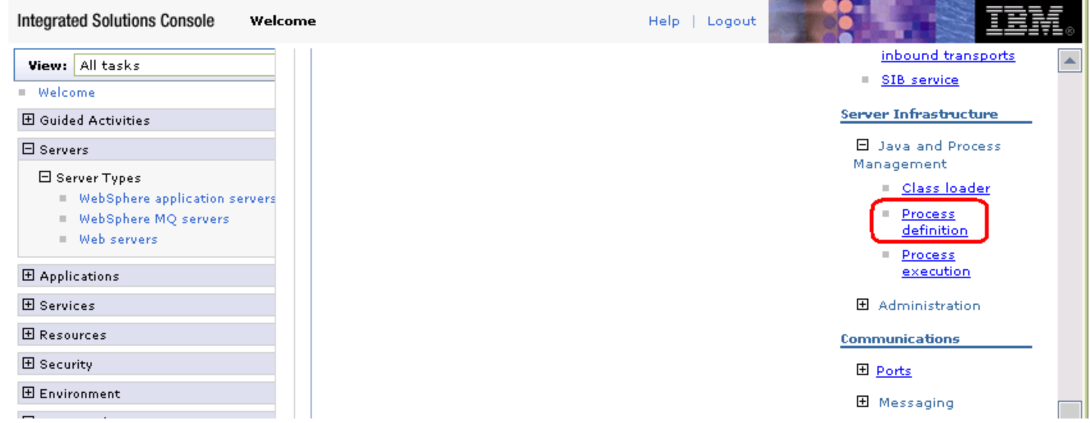
   Navigate to the Java Virtual Machine page. For example:

   Navigate to: **Servers > Server Types > WebSphere Application servers**

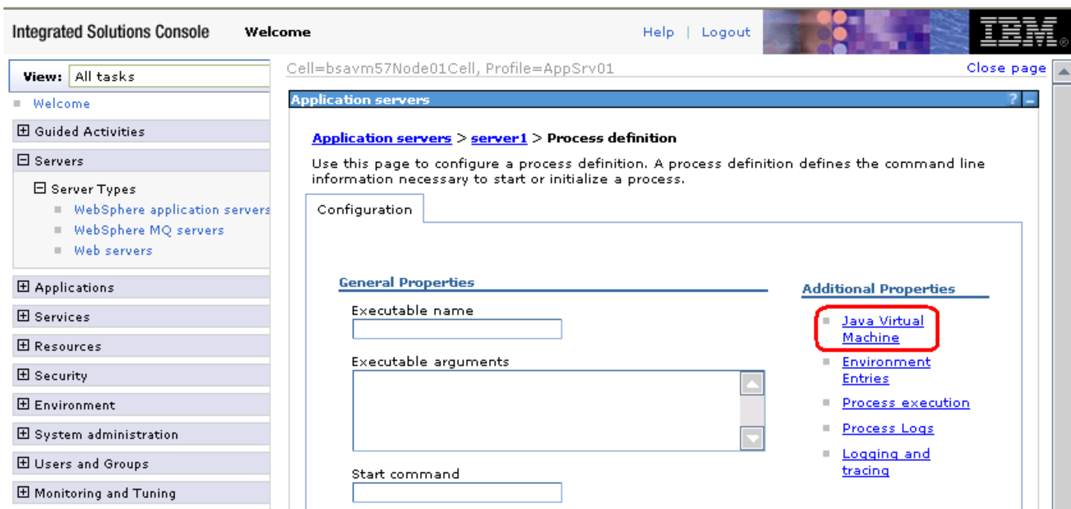
   Then click the application server instance name (such as **server1**).



Then, under **Server Infrastructure > Java and Process Management**, click **Process Definition**.



Then, under **Additional Properties**, click **Java Virtual Machine**.



2. On the Java Virtual Machine page, in the **Generic JVM Arguments** box, enter the JVM parameters.

   - For WebSphere running on IBM JRE 1.6 (WebSphere 7.0 or 8.0/ 8.5 default) enter the following JVM parameters. In the example replace <agent_install_dir> and <probe_id> with values for your environment.

     ```
     -Xbootclasspath/p:<agent_install_dir>\classes\auto\<probe_id>\instr.jre
     -javaagent:<agent_install_dir>\lib\probeagent.jar
     -Xshareclasses:none
     -Dprobe.id=<probe_id>
     ```

   - For WebSphere running on IBM JRE 1.7 (configurable on WebSphere 8.0 or higher), you must run the Diagnostics JRE Instrumenter manually and then insert the JVM parameters returned by the JRE instrumenter. See "Using the JRE Instrumenter in Manual Mode " on page 62.

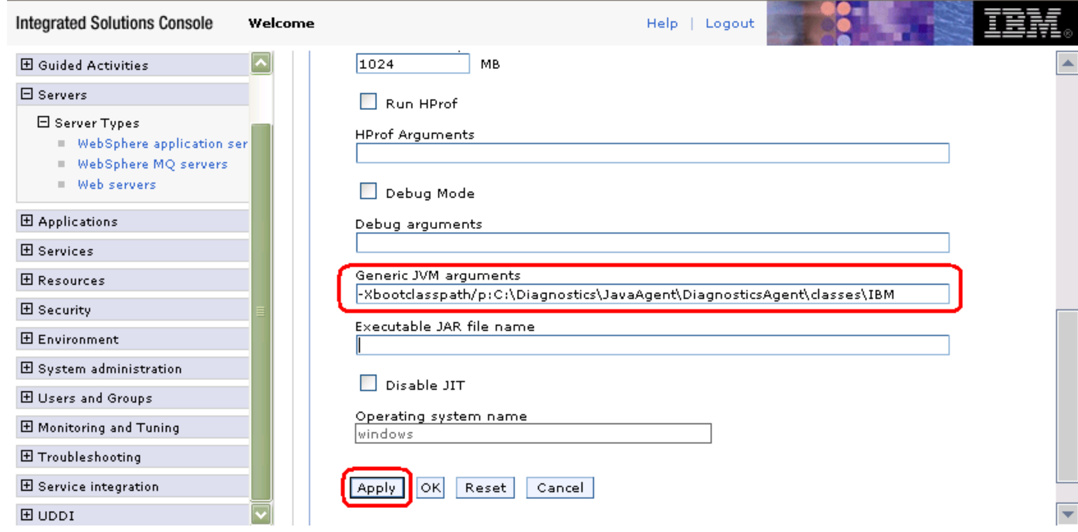
     In addition to specifying the JVM parameters returned by JRE Instrumenter, include **-Dprobe.id=<probe_id>**.

     If you have enabled SSL communication for the probe, also add the following line:

     -Djava.security.properties=<agent_install_dir>/etc/ibm.default.java.security

> **Caution:** Using the wrong JVM parameters for your version of WebSphere can cause extreme performance degradation of the monitored application. There are two categories of WebSphere application servers listed above, each with its own method of required instrumentation.

3. Apply and save your changes, and restart the application server.



See "Verify the Application Server is Running the Java Agent" on page 58 for more information.

Configuring WebSphere for JMX Metric Collection

You might need to configure the Performance Monitoring Infrastructure (PMI) service on the WebSphere server to start receiving JMX metrics.

> **Note:** If Diagnostics is not able to identify your application server as a WebSphere server, you must

enable PMI and add the Jar files to the server.policy file.

To configure WebSphere server for JMX metrics collection:

1. Open the WebSphere Administrative Console.
2. In the Console navigation tree, select **Servers > Application Servers**. The console displays a table of the application servers.
3. Click the name of the application server you want to configure from the Application Servers Table. The console displays the **Runtime** and the **Configuration** tabs for the selected application server.
4. Click the **Configuration** tab.
5. In the **Configuration** tab:
   - Under the Performance heading, click **Performance Monitoring Infrastructure (PMI)**.
   - Under the General Properties heading, select the **Enable Performance Monitoring Infrastructure (PMI)** check box.
   - Under the Currently monitored statistic set heading select **Extended**.
6. Click **Apply** or **OK**.
7. If Java 2 Security is enabled on the application server, open the server policy file **<WebSphere Installation Directory>/profiles/<your_profile_name>/properties/server.policy)** and add the following security permissions to enable JMX collection. Replace <agent_install_dir> with the value for your environment.

   ```
   grant codeBase "file:/<agent_install_dir>/lib/probe-jmx.jar"
   { permission java.security.AllPermission; }
   grant codeBase "file:/<agent_install_dir>/lib/probe-jmx-was6.jar" {
   permission java.security.AllPermission;
   };
   ```

8. Restart the application server.

Example 10: Configuration for WebSphere Application Server Liberty

If your application is based on the Liberty Profile of WebSphere Application Server, you need to configure Diagnostic Java Agent on Linux or AIX, using one of the following methods:

- **Automatic Implicit Mode**
- **Automatic Explicit Mode**

We recommend Automatic Explicit Mode if you encounter any stability issues with the JVM running the Liberty Application Server.

Note: You cannot use both methods simultaneously. For example, if you are using Explicit Mode, you cannot include a jvm.options file configured for Implicit Mode.

To Configure Diagnostics Java Agent in Automatic Implicit Mode on Linux or AIX:

1. In the directory **<application_server>/usr/servers/<server_name>** (for example, WebSphere_8.5/Liberty/usr/servers/defaultServer) create a text file named **jvm.options** with the following content:

   ```
   -javaagent:<agent_install_dir>/lib/probeagent.jar
   -Xbootclasspath/p:<agent_install_dir>/classes/auto/<probe_id>/instr.jre
   -Dprobe.id=<probe_id>
   -Xshareclasses:none
   ```

   where **<agent_install_dir>** is the absolute path to the installation directory and **<probe_id>** is the value of the probe ID.

   If you have enabled SSL communication for the probe, also add the following line:

   -Djava.security.properties=<agent_install_dir>/etc/ibm.default.java.security

2. In the same directory create another text file named **bootstrap.properties** with the following content:

   ```
   org.osgi.framework.bootdelegation=com.mercury.opal.capture.*
   ```

   Note: If either of the above files already exist, add the content above to the existing file.

To Configure Diagnostics Java Agent in Automatic Explicit Mode on Linux or AIX:

1. Open the file **<application_server>/bin/server** and locate the following lines:

   ```
   ##
   ## serverEnvAndJVMOptions: Read server.env files and set environment variables.
   ##                         Read jvm.options file into ${JVM_OPTIONS_QUOTED}
   serverEnvAndJVMOptions()
   {
    serverEnv

    # Avoids HeadlessException on all platforms and Liberty JVMs appearing as
   applications and stealing focus on Mac.
    JVM_OPTIONS_QUOTED=-Djava.awt.headless=true
    SERVER_JVM_OPTIONS_QUOTED=${JVM_OPTIONS_QUOTED}

    # Add -XX:MaxPermSize unless WLP_SKIP_MAXPERMSIZE is set.
    if [ -z "${WLP_SKIP_MAXPERMSIZE}" ]; then
    SERVER_JVM_OPTIONS_QUOTED="${SERVER_JVM_OPTIONS_QUOTED} -XX:MaxPermSize=256m"
   ```

2. Add the following lines below the lines listed above:

   ```
   # Configuring Micro Focus Diagnostics Java Agent
   AGENT_HOME=<agent_install_dir>
   PROBE_ID=<probe_id>
   $JAVA_HOME -jar $AGENT_HOME/lib/jreinstrumenter.jar -f $PROBE_ID
   PROBE_OPTS="-Xbootclasspath/p:$AGENT_HOME/classes/$PROBE_ID/instr.jre"
   PROBE_OPTS="$PROBE_OPTS -javaagent:$AGENT_HOME/lib/probeagent.jar"
   PROBE_OPTS="$PROBE_OPTS -Dprobe.id=$PROBE_ID"
   PROBE_OPTS="$PROBE_OPTS -Xshareclasses:none"
   SERVER_JVM_OPTIONS_QUOTED="${SERVER_JVM_OPTIONS_QUOTED} ${PROBE_OPTIONS}"
   ```

3. In the directory **<application_server>/usr/servers/<server_name>** create a text file named **bootstrap.properties** with the following content:

   ```
   org.osgi.framework.bootdelegation=com.mercury.opal.capture.*
   ```

4. If the directory **<application_server>/usr/servers/<server_name>** contains a file **jvm.options**, make sure that the file does not contain the Diagnostics specific entries as described in the Automatic Implicit Mode section above.

Verify the Application Server is Running the Java Agent

You verify the agent is monitoring the application server after you restart the application server to pick up the changes to the startup script.

For an agent in Diagnostics Profiler Mode:

Start the Profiler UI and view the probe. See "How to Access the Java Diagnostics Profiler" on page 227.

For an agent reporting to an on-premise Diagnostics Server:

1. In your browser, navigate to `http://<diagnostics_server_host>:2006`, or open **Administration** from the computer's Start menu.

   Port number 2006 is the default port for the Diagnostics Commander Server. If the Diagnostics Server was installed and configured to use an alternate port, specify that port number in the URL.
2. Log in. Obtain the login credentials from the Diagnostics Administrator. The default user/password of **admin/admin** may work.
3. In the navigation pane of the Diagnostics Applications window, double-click **Entire Enterprise**. The Diagnostics views open.
4. Open the Application Servers view group, and select **Java Probes**.

You can also check the System Health view to find information about the Java agent deployments and the machines that host them. See "System Views" in the Diagnostics User Guide.

For an agent reporting to a Saas-hosted Diagnostics Server:

Contact your SaaS system administrator.

Troubleshooting:

- You can also check for entries in the **<agent_install_directory>\log\<probe_id>\probe.log** file. If there are no entries in the file, you did not instrument the JRE or did not enter the Java parameter such as **Xbootclasspath** correctly. In the **probe.log** file look for errors and look for an entry that says "Successfully downloaded first command" which indicates that the communication between the probe and the server has been established.
- To verify that the Java Agent is connected to the Diagnostics Server, direct your browser to the host running the application, using port 35000. For example:

  `http://agentsystem.mycompany.com:35000`

  A page showing the probe status at the bottom is displayed:

About the JRE Instrumenter and Different Options to Invoke

The JRE Instrumenter is a utility to instrument a JRE so that the Java Agent can provide advanced features such as the patent-pending Collection Leak Pinpointing (CLP). It does not modify the installed JRE in any

way, but rather places copies of instrumented classes somewhere under the **<JavaAgent_install_dir>/DiagnosticsAgent/classes** directory. You can use the JRE Instrumenter to instrument multiple JREs if they are installed on your system.

The JRE Instrumenter instruments some standard Java classes used by the application server JVM and applications running on it. It also provides you with the JVM parameters that must be used when the application server is started so that the application server uses the instrumented classes.

With different command-line options, the JRE Instrumenter can be invoked and used in three different ways, each of which has its own advantages and limitations. You will use one of these methods according to the characteristics of your application servers (see "Examples for Configuring Application Servers " on page 33 for examples).

- **Automatic Explicit Mode.** If your application server is or can be started by a script, it is recommended that you use this mode. To use this mode, you add a line to your application server startup script to explicitly and non-interactively run the JRE Instrumenter to instrument the JRE. Your script will continue to start the application server JVM (with additional parameters) using the freshly instrumented JRE. See "Using the JRE Instrumenter in Automatic Explicit Mode" on the next page.
- **Automatic Implicit Mode.** With this mode, you do not need to explicitly run the JRE Instrumenter — you only need to modify your application server JVM parameters to invoke the Java Agent and ask it to run the JRE Instrumenter as needed. When the Java Agent is used for the first time, it implicitly runs the JRE Instrumenter to instrument the JRE. However, the first time this instrumented JRE will not be used; your application server will be using an uninstrumented JRE. The next time your application server is started, it will use the instrumented JRE. Therefore, if you want to use the full monitoring features of the Java Agent, you need to restart your application server twice after you enable the Java Agent. See "Using the JRE Instrumenter in Automatic Implicit Mode" on page 61.
- **Manual Mode.** With this mode, you need to manually and interactively run the JRE Instrumenter, either at the end of the Java Agent installation or at a later time, to instrument the JRE. You then modify your application server JVM parameters according to the parameters provided by the JRE Instrumenter. This method is how the JRE Instrumenter works in earlier versions of Diagnostics. See "Using the JRE Instrumenter in Manual Mode " on page 62.

If your JRE is updated (such as, applying an application server patch) or if you update the Java Agent, you may need to instrument the JRE again. This issue will be discussed in each mode.

Below is a table that summarizes the requirements of each of the four different methods of doing instrumentation:

		Recommended Instrumentation (Using the JRE Instrumenter)		
	Basic Instrumen-tation	**In Automatic Explicit Mode**	**In Automatic Implicit Mode**	**In Manual Mode**
Minimum required JRE version	1.6	1.6	1.6	1.6
Requires the application server being started by a script	No	Yes	No	No
Requires knowing where the JRE is installed	No	No	No	Yes
Requires manually running the JRE Instrumenter	No	No	No	Yes

Requires knowing where the JVM parameters can be configured	Yes*	Yes*	Yes*	Yes*
Requires restarting the application server after enabling Java Agent	Yes, once	Yes, once or twice	Yes, twice	Yes, once
Requires maintenance after JRE upgrade/patch	No	No	Yes	Yes

* If you cannot find where the JRE invocation parameters can be defined, you may still have the option of using an environment variable such as **JAVA_OPTIONS** to do that.

Using the JRE Instrumenter in Automatic Explicit Mode

Using the JRE Instrumenter in the Automatic Explicit Mode is recommended when an application server is started by a script, such as WebLogic and JBoss application servers. It is also recommended for WebSphere application servers if they are, or can be, started by a script - this is the case for most platforms. It is also recommended for Tomcat if it is not installed as a Windows service (when Tomcat as a Windows service has been auto-deployed, the JRE Instrumenter runs in Automatic Explicit Mode by default).

To use Automatic Explicit mode, you need to accomplish two tasks:

- Modify your application server startup script to run the JRE Instrumenter using the same JRE used by your application server. The output from the JRE Instrumenter will give you the JVM parameters you will need in the next task.
- Configure your application server JVM parameters found in the output from the JRE Instrumenter.

Note: Make sure you understand the structure of the startup script, how the property values are set, and how to use environment variables before you make any configuration changes. Always create a backup copy of any file you plan to modify before making the changes.

In modifying the application server startup script, you first need to identify the line (or lines) in which the JRE is invoked to start the application server JVM. Then, right above this line, you add a line like the following to invoke the JRE Instrumenter using the same JRE used by your application server:

```
<java_command> -jar <agent_install_directory>/lib/jreinstrumenter.jar -f <pathname>
```

The **<java_command>** must be **exactly** the same java command that is used to start your application server JVM, since it is the JRE that is instrumented by the JRE Instrumenter. You can usually get this java command by copying the beginning portion of the line that starts your application server JVM.

Below is a table showing the java command used by the original startup script of some commonly used application servers. (Note that this table is provided as helpful tips only; your application server startup script may use a different java command.)

Application Server	Shell Scripts (.sh)	Windows Command Scripts (.bat or .cmd)
JBoss	"$JAVA"	"%JAVA%"
Tomcat	${_RUNJAVA}	%_RUNJAVA%.
WebLogic	${JAVA_HOME}/bin/java	%JAVA_HOME%\bin\java
WebSphere	${JAVA_EXE}	%JAVA_EXE%

The **<agent_install_directory>** indicates the directory where the Java Agent is installed.

The **<pathname>** must be relative. The JRE Instrumenter will put the instrumented classes in the **<agent_install_directory>/classes/<pathname>/instr.jre** directory. If you run multiple application servers with Diagnostics, you should give each application server a unique <pathname> (such as the probe name) so that the multiple instances of the JRE Instrumenter do not interfere each other. See also "Configure Monitoring of Multiple Java Processes on an Application Server" on page 66 for details.

After you add the line as described above to the startup script, every time you start your application server using the startup script, the JRE Instrumenter is invoked and instruments the current JRE. It also prints out the JVM parameters that you should use in the next task. You can usually find the output of the JRE Instrumenter among the output from running the startup script.

Below is an example output from the JRE Instrumenter that instruments a typical JRE:

```
-Xbootclasspath/p:<agent_install_directory>/classes/<pathname>/instr.jre

-javaagent:<agent_install_directory>/lib/probeagent.jar
```

The second task for using the Automatic Explicit JRE instrumentation is to modify your application server JVM parameters according to the output of the JRE Instrumenter. In many cases, you just need to modify the java command-line options in the startup script to include the JVM parameters provided by the JRE Instrumenter. However, in some scenarios (such as for WebSphere application servers), you may need to modify a configuration file or use an administration console to add these JVM parameters.

Note: To get the output from the JRE Instrumenter, you need to modify the startup script as described in the first task and restart the application server. Then, after you make changes to the application server JVM parameters, you need to restart the application server again (causing you to restart the application server twice). However, for most of the JREs, the actual JVM parameters provided by the JRE Instrumenter will be the same as or will include what is provided in the examples above. Therefore, you can safely add these "default" JVM parameters even before you run the modified script. This approach is used in the instructions for specific application servers. Refer to the example for your application server (JBoss, WebLogic, WebSphere, Tomcat) to see detailed instructions for how to configure using automatic explicit mode.

Alternatively, you can redirect (or pipe) the output from the JRE Instrumenter to the java command-line options, or get the JVM parameters from a difference source to avoid restarting twice.

Using the JRE Instrumenter in Automatic Implicit Mode

Using the JRE Instrumenter in the Automatic Implicit Mode is recommended when an application server cannot be started by a script, such as GlassFish, NetWeaver, Tomcat installed as a Windows service (and not auto-deployed during setup), and TIBCO ActiveMatrix and BusinessWorks.

To use this mode, you do not need to explicitly invoke the JRE Instrumenter; it is implicitly invoked by the Java Agent. You just configure your application server JVM parameters to invoke the Java Agent and, when the Java Agent sees that the JVM boot class path contains a path pointing to a location matching the following pattern, it enters the automatic instrumentation mode to create the instrumented classes and populates the specified directories with copies of the instrumented classes:

```
<agent_install_directory>/classes/auto/<name>/instr.jre
```

For example if you add the following JVM parameters

```
-Xbootclasspath/p:<JavaAgent_install_
dir>/DiagnosticsAgent/classes/auto/ServerOne/instr.jre
-javaagent: <agent_install_directory>/lib/probeagent.jar
```

Then during the first execution of the application server, the directory **<agent_install_directory>/classes/auto/ServerOne/instr.jre** may not even exist. The Java Agent will create and populate the specified directory with the instrumented classes. And it will use the exact (uninstrumented) JRE that it runs on.

The first execution of the application server will not benefit from the instrumented JRE, but all subsequent executions will use the instrumented classes prepared in the first run.

Note: If you update the JRE used by your application server (such as applying an application server patch) or if you update the Java Agent, before you start the application server again you must delete the **<agent_install_directory>/classes/auto/ServerOne** directory (use your directory name for ServerOne) so that the new JRE will be instrumented. Otherwise, your application server may not start. You can also manually delete this directory when you want the Java Agent to instrument the JRE again.

Using the JRE Instrumenter in Manual Mode

You can manually run the JRE Instrumenter and copy the provided JVM parameters into your application server startup settings. Using the JRE Instrumenter in the Manual Mode is recommended for Oracle application servers.

The JRE Instrumenter performs the following functions:

- Identifies JREs that are available to be instrumented.
- Searches for additional JREs in directories you specify.
- Instruments the JREs you specify and provides the parameter you must add to the startup script for the JRE to point to the location of the instrumented classes.
- When the Instrumenter is run using the graphical interface or console mode in a Windows or UNIX environment, the Instrumenter places the instrumented classes in a folder under the **<agent_install_directory>/classes/<JRE_vendor>/<JRE_version>** directory.

Note: If you update the JRE used by your application server (such as applying an application server patch) or if you update the Java Agent, you must run the JRE Instrumenter again to instrument the new JRE and change the JVM parameters accordingly. Otherwise, your application server may not start.

Running the JRE Instrumenter Utility in UI Mode

When the JRE Instrumenter is run without any options the Instrumenter displays the dialogs of its graphical user interface.

To start the JRE Instrumenter utility on a Windows system run the **<agent_install_directory>\bin\jreinstrumenter.cmd** command.

To start the JRE Instrumenter utility on UNIX or Linux run the **<agent_install_directory>\bin\jreinstrumenter.sh** command.

The Instrumenter lists the JVMs that were discovered by the Instrumenter and are available for instrumentation. The JVMs that were instrumented are listed with a green square preceding the name of the JVM.

If the **JRE Directory** is not listed on the dialog, click the **Add JRE(s)** button to browse to the JRE. Navigate to the directory location where you want to begin searching for JVMs and then select the file where you want to begin the search and click Search from here. The Instrumenter searches and then lists the JVMs found in the Available JREs list.

Select the JRE to be instrumented and then click **Instrument**.

The JRE Instrumenter instruments some of the classes for the selected JVM and places the instrumented classes in a folder under the **<agent_install_directory>/classes** directory. It also displays the JVM parameter that must be used when the application server is started in the box below the Available JREs list.

When the JRE Instrumenter instruments a JRE, it also creates the JVM parameters you must include in the startup script for the application server to cause your application to use the instrumented classes. When you select an instrumented JRE from the Available JREs list, the JVM parameters are displayed in the box below the list.

Click **Copy Parameter** to place the corresponding parameter on the clipboard. The JVM parameter is copied to the clipboard so that you can use the JVM parameters in configuring your application server to activate monitoring by the Java Agent.

> **Note:** You will use the clipboard contents later in configuring you application server, so be careful to not overwrite the clipboard contents.

Click **Exit** to close the JRE Instrumenter window and continue with configuring your application server JVM parameters.

For general instructions for how to insert the JVM parameter into application server startup scripts see "Specifying Probe Properties as Java System Properties" on page 31. For specific examples of how to insert the JVM parameter into startup scripts for different application servers such as JBoss, WebLogic and Tomcat see "Examples for Configuring Application Servers " on page 33.

Running the JRE Instrumenter in Console Mode

Open **<agent_install_directory>\bin** to locate the JRE Instrumenter executable. Run the following command:

```
./jreinstrumenter.sh -console
```

When the Instrumenter runs, it displays a list of the processing options that are available. The following table directs you to the documentation for each of the processing options:

Instrumenter Function	Description
jreinstrumenter -l	Display a list of the JVMs that are known to the JRE Instrumenter. Displays the JVM vendor, JRE version, and the location where the JRE is located.

Instrumenter Function	Description
jreinstrumenter -i <jre_directory>	Select a JRE in a specific directory for instrumentation. Replace <jre_directory> with the path to the folder where the JRE you selected from the Available JVM list is found. This command instructs the JRE Instrumenter to instrument the classes for the selected JVM and to place the instrumented classes in a folder under the **<agent_install_directory>/classes/<JVM_vendor>/<JRE_version>** directory.
jreinstrumenter -a <directory>	Search for JVMs within a specific directory and add any JVMs that are found to the list of the JVMs known to the JRE Instrumenter. Replace <directory> with the path to the location where you would like the Instrumenter to begin searching. The Instrumenter searches the directories from the location specified including the directories and subdirectories. When it completes its search, it displays the updated list of Available JVMs.

Copy the JVM parameter from the output of the JRE Instrumenter so that you can paste it into the location that allows it to be picked up when your application server starts in order to activate monitoring by the Java Agent.

Exit the JRE Instrumenter and continue with configuring your application server JVM parameters.

For General instructions for how to insert the JVM parameter into application server startup scripts see "Specifying Probe Properties as Java System Properties" on page 31. For specific examples of how to insert the JVM parameter into startup scripts for different application servers such as JBoss, WebLogic and Tomcat see "Examples for Configuring Application Servers " on page 33.

Including the JVM Parameter in the Application Server's Startup Script

When the JRE Instrumenter instruments a JVM, it also creates the JVM parameter you must include in the startup script for the application server in order to cause your application to use the instrumented classes. When the Instrumenter finishes instrumenting the JVM, it displays the JVM parameter.

Copy the JVM parameter to the clipboard and paste it into the location that allows it to be picked up when your application server starts. General instructions are provided below.

See "Examples for Configuring Application Servers " on page 33 for specific examples of how to insert the JVM parameter for application servers such as WebLogic, WebSphere, JBoss and others.

To update the application server configuration:

1. Locate the application server startup script or the file where the JVM parameters are set.
2. Create a backup copy of the application server startup script before you make any changes to the script.
3. Use an editor or the application server console to open the startup script.
4. Add the Java parameter from the JRE Instrumenter to the java command line that starts the application server, for example:

   ```
   -Xbootclasspath/p:<agent_install_dir>\classes\Sun\1.5.0\instr.jre;
   <agent_install_directory>\classes\boot
   ```

In this instance, **<agent_install_directory>** is the path to the directory where the Java Agent was installed.

This connects the probe to the application.

The following is an example of a WebLogic java command line in a startup script before adding the Java parameter:

```
"%JAVA_HOME%\bin\java" -hotspot -ms64m -mx64m -classpath "%CLASSPATH%"
-Dweblogic.Domain=petstore -Dweblogic.Name=petstoreServer -Dbea.home="C:\\bea"
-Dweblogic.management.password=%WLS_PW%
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Dcloudscape.system.home=./samples/eval/cloudscape/data
-Djava.security.policy="C:\bea\wlserver6.1/lib/weblogic.policy" weblogic.Server
```

The following is an example of a WebLogic java command line in a startup script after adding the Java parameter:

```
"%JAVA_HOME%\bin\java" -hotspot -ms64m -mx64m
-Xbootclasspath/p:<agent_install_directory>\classes\Sun\1.5.0_17\instr.jre;
-javaagent:<agent_install_directory>\lib\probeagent.jar
-classpath "%CLASSPATH%"
-Dweblogic.Domain=petstore -Dweblogic.Name=petstoreServer
-Dbea.home="C:\\bea" -Dweblogic.management.password=%WLS_PW%
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Dcloudscape.system.home=./samples/eval/cloudscape/data
-Djava.security.policy="C:\bea\wlserver6.1/lib/weblogic.policy" weblogic.Server
```

5. Save the changes to the startup script.
6. Restart the application server under test.
7. To verify that the probe was configured correctly, check for entries in the **<agent_install_directory>\log\<probe_id>\probe.log** file. If there are no entries in the file, you did not instrument the JRE used by the application server or did not configure your application server JVM parameters to invoke the Java Agent (see the instructions in this chapter for your application server).

Other Configuration Options

The following sections give you other configuration options:

- "Probe Registration Auto-Assigment" below
- "Configure Monitoring of Multiple Java Processes on an Application Server" on the next page
- "Adjusting the Heap Size for the Java Agent in the Application Server" on page 69
- "Configuring the SOAP Message Handler" on page 69
- "Configuring the Discovery of a New J2EE Server for CI Population" on page 71
- "Special Considerations for Applications Based on the OSGi Framework" on page 72

Probe Registration Auto-Assigment

A typical use of probe registration auto-assignment is when you have multiple agents sharing a single installation. Probe auto-assignment is configured using the following properties in the **<Agent host machine>/etc/dispatcher.properties** file:

- **commander.registrar.url** - The Commander Registrar URL for Probe to Mediator Auto-Assignment.
- **registrar.url** - This property should be set to blank initially, and should not be manually modified if you want to use auto-assignment.
- **always.use.commander.registrar.url** - By default, the auto-assigned mediator will be recorded within this file by overwriting the **registrar.url** property.
- **force.auto.assign** – If this property is set to **True**, if the probe has a previously assigned mediator and the load on the mediator is over the maximum load limit, the commander will not assign that mediator, but instead will return a new, mostly-filled mediator.

  If this property is set to **False**, the system does not check whether the mediator is over the load limit. The default value is **False**.

For further details on these properties and how they relate to auto-assignment, see "Probe Registration Auto-Assigment for Large Deployments" in the Diagnostics Server Installation and Administration Guide.

For details on how to configure a single Java Agent to be shared by multiple JVMs, see "Configure Monitoring of Multiple Java Processes on an Application Server" below.

Configure Monitoring of Multiple Java Processes on an Application Server

When you want to collect performance data for multiple Java processes on a host, you have two options:

- You can configure a separate Java Agent installation for each process (JVM) on a host.
- You can configure a single Java Agent to be shared by all of the processes (JVMs).

This section describes how to configure a single Java Agent installation to be shared by multiple JVMs.

To configure a separate Java Agent installation for each process, simply ensure that each <agent_install_directory> is uniquely named.

Configure a Single Java Agent to be Shared by Multiple JVMs

To allow multiple JVMs to share a single Java Agent installation, you must configure a separate probe for each JVM as described below. This ensures a unique name and port for each probe. Optionally each probe can have its own points file and mediator assignment.

To configure a single Java Agent installation to be shared by multiple JVMs:

1. Determine how the JRE will be instrumented for all the Java applications that you plan to monitor. See "Preparing Application Servers for Monitoring with the Java Agent" on page 30.

   Multiple JREs may exist. Each can have their own instrumentation method.

2. Specify the range of ports from which the probe can automatically select. The Java Agent communicates using the Java Agent listening port. A separate port is assigned for communications for each JVM that a probe is monitoring. By default, the port number range (Min/Max) is set to **35000**–**35100**. You must increase the port number range when the probe is working with more than 100 JVMs.

   If a firewall separates the probe from the other Diagnostics components, configure the firewall to allow communications using the ports in the range you specify. For more information, see the chapter “Configuring Diagnostics to Work in a Firewall Environment" in the Diagnostics Server Installation and Administration Guide.

   If you configure the firewall to allow probe communications on a range of ports that is different than the default, update the port range values as follows.

a. Locate the **webserver.properties** file in the folder **<agent_install_directory>/etc**.

b. Set the following properties to adjust the range of ports available for probe communications.

   The minimum port in the port number range uses the following property:

```
jetty.port=35000
```

   The maximum port in the port number range uses the following property:

```
jetty.max.port.offset=100
```

3. Assign a unique probe name using one of the following methods.

   By default, the probe id is set to the value specified during the Java Agent Setup. This is set in **probe.properties** as the **id** property. The probe id needs to be unique for each probe on the same host instead of sharing the id set in **probe.properties**.

   The command line properties must be entered on one line, without any line breaks. The probe ids defined on the Java command line override the probe names defined in the **probe.properties** file using the probe's **id** property.

   a. Assign a probe Id to the probe for each JVM, using the Java command line or by editing the application startup script.

```
-Dprobe.id=<Unique_Probe_Name>
```

   The following example shows a WebLogic startup script before reconfigured to run with Diagnostics:

```
"%JAVA_HOME%\bin\java" -hotspot -ms64m -mx64m -classpath "%CLASSPATH%"
-Dweblogic.Domain=petstore -Dweblogic.Name=petstoreServer -Dbea.home="C:\\bea"
-Dweblogic.management.password=%WLS_PW%
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Dcloudscape.system.home=./samples/eval/cloudscape/data
-Djava.security.policy=="C:\bea\wlserver6.1/lib/weblogic.policy"
weblogic.Server
```

   The following example shows a WebLogic startup script after adding the **probe.id** parameter:

```
"%JAVA_HOME%\bin\java" -hotspot -ms64m -mx64m"
-Xbootclasspath/p:C:\MercuryDiagnostics\JAVAProbe\classes\Sun\1.6.0_
24\instr.jre;C:\MercuryDiagnostics\JAVAProbe\classes\boot"
-classpath "%CLASSPATH%"
-Dprobe.id=<Unique_Probe_Name> -Dweblogic.Domain=petstore
-Dweblogic.Name=petstoreServer
-Dbea.home="C:\\bea" -Dweblogic.management.password=%WLS_PW%
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Dcloudscape.system.home=./samples/eval/cloudscape/data
-Djava.security.policy=="C:\bea\wlserver6.1/lib/weblogic.policy"
weblogic.Server
```

   b. When a single Java parameter is specified but multiple probes are started using the same script, use the %0 string to generate a custom probe identifier for each probe—for example, in a clustered environment where a single startup script is used to start multiple probed application server instances.

On Linux:

```
-Dprobe.id=<probeName>%0
```

On Windows:

```
-Dprobe.id=<probeName>%%0
```

Use the first % to escape the second %.

The **%0** is replaced dynamically with a number to create a unique probe name for each probe; for example, <probeName>0, <probeName>1, and so on.

4. (Optional) Specify the points file each probe will use. By default, the points file name is **auto_detect.points**. You can specify that a custom points file be used when you must use more than one custom instrumentation plan, or where you have several JRE versions on the same machine using a single agent installation, and one or more of the JREs needs specific methods and classes included in a layer to support custom instrumentation.

   ```
   -Dprobe.points.file.name="<Custom_AutoDetect_Points_File>"
   ```

   where <Custom_AutoDetect_Points_File> is the name of your custom points file such as **MyProbe1_private.points**.

5. (Optional). Specify the mediator to which each probe will send its collected data. You can designate a specific mediator or enable auto-assignment to mediators. By default, the mediator that was specified at installation time is used. You can override that setting for any probe.

   - To designate a specific mediator assignment for the probe, add the following to the application server startup script or command line:

     ```
     -Ddispatcher.registrar.url=http://<mediator_host>:2006/registrar/
     ```

     where <mediator_host> is the host name or IP of the mediator server host to which the probe sends its metrics.

   - To designate that a mediator be automatically assigned to the probe, perform the following:

     i. Enable auto-registration on each mediator server that you want to make available to the probe as an assignment option. Set the **commander.max.load.count.5s**, **commander.max.load.count.20s**, **mediator.max.load.count.5s**, and **mediator.max.load.count.20s** properties in **server.properties** file. For example:

     ```
     commander.max.load.count.5s = 0
     commander.max.load.count.20s = 0

     mediator.max.load.count.5s = 450000
     mediator.max.load.count.20s = 450000
     ```

     In this case, the mediator can hold up to 450000 active nodes.

     When **commander.max.load.count.5s**and **commander.max.load.count.20s** are set to zero, the server will not participate in auto-assignment. That is, the Commander Server will not get

auto-assigned to act as a mediator. This is recommended in a multi-server environment--only use the mediators to process incoming agent data.

ii. On the agent host, set the following properties in **etc/dispatcher.properties** to allow the commander to auto-assign mediators to the probes:disable start page

```
commander.registrar.url = http://<commander_host>:2006/registrar/
...
always.use.commander.registrar.url = true
```

The **commander.registrar.url** property specifies the Commander Server in the deployment. This is the Commander Server to which the mediators available for auto-assignment report.

The **always.use.commander.registrar.url** property set to "true" enables auto-registration for this probe. Note that when auto-registration is enabled, the **registrar.url** setting in **dispatcher.properties** is ignored.

For details, see the comments for these properties in the **etc/dispatcher.properties** file.

Adjusting the Heap Size for the Java Agent in the Application Server

The size of the heap can impact the performance of the Java Agent and the application server. The default value for the heap size is 64 MB, but an application server usually increases it to a larger amount. When you add the Java Agent to an application server, you may need to increase the heap size to accommodate the memory used by the Java Agent. For details, see "Requirements for the Diagnostics Java Agent Host in the relevant version of the **Diagnostics System Requirements and Support Matrices Guide** on the Software Support site (https://softwaresupport.softwaregrp.com/group/softwaresupport/).

The heap size is set in the application server JVM configuration using the following JVM argument:

```
-Xmx<size>
```

You can increase the heap size by updating the value specified in the **-Xmx<size>** option. See your JVM documentation for help on setting this parameter.

Configuring the SOAP Message Handler

The Diagnostics SOAP message handler is required for Java probes to support the following features:

- Collect payload for SOAP faults.
- Determine SOA consumer ID from SOAP header, body, or envelope.

For most application servers, the instrumentation points and code snippets are written to automatically configure the Diagnostics handlers for web services being monitored.

> **Note:** For some application servers, special instrumentation is provided in Diagnostics to automatically load the Diagnostics SOAP message handler.
>
> However, some manual configuration is required for WebSphere 5.1 JAX-RPC and Oracle 10g JAX-RPC. See "Loading the Diagnostics SOAP Message Handler " on the next page.
>
> In addition, the Diagnostics SOAP message handler is not available for all application servers, nor is

custom instrumentation available to capture SOAP faults or consumer IDs from SOAP payloads. Therefore, this feature is not available on all versions of all application servers. For the most recent information on Diagnostics SOAP message handler support, see the Diagnostics Support Matrix at Diagnostics_System_Requirements Guide.

This section includes the following:

- "Disabling the SOAP Message Handler" below
- "Loading the Diagnostics SOAP Message Handler " below

Disabling the SOAP Message Handler

By default, the SOAP message handler is enabled. You can disable the handler as follows:

In the **<agent_install_dir>/etc/inst.properties** file edit the **details.conditional.properties** property to include **mercury.enable.autoLoadSOAPHandler = false**.

If the SOAP message handler is disabled, you must manually configure where in the chain the handler gets installed.

Loading the Diagnostics SOAP Message Handler

The SOAP message handler is loaded automatically on most application servers but requires manual configuration on these application servers:

WebSphere 5.1 JAX-RPC

To configure the SOAP message handler on WebSphere 5.1 JAX-RPC, follow these steps:

Note: For WebSphere 6.1 JAX-WS web services, Diagnostics handlers are not supported. You must recompile the application with the Diagnostics SOAP handler classes.

1. Locate the Web service deployment descriptor (**webservices.xml**) for the application. The directory path should look similar to the following:

   ```
   <install_
   root>\config\cell\<Server>\applications\<WebServiceEAR>\deployments\<WebServiceNa
   me>\<WebServiceJAR|WARName>\WEB-INF
   ```

   Here is an example:

   ```
   C:\Program
   Files\WebSphere\AppServer\config\cells\MyServer1\application\WebServicesSamples.ear\d
   eployments\WebServicesSamplea\AddressBookJ2WB.war\WEB-INF
   ```

2. Edit the webservices.xml and add the Diagnostics handler for each <port-component>:

   ```
   <port-component>
   ...
    <handler>
     <handler-name>Diagnostics RPC Handler</handler-name>
      <handler-class>
      com.mercury.opal.javaprobe.handler.soap.ProbeRPCHandler
      </handler-class>
    </handler>
   ```

```
...
</port-component>
```

3. Copy the Diagnostics handler jar (**<agent_install_dir>\lib\probeSOAPHandler.jar**) to the WebSphere **lib** directory.

   Here is an example:

```
cp C:\MercuryDiagnostics\JavaAgent\DiagnosticsAgent\ lib\probeSOAPHandler.jar
C:\Program Files\WebSphere\AppServer\lib
```

   These steps were developed with IBM WebSphere 5.1.0 Application Server on Windows.

Oracle 10g JAX-RPC

To configure the SOAP message handler on Oracle 10g JAX-RPC, follow these steps.

1. Locate the Web service deployment descriptor (**webservices.xml**) for the application. The directory path should look similar to the following:

```
<OC4J_install_root>\j2ee\home\applications\<app name>\ <deployment name>\WEB-
INF\webservices.xml
```

2. Edit the webservices.xml and add the Diagnostics handler for each <port-component>:

```
<port-component>
...
<handler>
 <handler-name>Diagnostics RPC Handler</handler-name>
  <handler-class>
  com.mercury.opal.javaprobe.handler.soap.ProbeRPCHandler
  </handler-class>
</handler>
...
</port-component>
```

3. Copy the Diagnostics handler jar (**<agent_install_dir>\lib\probeSOAPHandler.jar**) to the **<OC4J_ install_root>\j2ee\home\applib** directory.

These steps were developed with Oracle Containers for J2EE (OC4J) 10g Release 3 (10.1.3.3) on Windows.

Configuring the Discovery of a New J2EE Server for CI Population

The agent provides data to populate the J2EE Application Server and J2EE Application Domain CIs in BSM/APM.

The probe automatically populates CIs for well known J2EE servers such as JBoss and WebLogic.

You can also configure application server discovery to populate CIs for other J2EE servers. Application server name can be directly specified or configured to be discovered by JMX or be discovered by a point/code snippet.

You configure application server discovery in the probe **etc/metrics.config** file as described below.

The class AppServerDiscoveryCollector is located in the **<agent_install_dir>/lib/probe-jmx.jar** file and you can write you own collector class to do both application server discovery and metrics collection.

The following is the configuration for application server discovery for a generic application server. Note the collector name is case sensitive and should be different from any collector name in the **metrics.config** file.

```
<user-defined-collector-name>.class.name =
com.mercury.diagnostics.capture.metrics.jmx.AppServerDiscoveryCollector
<user-defined-collector-name>.class.path = probe-jmx.jar
<user-defined-collector-name>.app_server.configure.discovery = true
<user-defined-collector-name>.app_server.type = <user-defined-type>
<user-defined-collector-name>.app_server.server_name =
<user-defined-server-name>
<user-defined-collector-name>.app_server.domain_name =
<user-defined-domain-name>
```

And then you should add the following Java system property definition in the app-server/javaprobe startup script or java command line.

```
-Dapp_server.discovery.collector=<user-defined-collector-name>
```

Every 15 minutes the probe refreshes the collectors (including the AppServerDiscoveryCollector) and makes the discovery based on any new configuration.

For the advanced user who knows how to use JMX to discover the new application server name and J2EE domain name, you may add the following configuration in the probe **etc/metrics.config** file.

```
<user-defined-jmx-collector-name>.class.name =
com.mercury.diagnostics.capture.metrics.jmx.JMXCollector
<user-defined-jmx-collector-name>.class.path = probe-jmx.jar
<user-defined-jmx-collector-name>.depends.on.class =
javax.management.MBeanServer
<user-defined-jmx-collector-name>.app_server.configure.discovery = true
<user-defined-jmx-collector-name>.app_server.type = <user-defined-type>
<user-defined-jmx-collector-name>.app_server.server_name =
<user-defined-server-name>
<user-defined-jmx-collector-name>.app_server.server_name.discovery.by.jmx =
<jmx-ObjectName>.<jmx-AttributeName>
<user-defined-jmx-collector-name>.app_server.domain_name =
<user-defined-domain-name>
<user-defined-jmx-collector-name>.app_server.domain_name.discovery.by.jmx =
<jmx-ObjectName-1>.<jmx-AttributeName-1>@<jmx-ObjectName-2>.<jmx-AttributeNa
me-2>
```

Special Considerations for Applications Based on the OSGi Framework

If your application is based on the OSGi framework, you may need to set some additional properties. If not already the default value, set the **osgi.java.profile.bootdelegation** property to the default value "ignore".

Then append com.mercury.* to the end of the **org.osgi.framework.bootdelegation** property in your **osgi.java.profile**. For example:

```
org.osgi.framework.bootdelegation= <existing packages>,com.mercury.*
```

Chapter 5: Configuring for Azul or Cloud Environments

This chapter includes:

- "Java Agents on Azul" below
- "Java Agents in Cloud Environments" on the next page

Java Agents on Azul

Azul provides two highly scalable and highly performing solutions for enterprise Java users: Vega and Zing. Vega is a special hardware appliance which connects to the user local network. Zing is a virtual equivalent of Vega, provided in a form of a guest image for VMware or KVM. A major advantage of the Azul appliances is its innovative pauseless garbage collector, which runs continuously and can handle heaps up to tens of gigabytes. Both appliances are supported by Diagnostics equally, although we tested only Zing in the lab.

The Java SDK or JRE provided by Azul installs on a traditional system, such as Linux or Solaris, but when it is invoked, it delegates the execution of any Java code to the appliance. Thus, although the Java application seems to be running where it was invoked, it actually runs on a different system. This is done seamlessly, so the application interacts with its environment just as if it was running on a local system. If the application makes a JNI call, it is made across the network to be executed on the originating host.

This execution model creates a number of issues for Diagnostics users. The JNI calls made by the probe are costly, but what is more important, they do not provide the results the user might expect.

- The CPU timestamps do not work correctly. They measure the CPU time used on the originating server, and therefore are useless.
- Process metrics are useless, too, because they measure the front-end process.
- In most cases, all system metrics are useless as well. They measure the originating system and are irrelevant to the application running on the appliance.
- Garbage collection metrics are confusing. Since Azul uses continuous garbage collector, seeing garbage collection percentages over 100% is normal.
- Heap Breakdown and Heap Walker do not work.
- VMware special timers do not work (even if using virtual appliance on VMware)\

Configuring Diagnostics for Azul VM

Invoking Azul java command requires adding parameters that properly identify the appliance to be used for running the application. This creates a difficulty for JREinstrumenter (unless run in Automatic Implicit mode), which needs to run the JRE to be instrumented in order to determine its version and vendor, but is not capable of adding the required parameters.

The solution is to edit the file **azul.properties** found in the Azul JRE installation and define the required parameters. The settings are needed while the JREinstrumenter runs and can be removed for running the application with Diagnostics.

To eliminate possible confusion and pointless overhead, we recommend to use the following settings while using Diagnostics Agent:

- In **metrics.config**, comment out all metrics for "system" and "ProcessMetrics" collectors, and Garbage Collection metrics for the "Java Platform" collector.
- In **capture.properties** set use.cpu.timestamps=false.

Java Agents in Cloud Environments

The Java Agent provides out-of-box support for monitoring Java applications in a cloud environment, such as ActiveState's Stackato or aPaaS. However, monitoring Java applications in these environments requires a slightly different Java Agent configuration and deployment procedure.

Cloud environments use dynamic application server instances that are scaled in and out as needed. Agents use a naming strategy in this environment that provide a consistent name for the application server instance in the Diagnostics Enterprise UI. A probe deployed on Stackato will have an assigned name that consists of the application name as defined by Stackato, and a suffix of its instance identifier. For example, an application named "OnlineBanking" with 3 instances would have the following probe names:

OnlineBanking_1
OnlineBanking_2
OnlineBanking_3

In general, the steps to configure and deploy the Java Agent in a cloud environment are as follows:

1. Add the Java Agent installed files to the directory structure that contains the application to be monitored, so that the agent is included when the application is pushed up to the cloud.

   Copy the **<agent_install_directory>/JavaAgent/Diagnostics** directory to your application workspace, and ensure that it is bundled with your resulting application assembly. Whether this is a **.war** file, **.ear** file, or directory structure, the Java Agent bits need to be included when the application gets pushed up to Stackato.

2. Configure the Java Agent as needed.

   Run the Java Agent Setup program as described in "Installing Java Agents" on page 16.

   - When prompted for the Agent Configuration, specify either "Enterprise Mode (AM License)" with "Diagnostics" or "Diagnostics Mode for Load Runner/Performance Center (AD License)".
   - When prompted to provide the Agent Name, enter any string. This placeholder value will be overwritten in the next step.

3. Configure the Stackato **stackato.yml** to deploy and enable the Java Agent. For details, see one of the sections below:

   - "Deploying a Java Agent on a Stackato-provided Application Server Container" below
   - "Deploying a Java Agent on a Stackato Stand-alone Application" on page 77

Deploying a Java Agent on a Stackato-provided Application Server Container

The steps assume the Stackato system is installed, configured correctly, and accessible to you. The examples show the steps needed to modify your application and **stackato.yml** in order to enable the Java Agent.

The below auto-deployment steps work with either the Tomcat or JBoss containers that Stackato uses.

1. Edit the **stackato.yml** configuration file in the Stackato workplace to add the Java Agent configuration commands to execute upon deployment on Stackato.

   The commands that you add depend on whether the application package that you deploy ends up extracted on deploying, as they refer to the Java Agent files within this directory structure.

   - **Application package is automatically exploded on deploying:**

     This is the most common case, for example the Stackato Tomcat application server automatically explodes the **.war** file upon deploying.

     If your application is pushed up as a directory or as a **.war** file, add the following to the **stackato.yml**:

     ```
     hooks:
      post-staging:
         - mv JavaAgent $STACKATO_APP_ROOT/
         - java -jar $STACKATO_APP_
     ROOT/JavaAgent/DiagnosticsAgent/lib/setupModule.jar
     ```

     where

     $STACKATO_APP_ROOT is defined by Stackato.

     The **JavaAgent** directory (which in this example contains the Java Agent bits) is moved up to the $STACKATO_APP_ROOT and a command is launched to deploy it to the startup script of the application server.

   - **Application package is not automatically exploded on deploying:**

     If the **.ear** file does not end up extracted when the application is pushed to Stackato, for example deploying an **.ear** file on JBoss, additional commands are required to temporarily extract the Java Agent bits from the **.ear** file and copy them up so that they can be deployed on the container.

     Add the following to the **stackato.yml**

     ```
     hooks:
       post-staging:
         - mkdir tmpdir
         - unzip -q jboss-as-kitchensink-ear.ear -d tmpdir
         - mv tmpdir/JavaAgent $STACKATO_APP_ROOT/
         - rm -r tmpdir
         - java -jar $STACKATO_APP_
     ROOT/JavaAgent/DiagnosticsAgent/lib/setupModule.jar
     ```

     where

     $STACKATO_APP_ROOT is defined by Stackato.

     The **JavaAgent** directory is included as part of the **.ear** file.

2. Deploy the repackaged application to Stackato. For example, run the following command in the top directory of your workplace:

   ```
   stackato push -n
   ```

After staging the application, Stackato executes the post-staging steps that you specified in the **stackato.yml** configuration file. The first step moves the Agent bits to a fixed location, and the second step invokes the

Agent command to automatically deploy itself within the application server (either Tomcat or JBoss) that Stackato uses as a container for your application.

> **Note:** The automatic deployment tool expects to find the Java Agent at **STACKATO_APP_ROOT/JavaAgent/DiagnosticsAgent**. This directory where the agent is moved to cannot be changed elsewhere in **stackato.yml.**

Deploying a Java Agent on a Stackato Stand-alone Application

When deploying the Java Agent on a Tomcat or JBoss application server, the agents can auto-deploy to those application servers. If your application is instantiated by a script that you provide to Stackato, then you need to manually specify the parameters to enable the Java Agent.

To do this, add the following commands to your application startup script:

- - Ddiag.config.override=stackato
- - javaagent:${HOME}/<agent dir in your app>/lib/probeagent.jar

For example, assume a **stackato.yml** file as follows:

```
name: onlinebank
mem: 512M
framework:
type: generic
processes:
web: /app/app/myStartupScript.sh
```

You need to edit the **myStartupScript.sh** to add the following to the command that is invoking Java:

```
-Ddiag.config.override=stackato -javaagent:${HOME}/agent/lib/probeagent.jar
```

The -Ddiag.config.override argument directs the probe to read the file: **<agent_install_directory>/etc/overrides/stackato.settings** when the application starts. The **stackato.settings** file contains the necessary property settings for probes in Stackato–overriding their specified value (if any) in the standard property and configuration files for the agent. This file contains the rules for determining the probe and host names according to the Stackato environment. The out-of-box settings should be appropriate for most scenarios, but if you want to customize the names created for the probes or their hosts, you can change the settings in this file.

You can add additional property settings to the **stackato.settings** file or create a custom version of this file as needed and rename it. The custom settings file must be located in the **<agent_install_directory>/etc/overrides** directory and have the ".settings" suffix.

Note that any overrides in the **stackato.settings** file to dynamic properties are overridden unconditionally. Changes to any dynamic properties that occur after the application starts are ignored.

Just like for non-cloud agent deployment, the jreinstrumentor must be run in order to enable collection leak pinpointing. See "Examples for Configuring Application Servers " on page 33 for details.

Chapter 6: Preparing Application Servers for Client Monitoring with the Java Agent

This chapter includes:

- "About Client Monitoring" below
- "Enabling Client Monitoring" on the next page
- "Configuring and Disabling Client Monitoring" on page 80
- "Manually Instrumenting HTML/JSP Pages for Client Monitoring" on page 81

About Client Monitoring

Client Monitoring measures web page performance as seen by the user's browser and correlates these measurements with the back end server request.

Three important metrics are measured:

- The back-end time is the amount of time it takes from when a web page request is sent until the first byte of the response is received.
- The front-end time is the amount of time it takes from when the first byte of the response is received until the page is loaded.
- The total-time is the sum of the front and back end times.

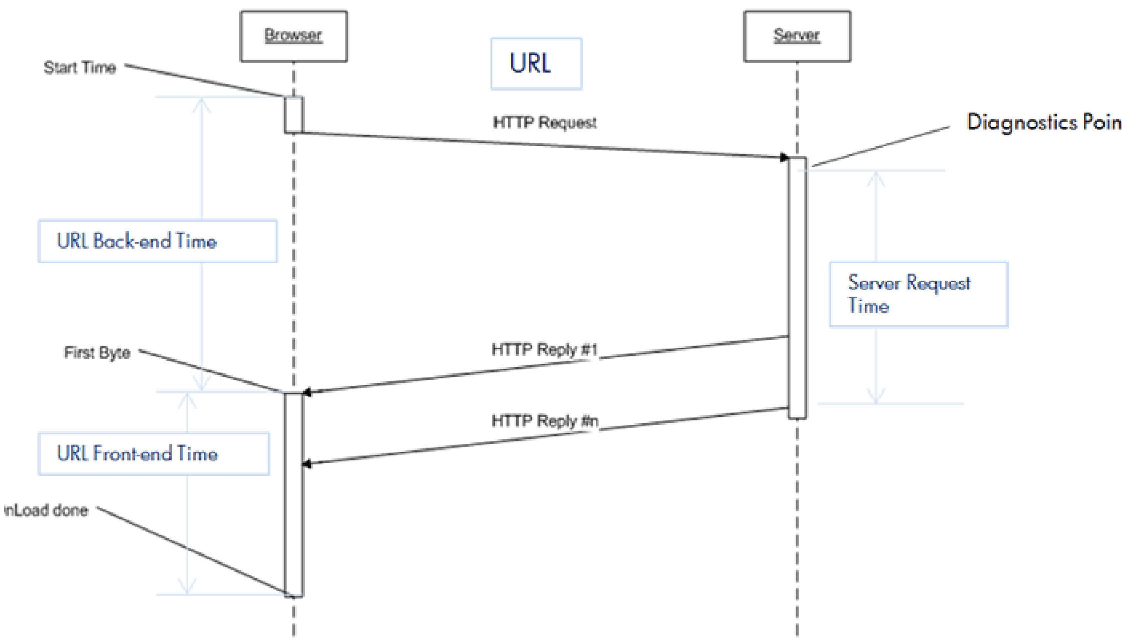
Client Monitoring aggregates these measures and presents them by URL, Location, and Browser-OS combination.

By monitoring web page performance, application owners can quickly identify performance issues, characterizing them by tier (front or back-end), location, and browser.

When the issue is on the back-end, client monitoring correlates the URL to the associated server request and its call-profile.

Note: Client Monitoring is not supported in Diagnostics Profiler mode.

An example showing client monitoring is shown below:



Enabling Client Monitoring

Enabling client monitoring requires you to deploy a .war file on the application server and in some cases to configure the web server. Client Monitoring views are available in the Diagnostics Enterprise UI.

For the list of browsers that can be monitored by the Client Monitoring feature, refer to the relevant version of the **Diagnostics System Requirements and Support Matrices Guide** on the Software Support site (https://softwaresupport.softwaregrp.com/group/softwaresupport/).

To enable Client Monitoring:

When client monitoring is enabled, most JSP pages served via JBoss, Tomcat, WebSphere and WebLogic will be automatically modified to include additional Java Script calls near the <head> tag. You can see which pages are instrumented by opening the page in your browser and selecting view source.

Other application servers may require manual page instrumentation for client monitoring. See "Manually Instrumenting HTML/JSP Pages for Client Monitoring" on page 81.

Client monitoring, including **automatic JSP instrumentation**, will remain disabled until this .war file is deployed.

1. Deploy **DiagnosticsCM.war** file.

   Use the application server's Administrative Console to deploy the **<agent_install_dir>\contrib\DiagnosticsCM.war** as an application.

   Client monitoring will remain disabled until this .war file is deployed.

   For WebSphere application servers, be sure to set the context root to **/DiagnosticsCM** instead of the default (/).

2. If you have configured a web server as the front-end of your application, then you also need to add the following context root to your Web Server's configuration:
   /DiagnosticsCM/*

Tip: You can verify the web server is correctly configured if your browser can access this link: (it will return a blank page)
`http://hostname:port/DiagnosticsCM/B/`.

Example - Setting up an Apache HTTP Server Reverse Proxy for Client Monitoring

Note: These are very basic instructions. These configuration files are highly customized in each customer's environment. Please consult the Apache HTTP Server documentation for more details.

In order for client monitoring JavaScript file to be successfully downloaded by browsers and for client-side metrics to be received by the probe, it is necessary to configure the web server to correctly forward those requests to the application server. This is typically achieved by setting up a reverse proxy or gateway.

1. Update the conf\httpd.conf file by adding the following lines, replacing <HostName> and <HostPort> with the host name and port of the application server, and restart the web server.

   ```
   ProxyPass         /DiagnosticsCM http://<HostName>:<HostPort>/DiagnosticsCM
   ProxyPassReverse  /DiagnosticsCM http://<HostName>:<HostPort>/DiagnosticsCM
   ```

2. Check if your changes are successful by driving traffic to your web application via the web server and checking the web server's log messages in the log/access.log file. Error messages will have an http response code in the 400-500 range such as "GET /DiagnosticsCM/boomerang-min.js HTTP/1.1" 404. When successful, you should see log messages such as "GET /DiagnosticsCM/boomerang-min.js HTTP/1.1" 200.

   If you don't see either of these messages, then client monitoring is not correctly set up in your environment.

Configuring and Disabling Client Monitoring

If desired, Client Monitoring can be dynamically controlled by updating several properties in **<agent_install_directory>\etc\dynamic.properties**.

The **client.monitoring.enable** property provides a master switch to dynamically enable and disable the client monitoring feature. When set to false, all client monitoring data events received are dropped, JSP page auto-instrumentation will be disabled, and **client.monitoring.sampling.percent** is set to 0.0 (to disable manually instrumented JSP pages' client monitoring Java Script code).

You can reduce the client monitoring load on your server by adjusting the c**lient.monitoring.sampling.percent** property in **dynamic.propertes**.

You can also specify that you want a strict check on the referrer by setting **client.monitoring.strict.referrer** to **true**. This will help ensure that only events that originate from a web page instrumented with client monitoring are used. The default value is false but the recommended value is true if this setting works in your environment.

You can also stop or uninstall/undeploy the DiagnosticsCM.war using your application server management console.

Manually Instrumenting HTML/JSP Pages for Client Monitoring

Add the following code to your HTML/JSP pages immediately after the <head> tag:

```
<!-- Client Monitoring -->
<script>
if (window.t_firstbyte === undefined) {
  var t_firstbyte = Number(new Date());
}
</script>
<script type='text/javascript' src='/DiagnosticsCM/boomerang-min.js'>
</script>
<script>
BOOMR.init({beacon_url:"/DiagnosticsCM/B",
  RT:{cookie:"X-HP-CM-RT",cookie_exp:600,expandFrames:true,hashURLs:true},
  HP:{cookie:"X-HP-CM-GUID"}});
</script>
```

If you prefer to manually instrument HTML/JSP pages you can permanently disable auto-instrumentation by setting the following properties in inst.properties to false. These changes require a restart of the application server.

in **<agent_install_dir>\etc\inst.properties**:

```
details.conditional.properties= \
mercury.enable.clientmonitoring.JspWriterImpl.autoinstrumentation=false,\
mercury.enable.clientmonitoring.CoyoteWriter.autoinstrumentation=false,\
mercury.enable.clientmonitoring.BodyContentImpl.autoinstrumentation=false,\
```

Chapter 7: Upgrading the Diagnostics Java Agent

This chapter presents the information that you need to upgrade the Diagnostics Java Agent.

This chapter includes:

- "Upgrade Java Agents" below
- "Upgrade Notes and Limitations" on page 84

Upgrade Java Agents

Note: As of Diagnostics 9.23, the format and process for the Java Agent installation package have changed. For detailed instructions on installing the Java Agent, see "Installing Java Agents" on page 16.

Consider the following when planning the Diagnostics Agent upgrade:

- You must upgrade the Diagnostics Server before upgrading the agents that are connected to it because Diagnostics Servers are not forward compatible.
- With each new release of Diagnostics you should re-record the Java agent silent install response files prior to performing silent installation on multiple machines.

Note: The new agent installation will not begin monitoring your applications until you have updated the startup scripts to start the new agent and restarted the applications as described in these instructions.

To upgrade a Java Agent:

1. Install the Diagnostics Agent for Java **into a different directory** than the current agent's installation directory.

   During the installation, be sure to do the following. This ensures that the persisted data for your application will match up with the metrics captured by the new agent.

   - Configure the Java Agent to work with a Diagnostics Server or as a standalone Diagnostics Profiler.
   - For the agent name, use the same probe name as used by the previous agent.
   - For the agent group name, use the same group name as used by the previous agent.
   - For the mediator server name and port, use the same information as used by the previous agent.

   See "Installing Java Agents" on page 16 for additional information you need for installing a Java Agent.
2. Compare the new agent's **\etc** directory and the previous agent's **\etc** directory so that you can determine the differences between the two.

We recommend that you execute the **Property Scanner** utility provided with the Java Agent which will indicate the differences (properties and points) between two different Java Agent installations. To execute the Property Scanner utility, change the current directory to **<agent_install_dir>/contrib/JASMUtilities/Snapins** and execute the **runPropertyScanner.cmd –console** (**.sh** for Unix) command as follows:

```
runPropertyScanner –console –diffOnly yes –Source1 ..\..\..\etc –Source2 OtherEtc
```

Sample Input:

```
C:\MercuryDiagnostics\JavaAgent8\DiagnosticsAgent\contrib\JASMUtilities\Sna
pins>runPropertyScanner -console -diffOnly yes -Source1
C:\MercuryDiagnostics\JavaAgent\DiagnosticsAgent\etc -Source2
C:\MercuryDiagnostics\JavaAgent8\DiagnosticsAgent\etc
```

Sample Output:

```
PropertyFile=dispatcher.properties
Property=stack.trace.method.calls.max
Source1=
Source2=1000
```

Apply any differences that were caused by the customizations that you made to the previous agent’s **\etc** directory to the new agent’s **\etc** directory so that they will not be lost. You should look for the following changes:

Property File	Configuration Properties to Be Copied to the New Diagnostics Server
auto_detect.points	Copy custom points that you have created and points that you have modified from the **auto_detect.points** file in the old **etc** directory to the new **etc** directory. Be sure to check the points for RMI, LWMD, args_by_class when looking for points you may have modified.
capture.properties	Depth and latency trimming.
dispatcher.properties	minimum.sql.latency sql.parsing.mode
dynamic.properties	cpu.timestamp.collection.method
metrics.config	Verify that any metric that you uncommented in the previous version is also uncommented in the new version so that you can continue to use the metrics that you are used to.
security.properties	If the system is set up for SSL mode, set all properties and copy the certificates from the old property file to the property file.

3. Prepare your application servers to be monitored using the JRE instrumentation methods described in the "Examples for Configuring Application Servers " on page 33. In particular you need to update the application’s startup script or JVM parameters to point to the upgraded agent installation.

4. At an approved time, shut down the applications that were being monitored by the old agent.

5. Restart the applications to allow the new version of the agent to begin monitoring your applications.
6. Clear your browser's cache and the Java plug-in cache. Restart the browser before you attempt to access the Diagnostics Profiler for Java user interface. Failure to do this may result in a size mismatch error message.
7. You can verify that the upgraded Diagnostics Agent is running by checking the version in the System Health view in the Diagnostics UI. The version should be the latest version if the upgrade was successful. To access the System Health view you must open the Diagnostics UI as the System customer from `http://<Diagnostics_Commanding_Server_Name>:2006/query/`. Then in the Views pane you can select the System Views view group.
8. When all your applications have been migrated over to be the latest version and everything is working properly, you can delete the old directory. Don’t try to uninstall the old version because this will actually uninstall the new version.

Upgrade Notes and Limitations

As of Diagnostics version 9.24, by default HTTP methods (such as PUT, GET, and POST) are used as an identifying component for each HTTP/S Server Request and a separate HTTP Server Request is generated for each HTTP method to the same URL. In earlier versions of Diagnostics, the first instrumented Java method executed by the Server Request is used for identification and one HTTP Server Request is generated for all HTTP methods to the same URL.

We recommend using the new method of server request identification, even though this is not backward compatible and breaks trend lines. If you must maintain continuity of trend lines, in the **dispatcher.properties** file, change the value of the **fragment.use.http.method** setting to **false**.

Part 3: Advanced Java Agent Configuration and Instrumentation

Chapter 8: Monitoring Profiles

This chapter describes monitoring profiles.

This chapter includes:

- "About Monitoring Profiles" on the next page
- "Understanding Types of Diagnostics Deployments" on the next page
- "The Predefined Monitoring Profiles" on page 89
- "Custom Monitoring Profiles" on page 89
- "Applying a Specific Monitoring Profile to a Probe" on page 90
- "Overriding Settings in the Monitoring Profiles" on page 91
- "Mapping Instrumentation Points to a Monitoring Profile" on page 92
- "Mapping Metrics to a Monitoring Profile" on page 92
- "Mapping Property Values to a Monitoring Profile" on page 92

About Monitoring Profiles

A monitoring profile is a collection of predefined settings that control the amount of collected data for a particular Java Agent instance (a probe).

Java Agents are highly configurable. Monitoring profiles are a safe and easy way to manage the impact of the probe on the monitored system and still obtain the needed performance data.

Understanding Types of Diagnostics Deployments

Each probe has the ability to capture many events such as method invocations, server requests, and system usage metrics from the Java application it is monitoring. In general, the more collected data, then the more information is readily available to identify performance issues. However, the more collected data, then the more overhead on the monitored system. Overhead can affect the monitored application's ability to provide its services as well as the probe's ability to report the data in the Diagnostics Enterprise UI or Profiler UI. The type of deployment determines how much overhead is acceptable.

Diagnostics operates in different environments, ranging from development desktops to systems deployed in production. The following tables describes the three main categories of Diagnostics deployments.

Diagnostics Deployment	Data Persistence	Goals of This Deployment	Collected Data/Overhead
Enterprise–Java Agent sends data to a Diagnostics Server. Optionally integrated with BSM/APM. **Users**: Operations	Diagnostics collects and stores data from hundreds or thousands of probes and keeps the data for up to 5 years.	• Designed for Production • Alert users to performance or availability issues, and diagnose memory leaks. • Maximize availability of business critical applications • Reduce MTTR of business critical problems • Produce actionable data for development	Lower amounts of collected data Lower overhead
Performance Center/Load Runner Integration **Users**: QA	Configurable, but the expectation is that Diagnostics collects and stores data from dozens of probes and keeps the data for as long as the testing cycle–typically several months.	• Designed for load testing • Diagnose distributed application issues, help users tune the application for better performance and scalability. • Reduce MTTR of performance issues • Provide actionable root cause data to development	
Diagnostics Profiler **Users**: Development	Diagnostics does not persist any data.	• Designed for development environment • Diagnose slow methods, exceptions, and coding issues • Ready applications for load testing	Higher amounts of collected data Higher overhead

The Predefined Monitoring Profiles

Diagnostics provides three predefined monitoring profiles–one for each type of Diagnostics deployment above.

By default, a probe uses one of the predefined monitoring profiles on startup. The Agent Mode of the probe determines which predefined monitoring profile is used, as follows:

Predefined Monitoring Profile	Agent Mode (Specified During Agent Setup)	Collected Data/Overhead
Application monitoring in production environment	Enterprise Mode (AM License) and Diagnostics	Lower amounts of collected data Lower overhead
Application monitoring in pre-production environment or extended monitoring in production environment	• Diagnostics Mode for LoadRunner/Performance Center (AD License) • Enterprise Mode (AM License)	
Application profiling in developer environment	Diagnostics Profiler Mode	Higher amounts of collected data Higher overhead

Settings specified by the predefined monitoring profiles are overridden if the setting is specified elsewhere. See "Overriding Settings in the Monitoring Profiles" on page 91.

Custom Monitoring Profiles

You can use a custom monitoring profile instead of the predefined monitoring profiles. To create and use a custom monitoring profile, follow these steps:

1. Choose a numerical value to represent the profile.

   Use a positive integer that is not already in use for a monitoring profile in this installation. The predefined monitoring profiles use the following numbers:

120	Application monitoring in production environment
140	Application monitoring in pre-production environment or extended monitoring in production environment
170	Application profiling in developer environment

   To help you manage multiple profiles, follow these naming guidelines:

   - All data collected by a numerically lower profile is also collected by the numerically higher profile.
   - The higher the number, the more data is collected, with a higher overhead.

   For example, if a particular production environment puts unusually strict restrictions on tool overhead, you could define a new profile named **115** with the modified settings.

2. Customize the settings.
   a. Use one of the predefined **.settings** files in **<agent_install_directory>/etc/defaults** as a starting point; copy and rename it to the same location. For example **<agent_install_directory>/etc/defaults/115.settings**.

      Modify the settings to limit the amount of collected data, for example:

```
#
# Settings for my '115' monitoring profile
#
...
dispatcher.minimum.fragment.latency = 100ms
...
```

      For information about the format of the **.settings** file, see "Mapping Property Values to a Monitoring Profile" on page 92.
   b. Modify the capture points file to map any instrumentation points to the new custom monitoring profile. See "Mapping Instrumentation Points to a Monitoring Profile" on page 92.

      Note: For best practices, ensure that all .settings files in your deployment contain the exact same set of properties. Because when a property is specified in one **.settings** file, it means that the property definition in the original property file is commented out. Therefore each **.settings** file must define the property.

   c. Modify the **metrics.config** file to map any metrics to the new custom monitoring profile. See "Mapping Metrics to a Monitoring Profile" on page 92.
3. Run the probe with the new custom monitoring profile. For details on how to do this, see "Applying a Specific Monitoring Profile to a Probe" below.

Applying a Specific Monitoring Profile to a Probe

To apply a monitoring profile to a probe, use one of the following methods:

- By setting the probe property **monitoring.profile** in **<agent_install_directory>/etc/probe.properties**. For example:

```
monitoring.profile = 115
```

  Changes to this setting are picked up dynamically–they take effect shortly after the changes are saved to the file.

- As a system property on the application server start up command line. For example:

```
-Dprobe.monitoring.profile=115
```

  If the specified monitoring profile does not exist (there is no settings file in the **<agent_install_directory>/etc/defaults** directory that corresponds to the number), the probe substitutes an existing **.settings** file corresponding to the value closest to the specified profile but not over the value.

- In the **Profiler UI**, select **Configuration tab > Probe Setting pane > General section >**

Monitoring Profile and select the required monitoring profile from the list. Click **Apply Changes**.

> **Note:** In the **General** section, you can also choose to disable monitoring data collection without stopping the Java Agent.

Changes to this setting are picked up dynamically–they take effect shortly after the changes are saved to the file.

Settings specified by the custom monitoring profiles are overridden if the setting is specified elsewhere. See "Overriding Settings in the Monitoring Profiles" below.

Overriding Settings in the Monitoring Profiles

Settings specified by the predefined or custom monitoring profiles are overridden (ignored) as follows:

- Settings in **<agent_install_directory>/etc/*.properties** files override the settings in the monitoring profile.

  By default, a setting managed by the predefined monitoring profiles is disabled in the associated property file. For example, in **capture.properties**:

  ```
  ## Latency trimming
  ...
  # The default value is defined by the current monitoring profile
  #
  #minimum.method.latency = 5ms
  ```

  To override the minimum.method.latency setting from the monitoring profile, simple uncomment it here and set the value.

  ```
  ## Latency trimming
  ...
  # The default value is defined by the current monitoring profile
  #
  minimum.method.latency = 7ms
  ```

  You can use this capability to easily customize settings that are specific to a deployment environment without changing the monitoring profile. For example, using specific JMX/PMI metrics or instrumentation points.

- Property settings specified as Java system properties on the application server startup command line override the settings in the monitoring profile

  For more information about specifying properties in this way, see "Specifying Probe Properties as Java System Properties" on page 31.

- Dynamic properties, if generally accepted by the probe, override the settings in the monitoring profile.

  For more information about dynamic properties, see "About Dynamic Configuration" on page 159.

Mapping Instrumentation Points to a Monitoring Profile

The **profile** keyword for the instrumentation point details maps the point to a monitoring profile. The keyword is specified in the form of profile:<number>. The number indicates that the point is enabled for all profiles at the value of number or higher. The point is disabled for all monitoring profiles lower than the specified value.

For example:

```
 [Servlet-all]
; ------------- extends HttpServlet ---------------------
; (See HttpCorrelation point for ignore documentation)
; In addition, ignore class we know we are not interested in

...
deep_mode  = soft
layer      = Web Tier/Servlet
detail     = profile:140
```

The instrumentation point is enabled on the predefined profiles 140 and 170, and all custom profiles 141 or higher. The point is disabled on the predefined profile 120, and on all custom profiles 139 or lower.

By default, the capture points file is located at **<agent_install_dir>\etc\auto_detect.points**. Your agent installation may be using a custom capture points file in a different location.

Instrumentation points can still be enabled and disabled dynamically, regardless of the selected monitoring profile. See "Adding a Disabled Point and Enabling it at Runtime" on page 124.

Mapping Metrics to a Monitoring Profile

The **P<number>?** notation in the metrics.config file maps the metric to a monitoring profile. Just as for instrumentation points, the number indicates that the metric is enabled for all profiles at the value of number or higher. The metric is disabled for all monitoring profiles smaller than the specified value.

For example, in **<agent_install_dir>\etc\metrics.config**:

```
P135?system/PageOutsPerSec = PageOutsPerSec|count|System
```

The metric is collected for a custom profile 135 and for any higher custom profiles. The metric is also collected for the predefined profiles 140 and 170 since they are higher than 135. The metric is not collected for any custom or predefined profiles less than 135.

Mapping Property Values to a Monitoring Profile

The monitoring profile property settings files map property settings to a monitoring profile.

The monitoring profile property settings files are in **<agent_install_dir>\etc\defaults**. Each predefined monitoring profile has its own property settings file, for example 120.settings. Custom monitoring profiles will also each have a settings file here–you need to create those files.

Each property files contains the property definitions for the respective monitoring profile. For example, the **120.settings** contains:

```
#
# Default settings for the '120' monitoring profile
#
title = Application monitoring in production environment

capture.minimum.method.latency = 51ms
capture.maximum.method.calls = 1000

dispatcher.minimum.fragment.latency = 51ms
dispatcher.minimum.sql.latency = 1s
...
```

In the file, property names are constructed by using the module name (which is generally the same as the property file root name) as the prefix for the property, and separating it from the property name with a dot.

For example, the **capture.maximum.method.calls** property above is for maximum.method.calls property from capture.properties. The maximum.method.calls property definition in capture.properties is commented out as follows.

```
# Never capture more than this number of methods per instance tree.
# This is regardless of latency and depth trimming.
# Note that this applies all methods, including outbound calls.
# The default value depends on the monitoring profile.
#maximum.method.calls=
```

All monitoring profile property files should contain the same property definitions–with potentially different values, of course. At the same time, the property definition in the original property file should be commented out.

When the probe resolves the properties, it checks **<agent_install_dir>/etc/defaults** last. That is, the probe only uses the property definition from the monitoring profile properties file when there is no definition found in the primary properties file.

This allows you to override some of the properties for all profiles with a single line change, simply by uncommenting the property in the primary property file and providing the universal, monitoring profile independent value. Also, for those properties that can be dynamically changed, this allows you to change the property by modifying the module specific property file, without even knowing which monitoring profile is or will be selected.

Chapter 9: Automatically Assigning a Probe to an Application

This chapter describes how to automatically assign a probe to an application.

This chapter includes:

- "About Automatic Probe Assignment" below
- "Configuring a Probe to Automatically Assign Applications" below
- "Configuring an Agent to Automatically Assign Applications" below
- "General Configuration" on the next page

About Automatic Probe Assignment

You can assign a probe to an application so that in the Diagnostics Commander UI, you can view the probe data within the context of that application. You can assign a probe to an application by the following methods:

- Configure the Java Probe or Agent to automatically create applications in the Diagnostics Commander and associate monitored data with the application. For details, see below.
- Manually create an application in the Diagnostics Commander and select the entities associated with it. For details, see "Working with Applications" in the Diagnostics User Guide.
- Use scripts with Composite Application Discovery (CAM). For details, see "Automating Composite Application Discovery in Diagnostics" in the Diagnostics Server Installation and Administration Guide.

Configuring a Probe to Automatically Assign Applications

To automatically create an application (if it does not already exist) and assign an individual probe to it, you set the probe property setting -Dprobe.belongsto.application as a Java system variable. For example, setting `-Dprobe.belongsto.application=MyGroupName/MyAppName` creates a group called MyGroupName and within it, an application called MyAppName, to which the probe is assigned. For details on setting a probe property as a Java system variable, see "Specifying Probe Properties as Java System Properties" on page 31.

Note: Use a single slash (/) as a separator. For example, MyGroupName/MyAppName.

Configuring an Agent to Automatically Assign Applications

To automatically create an application (if it does not already exist) and assign an agent to it, you configure the **belongsto.application** parameter in the **<agent_install_directorydir>etc/probe.properties** file. For example:

Setting `belongsto.application=MyGroupName/MyAppName` creates a group called **MyGroupName** and within it, an application called **MyAppName**, to which all probes on the agent are assigned.

Setting `belongsto.application=${MyAppGroup}/MyString/${PROBE_ID}`, creates a group with the name of the value in the **MyAppGroup** variable, within it a sub-group called **MyString**, and within that, an application with the name of the value in the **PROBE_ID** variable. Since the application name is specific to one probe, only that probe is assigned to it as each probe on the agent creates and application with a different name.

Note:

- Use a slash (/) as a separator. For example, MyGroupName/MyAppName.
- All the probes of an agent are assigned to the configured group, unless you use variables that create different groups or applications to which specific probes can be assigned.
- Changing the belongsto.application parameter in the <agent_install_directorydir>etc/probe.properties file requires you to restart the application the probe is monitoring.
- The **Belongs to Application** field in the Diagnostics Commander UI is only populated when there is at least one reported server request.

General Configuration

By default, automatically assigning a probe to an application is enabled and a task is run every 5 minutes to check for new group and application names to be created. You can disable this feature and change the frequency of the task, by editing the following parameters in the **<diag_server_install_dir>/etc/server.properties** file on the Diagnostics server:

belongsto.application.rules.disable. By default, this feature is enabled (set to **false**).

belongsto.application.frequency. By default, this is set to 5 minutes, which is the minimum you can set.

probe.topology.discovery. Adds connected probes to the application. By default, this feature is enabled (set to **true**).

Note: These parameters are dynamic and changing them does not require a system restart.

Chapter 10: Custom Instrumentation for Java Applications

This chapter explains how to control the instrumentation that Diagnostics applies to the classes and methods of the applications to enable the Java Agent to gather the performance metrics.

This chapter includes:

- "About Instrumentation and Capture Points Files" below
- "Using Regular Expressions in Points Files" on the next page
- "Coding Points in the Capture Points File" on page 98
- "Defining Points With Code Snippets" on page 103
- "Controlling Class Map Capture" on page 113
- "Instrumentation Examples" on page 114
- "Understanding the Overhead of Custom Instrumentation" on page 126
- "Instrumentation Control on a Per Layer Basis" on page 126
- "Instrumented Location Throughput Throttling" on page 127
- "Advanced Instrumentation Examples" on page 128
- "Configuring Cross VM Correlations for New or CustomTechnologies" on page 137
- "Tutorial for Configuring Cross VM Correlation for Custom Technologies" on page 140
- "Maintaining Instrumentation from the Java Profiler UI" on page 147
- "Default Layers Defined for Typical Java Classes and Methods" on page 156

About Instrumentation and Capture Points Files

Instrumentation refers to bytecode that the probe inserts into the class files of the application as they are loaded by the class loader of your virtual machine. Instrumentation enables a probe to measure execution time, count invocations, retrieve arguments, catch exceptions, and correlate method calls and threads.

Instrumentation is controlled by instrumentation *points*. The points define which methods to instrument, how they should be instrumented, and which instrumentation should be installed. Instrumentation points for each probe instance are specified in a capture points file.

The points in the capture points file are grouped into layers. Layers organize the performance metrics into meaningful tiers of information that can be compared as part of a triage process. They control the collection behavior of the instrumentation. You can customize the default layers and create new layers. For description of the default layers see "Default Layers Defined for Typical Java Classes and Methods" on page 156.

When you install the Java Agent, a predefined capture points file is installed with a set of points for the platform you are using. This default capture points file is located at **<agent_install_directory>\etc\auto_detect.points**.

You can customize the points in the capture points files to include methods, classes, packages, and namespaces for areas of the application that do not fall within the default points. A common situation that might require custom points is when a J2EE application contains business logic that is not derived from the javax.ejb.SessionBean interface. Another situation for custom points is when you want to override a default point to alter its layer or to track it from a specific caller method.

To add custom instrumentation, you can do one of the following:

- Modify the **<agent_install_directory>\etc\auto_detect.points** file with your instrumentation customizations. All probes on the same host use this instrumentation. You will need to back up this file and merge back your changes when upgrading the Java agent.
- Copy and rename the **<agent_install_directory>\etc\auto_detect.points** file and then add your instrumentation customizations. Specify the name and location of the new points file in the **<agent_install_directory>\etc\probe.properties** file. For example:

  ```
  # Name of the instrumentation points file to be used when reporting
  # to AM/BAC or AD/LoadRunner/PC. The default value is "auto_detect"
  # which points to probeInstall/etc/auto_detect.points file.
  #

  # points.file.name=auto_detect
  points.file.name=my_custom_points
  ```

  The file name must have the ".points" suffix although the file name that you specify in **<agent_install_directory>\etc\probe.properties** does not have the suffix. The file location that you specify is relative to the **<agent_install_directory>\etc** directory.

  All probes on the same host use this instrumentation. Using a copy of the file prevents you from needing to back up and restore it when upgrading the Java agent.
- Copy and rename the **<agent_install_directory>\etc\auto_detect.points** file following the naming guideline note below. Then add the instrumentation customizations that are needed for an individual probe.

  Note: A custom capture points file name must be different than the probe name. Custom capture points file names that match the probe name are reserved for internal use. To help you recognize the probe associated with a custom capture points file, use the probe name with a suffix of prefix. For example, for a probe named “MyProbe” you can specify a custom capture points file name of “MyProbe_custom”.

  Specify the name and location of the new points file as the "-Dprobe.points.file.name"JVM parameter when you start the application server. How you start the application server depends on the type of application server. For example on GlassFish the JVM parameters would be:

  ```
  -javaagent:<agent_install_directory>/lib/probeagent.jar
  -Xbootclasspath/p:<agent_install_directory>/classes/auto/<probe_id>/instr.jre
  -Dprobe.id=<probe_id>
  -Dprobe.points.file.name=WL10_MedRec_ovrserver130_custom
  ```

  Only this application server instance (JVM) uses the custom points file. The instrumentation in the **auto_detect.points** or other custom instrumentation file on the host is ignored.

Using Regular Expressions in Points Files

Points can include regular expressions that "wildcard" the instructions so that they apply to more than one method, class, and package or namespace specification. For more information about using regular expressions, see “Using Regular Expressions” in the Diagnostics Server Installation and Administration Guide.

Coding Points in the Capture Points File

The following arguments can be used to define a point in the capture points file:

```
[Point-Name]            = <unique name for the point>
;------------------------------------------------------------------------
class                   = <class name or regular expression>
method                  = <method name or regular expression>
signature               = <method signature or regular expressions>
ignore_cl               = <list of class names or regular expressions>
ignore_method           = <list of method names or regular expressions>
ignore_tree             = <list of class names or regular expressions>
method_access_filter    = <list of class names or regular expressions>
deep_mode               = <soft or hard mode>
scope                   = <list of methods or regular expressions>
ignoreScope             = <list of methods or regular expressions>
detail                  = <list of specifiers>
layer                   = <layer name>
layerType               = <layer type>
rootRenameTo            = <string>
keyword                 = <keyword>
priority                = <integer number>
active                  = <true, false>
```

The following sections describe the arguments.

- "Mandatory Point Arguments" below
- "Optional Point Entries" on the next page

Mandatory Point Arguments

Every point, except for the points for CLP, LWMD, RMI and SAP RFC, HttpCorrelation, and JDBC SQL, must contain the following arguments:

Argument	Description
Point-Name	A unique name for the point.
class	Specifies the name of the class or interface to be instrumented. The name should include the full package/namespace name using periods between the package levels. Any valid regular expression can be used.
method	Specifies the name of the method to be instrumented. To be successful, the method name must match a method defined in the class or interface specified by the class argument. Any valid regular expression can be used.
signature	Specifies the signature (parameter and result types) of the method using javap symbolic encoding for method signatures (<jdk_install>/bin.javap -s).

Argument	Description
layer	Specifies a layer, sublayer, or tier under which the data from this point is grouped. Layers are a part of the instrumentation collection control. Layers in a point can be specified with nested layers or sublayers by separating the layer names with a / (slash). The layer specified following the slash is a sublayer of the layer specified before the slash. A sublayer can have its own sublayers by coding another slash and layer name following a sublayer name. In the UI, the sublayers for a layer are displayed under their parent layer. For example, the sublayers JSP and Struts would be displayed under the web layer and a drilldown would exist from Web to JSP and Struts.

The following is an example of a custom point that contains the mandatory arguments:

```
[MyCustomEntry_1]
; comments here….
class = myPackage.myClass.MyFoo
method = myMethod
signature = !.*
layer = myCustomStuff[MyCustomEntry_1]
```

Note: Regular expressions can be used for most of the arguments in a point. They must be prefaced with an exclamation point. For more information about using regular expressions, see “Using Regular Expressions” in the Diagnostics Server Installation and Administration Guide.

Optional Point Entries

Point definitions can contain one or more of the following arguments:

Argument	Description
keyword	The **keyword** indicates an instrumentation point handled by a special instrumentation class. The value of the keyword is looked up as a property in **inst.properties**, and the value of the found property is the instrumentation class name. The special instrumentation points can use implementation-specific arguments not documented here, refer to the comments in the inst.properties file.
ignore_cl	Specifies a comma-separated list of class names or regular expressions to ignore. Any class matching one of the classes specified with **ignore_cl** is not instrumented.
ignore_method	Specifies a comma-separated list of methods to ignore. Any method matching one of the methods specified with **ignore_method** is not instrumented.
Ignore_tree	A list of classes or regular expressions. Any subclass of a class matching one of the list items is excluded from the instrumentation.

Argument	Description
method_ access_filter	A list of method specifiers, separated by commas. The available specifiers are **static**, **private**, **protected**, **package**, and **public**. Any method matching this point is not instrumented if its access specifier matches any of the listed values.
deep_mode	Specifies how subclasses are handled. This argument accepts three values: • **none** – A value of “none” is similar to not specifying a **deep_mode** argument. The instrumentation point applies only to the specified class and has no effect on how subclasses are handled. • **soft** – A value of “soft” requests that for every class or interface matching the class, method, and signature entries, any subclasses or subinterfaces at any depth that also implement the matching method and signature should also be instrumented. • **hard** – A value of "hard" means that the instrumentation point applies (in addition to the specified class) to **all** methods from **all** classes extending (or implementing) the specified class, wherever the method matches the instrumentation point specification (both method name and signature). Hard mode is typically used for points for interfaces. **Caution:** Hard mode can lead to extensive instrumentation and very high probe overhead. **Note:** Since deep_mode looks at the class hierarchy, it cannot be used for instrumentation points based on annotations.
scope	Constrains the context in which instrumentation is performed. If specified, the inserted bytecode will be caller side. Any valid regular expression can be used for the value of this argument. Scope values are a comma-separated list of package, class, and method names in standard Java notation.
ignoreScope	Lists method names or regular expressions and excludes certain packages, classes, and methods from those included in the scope specified in the scope argument.

<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>detail</td><td>Specifies more specific capture instructions. It is a comma-separated list of the following:

caller – causes caller side instrumentation to be performed. If this keyword is not specified, the default instrumentation, callee side instrumentation, is performed.
args:n – calls the toString() method of the n-th argument. The string that is returned is displayed in the method's argument field in the Diagnostics console. The captured string can be used as the aggregation parameter in the layer argument. The value for n can be 1 through 256.
args:0 – calls the toString() on the current class instance or callee object. Static methods return the class name of the callee object.
before:code:<code-key> – inserts the code-snippet specified in the key at the start for the bytecode for methods that comply with the point. The final string value on the stack when the code-snippet runs is displayed in the method's argument field in the Diagnostics console and can also be used as the aggregation parameter in the layer argument. The code-key argument specifies the secure code key you generated for the code snippet you created for the point. See "Defining Points With Code Snippets" on page 103 for information about code snippets and "Securing Code Snippets" on page 112 for information on code keys.
after:code:<code-key> – inserts the code-snippet specified by the key at every exit point from the bytecode of methods that comply with the point. The after code-snippets should not leave any values on the stack after they run.
disabled – prevents the instrumentation inserted into the bytecode from reporting data. A disabled point can be dynamically enabled using the Instrumentation control web page so that it will begin reporting data. This web page can be accessed using the Profiler URL

http://<agent_install_directory>:<probe_port>/inst/layer.

outbound – flags the method so it is listed on the Outbound Calls screen. Also causes the Diagnostics argument for this instrumentation entry to be parsed to determine if additional information about the outbound request can be displayed in the Diagnostics dashboards.
no-correlation – used with those “outbound” points that do not use correlation supporting technologies.
method-no-trim – indicates that no latency-based trimming should take place when a method instrumented by this point is executed.
method-trim – indicates that every invocation of the method instrumented by this point should be “trimmed”, that is, not reported. However, side-effects of the corresponding code-snippets, if any, take place normally.
lifecycle – identifies the instrumentation point as relevant for object lifecycle monitoring.
no-layer-recurse – prohibits recording of any methods called from the method instrumented by this point, unless the callee belongs to a different layer.
is-statement – marks calls into the java.sql.Statement class.
is-prepare-statement – marks calls returning java.sql.Statement objects to capture.
method-cpu-time – causes the CPU inclusive time to be collected for this method in addition to latency, unless CPU collection is completely turned off
</td></tr>
</table>

Argument	Description
	(cpu.timestamp.collection.method = 0). • **condition** – prohibits instrumentation by this point unless the specified condition is met. The conditions are static and are defined by the details.conditional.properties property in **inst.properties** (or on the command line). • **when-root-rename** – instructs the probe to rename the server request whenever the method instrumented by this point is the first one executed. • **add-field:<access>:<type>:<name>** – causes adding the specified field to the instrumented class. • **gen-instrument-trace** – causes printing of the thread stack trace onto stdout whenever this point is used for instrumentation. • **gen-runtime-trace** – causes printing of the thread stack trace onto stdout whenever the methods instrumented by this point are executed. • **trace** – causes printing of verbose instrumentation information into probe.log on each enter or exit from each method instrumented by this point. • **sub-point:<key>** – specifies additional processing during instrumentation; the key must be present in **inst.properties** and must identify a class name used for the processing. • **store-thread –** causes all special fields used in the corresponding code-snippet to be stored in a thread-local data structure. • **store-fragment** – causes all special fields used in the corresponding code-snippet to be stored as attributes of the current server request. • **store-method** – causes all special fields used in the corresponding code-snippet to be stored as attributes of the invocation of the method instrumented by this point. • **ws-operation** – specifies that the instrumentation entry is for an inbound web services call. Also causes the Diagnostics argument for this instrumentation entry to be parsed to determine if additional information about the web service request can be displayed in the Diagnostics dashboards.
rootRenameTo	Identifies server requests whenever the **when-root-rename** detail is in effect.
layerType	Specifies special handling for some instrumented methods and accepts the following values: • **method –** no special handling (default). • **trended_method** – identifies methods to be displayed in the Trended Methods view. • **Portlet** – identifies portlet lifecycle methods that are used for the Portal Components views. These are set by Diagnostics and should not be modified. • **sql** – identifies methods that are used to capture SQL for the SQL views. These are set by Diagnostics and should not be modified.
priority	Whenever there is more than one instrumentation point that can be applied to a given method, and the Diagnostics Agent cannot resolve the conflict on its own, the point's **priority** determines which point to use. Higher priority wins. The default is zero.
active	Activates or deactivates a point. When set to true, the point is activated. When set to false, the point is inactive and is ignored by the probe.

Defining Points With Code Snippets

Custom code arguments specify a snippet of code that is to be inserted into the bytecode for a point. Code snippets in a point are used when the value returned by calling an object's **toString()** method, as specified in the **args:n** argument, is not going to provide useful information for the Diagnostics console or when there is a requirement to display more than one argument for an instrumented method.

A code snippet in a point is declared using the keyword **before:code:<code-key>** or **after:code:<code-key>** in the detail argument of the point. The before and after is used to execute the code snippet before or after the instrumented method. The code snippet is typically secured using a code-key argument to prevent unauthorized modifications of the code snippet. The values for the code-key arguments can be generated using any running probe's code-key generator page and are valid on any Java Agent installation. For more information on the code-key see "Securing Code Snippets" on page 112.

The actual code snippets for a point are entered into the **<agent_install_directory>/etc/code/custom_code.properties** file. These snippets are then associated with the point in the capture points file using the value of the code-key. Code snippets are created using pseudo Java code that uses syntax similar to OGNL. Using code snippets, calls can be made from the instrumented bytecode to methods that can be accessed by the instrumented method. Objects returned by code snippets can be cast and can have their methods executed as well. Code snippets must end with a string or an object where **toString()** can be left on the stack of statements being parsed into bytecode. This final string of the code snippet is used for the returned argument value displayed in the Diagnostics console.

Code snippets can also be used to store values for a particular fragment directly or that could be used in a later code snippet. These features can be used through special fields and key word details like **store-fragment** and **store-thread**.

> **Note:** Code snippets are a very powerful tool that should be used carefully because of the potential impact to the overhead incurred by the probe. For this reason, Diagnostics requires that a code-key be specified along with the code snippet before the probe will use the code snippet during instrumentation.

This section includes:

- "Using Code Snippets" below
- "Code Snippet Grammar" on the next page
- "Code Snippet Helper" on page 106
- "Securing Code Snippets" on page 112

Using Code Snippets

To use code snippets when specifying a point in **<agent_install_directory>/etc/auto_detect.points**, the following detail:

```
class = javax.jms.TopicPublisher
method = publish
signature = !\(Ljavax/jms/Topic.*
deep_mode = soft
layer = Messaging/JMS/Producer
detail = outbound,no-correlation,before:code:6d0f3088
```

The before:code entry in the detail argument indicates that a code snippet was entered for the point. The code-key value secures the code in the code snippet and ties the point with the actual code snippet.

The code snippet associated with the point must be entered in **<agent_install_directory>/etc/code/custom_code.properties** as shown in the following example:

```
# Used by [JMS-TopicPublisher2]
6d0f3088 = #topic =
@ProbeCodeSnippetHelper@.checkForTempName(#arg1.getTopicName()); \
"DIAG_ARG:type=jms&name=topic:"+ #topic + "&target=topic://" + #topic;
```

The code snippet is associated with the point in the capture points file using the value of the code-key.

Code Snippet Grammar

The following describes the syntax that must be used to create the code snippets.

Literals

Only the following literal types are supported in code snippets.

Literal Type	Syntax Example
string	"a string"
boolean	true, false
integer	42
null constant	null
a no-type, no-value constant	void

String concatenation

Basic string concatenation is supported in code snippets.

Concatenation Type	Syntax Example
Two strings	"a string" + "another string"
A string and a literal	"a string" + 42

Local members

Default local members provide a way for code snippets to reference the current instance or objects that were passed to the instrumented method. These local members call methods or retrieve values from those references.

Variable	Use
#callee	References the callee object for an instance method. Equivalent to the java “this” reference. Must not be used when referencing a static method.
#arg1, #arg2, ..., #argN	References the arguments for the callee method call.

Variable	Use
#return	References the return value of the method end for after code snippets.
#classloader	Reserved for Software internal use.

Note: Some instrumentation points support *special* variable references. For example, the **CLApplicationDiscoveryPoint** supports a #classloader variable.

DIAG_ARG strings

Code snippets allow concatenation of a series of values building up a single DIAG_ARG value. This value allows for instrumentation of some common types of support data like Web Services and JMS by returning all the data for a particular type in one DIAG_ ARG formatted string.

Type	Field (Required)	Definition
ws	&ws_name	Web Service name
	&ws_op	Web Service Operation name
	&ws_ns	Web Service namespace
	&ws_port (inbound only)	Web Service Port Name
	&target (outbound only)	Outbound Web Service Target
jms	&name	Queue or Topic name
	&target	Target Queue or Topic name

The format of the DIAG_ARG string includes the type fields and values (local variables) concatenated into one string as follows:

```
"DIAG_ARG:type=ws&ws_name="+ #servicename +"&ws_op="+ #operation +\ "&ws_ns="+ #ns +"&ws_
port="+ #port;
```

The DIAG_ARG string must not be used in combination with the store-fragment special fields for web service inbound data (special fields starting with ##WS_inbound_*). Use ONLY one for collecting web service inbound data.

Special fields (store-fragment)

Default special fields provide an easy way for code snippets to pass fragment-related data for common events. This mechanism supplements the existing events, but is not expected to replace them. Fragment Local Storage has higher overhead cost than custom events. The following variables must be used with the **store-fragment** detail setting.

Variable	Use
##WS_consumer_id	Stores the consumer Id for a particular fragment.
##WS_SOAP_fault_code	Stores the SOAP fault code.
##WS_SOAP_fault_reason	Stores the SOAP fault reason.
##WS_SOAP_fault_detail	Stores the SOAP fault detail.

Variable	Use
##WS_inbound_service_name	Stores the inbound web service name.
##WS_inbound_operation_name	Stores the inbound web service operation name.
##WS_inbound_target_namespace	Stores the inbound web service target namespace.
##WS_inbound_port_name	Stores the inbound web service port name.

Special fields (store-thread)

Additionally special fields provide an easy way for code snippets to store related data for the life of the thread. Use these thread local storage variables with caution because they have overhead associated with them. Use them only with the store-thread detail setting.

These variables can be retrieved in later code snippets by making a call to the probe’s ThreadContextProxy class reference with either the getThreadContextValue(“string value”) or getAndRemoveThreadContextValue (“string value”) methods, with “string value” being the name of the variable without the leading ## signs. The last retrieval of the value should always call getAndRemoveThreadContextValue(“string value”) to clear the value from memory and to remove the value for the next thread.

Variable	Use
##SOAPHandler_wsname	Stores the web service name for later use by the SOAP Handler.
##<any_string>	Stores any value for later retrieval in a following code snippet.

Class references and static members

Static members/methods can be accessed by pre-pending the class with an @ symbol to identify it as a Static, and marking the method being accessed with an @ symbol as in the examples below:

```
@java.lang.System@.out ("Hello World");

@com.mercury.diagnostics.capture.metrics.countingCollector@.incrementCounter();
```

The arguments in the code snippets support Java class syntax when the Java class is surrounded with a marker that the parser can get hold of. The following examples show how to use the @ symbol as a marker:

```
@java.lang.System@

@java.lang.System@out (Static field)
```

Code Snippet Helper

Some functionality is very hard, or even impossible, to code using the limited syntax available within the code-snippets. Therefore, the code-snippet environment offers two helper classes:

- ProbeCodeSnippetHelper
- ProbeCodeSnippetHelperV5.

The following shows ProbeCodeSnippetHelper functionality.

```
/*
```

```
* (c) Copyright 2008 Hewlett-Packard Development Company, L.P.
*/
package com.mercury.opal.capture.proxy;
/**
* Used to help out Code Snippets
*/
public class ProbeCodeSnippetHelper {
/**
* When a Special Field holds a reference to the string below,
* it will not be stored in the Fragment Local Storage,
* or Invocation Local Storage
*/
public static final String DO_NOT_STORE = ...
/**
* Helper to convert an int to an Integer
* @param i
* @return a new Integer object having the value of i
*/
public static Object intToInteger(int i) {
...
}
/*
* Mark the current thread, if not marked yet
* @return true, if and only if the thread had been already marked
*/
public static boolean testAndSetRecursiveFlag() {
...
}
/*
* Unmark the current thread
*/
public static void clearRecursiveFlag() {
...
}
/**
* Helper method to call ResourceBundle.getString() and catch any exceptions that
* might be thrown
* @param theBundle the ResourceBundle on which to call getString
* @param key the key to pass getString
* @return the value returned from getString, or null if an exception occurred
*/
public static String getStringFromResourceBundle(ResourceBundle theBundle, String
key) {
...
/*
* Helper methods to allow our cross-vm coloring to piggyback ride across
* the custom outbound calls in which the application passes [only] a String.
* The actual transport technology is irrelevant.
* Instead of sending the original message, a composite message ("envelope")
```

```
* will be passed. The composite message includes both the original message
* and Diagnostics Probe ENCODED cross-vm coloring.
* On the receiving end, the composite message will be received, but only
* the original message will be passed to the application, and the coloring
* will be retained by the probe.
*/
/**
* Create a composite message, given the coloring and the original message.
* @param coloring - the correlation String obtained via the ENCODED coloring,
* may be null
* @param originalMessage - the original messsage sent by the application
* @return - the composite message, null if and only if the originalMessage is null
*/
public static String createDiagEnvelope(String coloring, String originalMessage) {
...
}
/**
* Extract the coloring from the composite message (envelope).
* @param envelope - the composite message or the original message
* @return the coloring as created on the sender side, or null if not present
*/
public static String extractColoringFromDiagEnvelope(String envelope) {
...
}
/**
* Extract the original message from the composite message (envelope).
* Works properly, even if the sender side has not been instrumented, and
* there's no envelope.
* @param envelope - the composite message or the original message
* @return the original message (before coloring)
*/
public static String extractOriginalMessageFromDiagEnvelope(String envelope) {
...
}
}
}
```

The following shows ProbeCodeSnippetHelperV5 functionality.

```
/*
* (c) Copyright 2008 Hewlett-Packard Development Company, L.P.
*/
package com.mercury.opal.capture.jdk15.agent;
/**
* Used to help out Code Snippets using Java 5 or later
*/
public class ProbeCodeSnippetHelperV5 {
/**
```

```
* Get the annotation of the specified type from the class or its superclass,
* or its implemented interfaces
* @param theClass The class to get the annotation for
* @param annClass The annotation class to look for
* @return
*/
public static Object getEndpointClassAnnotation(Class theClass, Class annClass) {
...
}
/**
* Get the method annotation of the specified type from the class
* or its superclass, or its implemented interfaces
* @param theClass the class
* @param methodName the method name
* @param argCount the argument count
* @param annClass the class annotation type
* @param methodAnnClass the method annotation type
* @return
*/
public static Object getEndpointMethodAnnotation(Class theClass, String methodName,
String argCount, Class annClass, Class methodAnnClass) {
...
}
/**
* Helper method to get an annotation element value. If the annotation
* does not have the element, return null.
* @param annClass The class of the annotation
* @param instance The annotation instance object
* @param elementName The element name
* @return The element value for the annotation instance, or null
*/
public static String getAnnotationElementValue(Class annClass, Object instance,
String elementName) {
...
}
/**
* This helper method is used to serialize a DOM document.
* This method uses APIs available in DOM Level 3 or newer, which are
* available with a 1.5 or later JVM.
* @param document
* @return The serialized form (XML) of the input DOM document
*/
public static String serializeDOMToString(Document document) {
...
}
}
```

Spanning multiple lines with the stack of method calls

The stack of method calls in a code snippet can span multiple lines. The parser that builds the bytecode requires a “\” (backslash) before each carriage return when it must continue parsing the stack of statements. The final line of the Code Snippet stack of statements should not contain a backslash and should simply end with carriage return.

```
@java.lang.System@.out ("Hello World");\

"Callee Name="+#callee.getName().toString();
```

Casting

When calling a method that returns an object, casting is typically required to call members on the returned object. Casting is supported on object references. To cast an object to another type, place the casting reference between the symbols “<“ and “>” following the reference to that object. The following are examples of casting.

```
#arg1<com.myCompany.myFoo>.myMethod();

This is equivalent to the Java statement:

((com.myCompany.myFoo)arg1).myMethod();


@some.class.Foo@foo<com.myCompany.myFoo>.myMethod();

Would be equivalent to the java statement:

((com.MyCompany.myFoo)some.class.Foo.foo).doSomething();


#foo = #arg1<bar>.b(); #foo.toString();

Creates the following java equivalent:

String foo = ((Bar)arg1).b(); ((Object)foo).toString();
```

Note: Casting is not supported for special types such as #classloader.

Method calls

Method calls can be included in snippet arguments. The support of method calls includes calls with or without arguments and method chaining. The following are examples of method calls that are included in code snippet arguments:

```
#arg1.toString()

#arg2.getSomething().getSomethingElse()

#callee.getSomething("foo", #arg1).somethingElse()

@some.Class@.staticMethod()
```

The dot still needs to appear after the static reference for the method call to be parsed properly.

```
@java.lang.System@out.println("Here I am!")
```

To speed up the generation of bytecode at runtime (by avoiding reflection), you can specify the type that is returned from a method as shown in the following example:

```
#arg1.getSomething()<some.class.Here>
```

This will not help if the method takes arguments, or if a static field is used.

Multiple statements

Code snippets can include multiple statements in a single code snippet. This is necessary for instrumentation, such as **CLApplicationDiscoveryPoint**, that expect multiple objects to be left on the stack. It can be handy in other situations as well.

```
@java.lang.System@out.println("Look out!");

#arg2.getSomething();
```

Local Member assignment

In addition to the default local member variables, you can create your own local members to hold object references returned by called methods.

To create a new local member, enter the "#" symbol before the name of the local member. The parser will create the local member. Once a local member is assigned a value it cannot be overwritten; simply create a new variable if you need to re-assign to a local member.

```
#myBar = #arg2.getName();\
#myUpperBar = #myBar.toUpper();\
"Target Name=http://"+myUpperBar+"/services";
```

Special Field assignment (store-fragment)

You can use a pre-defined special field to store the object references returned by called methods. Enter the "##" symbols before the name of the special field along with the **store-fragment** detail keyword on the instrumentation point.

```
##WS_SOAP_fault_code = #arg2;\
##WS_SOAP_fault_reason = #arg3;\
##WS_SOAP_fault_detail = (#arg4 == null ? null : #arg4.toString());"";
```

Special Field assignment (store-thread)

You can use a special field to store the object references returned by called methods. Enter the "##" symbols before the name of the special field along with the **store-thread** detail keyword on the instrumentation point.

```
# Used by [SOA_Broker_Payload_Handler]
##SOA_Manager_Inbound_Payload=#callee.getRequestDocument();"";
```

In a later code snippet you can retrieve the value stored by calling getThreadContextValue with the special field value above without the leading ## symbols.

```
#temp_soam_
payload=@com.mercury.opal.capture.proxy.ThreadContextProxy@.getThreadContextValue("SOA_
Manager_Inbound_Payload");
```

In a later code snippet you can retrieve and remove the special field value stored by calling getAndRemoveThreadContextValue method with the value same above without the leading ## symbols. It is very important that you call getAndRemoveThreadContextValue to free memory and clear the way for the next occurrence.

```
#temp_soam_payload=@com.mercury.opal.capture.proxy.ThreadContextProxy@.
getAndRemoveThreadContextValue( ("SOA_Manager_Inbound_Payload");
```

Conditional Logic

Code snippet syntax allows for limited conditional logic that is equivalent to the Java if-else statement. This syntax enables you to compare object references of the same type or integer or boolean primitives using both the == and != operators. Literal value and other primitive comparisons are not valid using this syntax.

The following is an example of how to compare references:

```
(value1 == value2 ? <if_True_codeSnippet>:<if_False_codeSnippet>)
```

The following is an example of how to verify that an object is not null before calling a method:

```
(#arg1 == null ? "Unknown" : #arg1.getSomething())
```

This would be equivalent to the following Java statement:

```
if (arg1==null) return "Unknown" else return arg1.getSomething();
```

Exception Handling

A limited form of exception handling is provided by the following syntax:

```
!{<code-snippet-code>}!
```

The specified code is executed and the value of the above expression is the thrown exception, or null if no exception was thrown during the execution of the code.

Securing Code Snippets

By default, you must specify a valid code-key along with the code snippet before the probe will use the code snippet during instrumentation. Requiring the code-key prevents accidently introducing instrumentation that could significantly increase the overhead of the probe.

When you generate the code-key, Diagnostics checks the syntax of the code snippet to make sure it is valid before it generates the key. When Diagnostics instruments an application, it checks the value entered for the code-key argument to make sure it matches the code-key it calculates for the code snippet for the point. If the code-keys do not match, Diagnostics ignores the code snippet and does not create the instrumentation point.

Generating the Code Snippet Code-Key

The Java Agent is installed with a tool that generates the code-key from the code snippet you input.

To generate a code-key:

1. Open the page at the following URL in your browser:

   ```
   http://<probe-host>:<probe-port>/inst/code-key
   ```

   Diagnostics displays the page where you can validate the code snippet syntax and generate the code-key as shown in the following example:

   Diagnostics

   This page provides you with the ability to validate a snippet of code for use in the probe's points file, as well as generate the required secure code-key.

   If a point's code does not match its key, the probe will refuse to use that code during instrumentation.

   Input your code snippet:

   Submit

   Resulting point section:

   Diagnostics J2EE Probe "WebLogic10_myd-vm930", version 9.30.6.269, pid 5380, profile 120

2. Enter the code snippet you specified in the code argument in the **auto_detect.points** file into the **Input your code snippet** text box and click **Submit**.

   The code snippet must include all of the text following the `code =` argument name.

3. Diagnostics presents the results of the code snippet validation and the code-key generation in the **Resulting point section** text box.

   If the code snippet is valid, Diagnostics displays the value of both the code-key and code arguments. Enter these values into the capture points file.

   If the code snippet is not valid, Diagnostics displays an error message that indicates the problem that was detected. Correct the problem and click **Submit** again to validate the corrected code.

Disabling the Code-Key Security Check

By default, Diagnostics verifies that the value of the code-key argument matches the value it generates when it is instrumenting the application. It is possible to disable this security check by inserting the **require.code.security.key** property into the **<agent_install_directory>/etc/code/custom_code.properties** file, under the **[Default]** section, with a value of `false`.

> **Note:** Be very careful when using this property. If you disable this check, you could experience unexpected processing overhead and unpredictable performance monitoring results.

Controlling Class Map Capture

The class map allows Diagnostics to provide more details about the classes and methods that are invoked by a server request. This information can help you to narrow your search for the source of a performance issue

and help you instrument the application code properly. The cost for using class map comes from the additional overhead that creating the map places upon the agent's host system.

By default the property **use.class.map=false** is set in the **probe.properties** file. Changing this to **true** provides a class map.

Instrumentation Examples

The examples in this section illustrate how you can customize the instrumentation of an application by creating and modifying the points in the capture points file.

This section includes the following examples:

- "Custom Layer and Sublayer" below
- "Wildcard Method" on the next page
- "Ignore Specified Methods" on the next page
- "Capture Methods for the Trended Methods View" on the next page
- "Capture Only a Specific Method In a Class" on page 116
- "Capture a Specific Method That Returns a String" on page 117
- "Capture with a Controlled Scope" on page 117
- "Hard and Soft deep_mode" on page 117
- "Argument Capture" on page 118
- "Inbound and Outbound Web Services" on page 119
- "Renaming Root Methods" on page 120
- "Adding a Field to a Class" on page 120
- "Passing Attributes to Instance Trees" on page 121
- "Filtering Out Methods by Their Access Flag" on page 121
- "Not Recording Direct Recursion" on page 121
- "Performing Caller Side Instrumentation" on page 121
- "Configuring Allocation Analysis" on page 122
- "Configuring Lightweight Memory Diagnostics (LWMD)" on page 122
- "Configuring Collection Leak Pinpointing" on page 123
- "Enabling Object Lifecycle Monitoring for JDBC Result Set" on page 123
- "Adding a Disabled Point and Enabling it at Runtime" on page 124
- "Specifying that a Method Should Never be Trimmed" on page 124
- "Specifying that a Method Should Always be Trimmed" on page 125
- "Enabling Collection of CPU Time for a Method" on page 125
- "Changing SAP RFC Instrumentation Based on SAP JCO Library Version" on page 125
- "Printing Instrumentation and Runtime Information for a Point (Debugging Only)" on page 125

Custom Layer and Sublayer

The following point creates a custom sublayer called "BAR" within the layer called "FOO" for the method myMethod in myCompany.myFoo class:

```
[myCompany.myFoo_customLayer]
class = myCompany.myFoo
method = myMethod
signature = !.*
layer = FOO/BAR
```

Wildcard Method

The following point captures all methods in the MyCompany.MyFoo class:

```
[myCompany.myFoo_AllMethods]
class = myCompany.myFoo
method = !.*
signature = !.*
layer = FOO/BAR
```

Ignore Specified Methods

The following point captures all methods in the MyCompany.MyFoo class except for the methods setHomeInterface and getHomeInterface:

```
[myCompany.myFoo_AllMethodsExcept]
class = myCompany.myFoo
method = !.*
ignoreMethod = !setHomeInterface.*, !getHomeInterface.*
signature = !.*
layer = FOO/BAR
```

The following point captures all methods in the MyCompany package/namespace except for those contained in the MyCompany.logging class:

```
[myCompany_All_Methods_except_from_MyCompany_Logging]
class = !myCompany\..*
method = !.*
ignore_cl = MyCompany.logging
signature = !.*
layer = FOO/BAR
```

Capture Methods for the Trended Methods View

The following point captures the required data to populate the Trended Methods View for the myMethod method:

```
[myCompany.myFoo_customLayer]
class = myCompany.myFoo
```

```
method = myMethod
signature = !.*
layer = FOO/BAR
layertype = trended_method
```

Capture Only a Specific Method In a Class

The following point captures all methods in the constructor for the MyCompany.MyFoo class:

```
[myCompany.myFoo_Constructor]
class = myCompany.myFoo
method = <init>
signature = !.*
layer = FOO/BAR
```

The following point captures all methods in the singleton constructor for the MyCompany.MyFoo class:

```
[myCompany.myFoo_Singleton]
class = myCompany.myFoo
method = <clinit>
signature = !.*
layer = FOO/BAR
```

The following point captures the setFoo method in the MyCompany.MyFoo class:

```
[myCompany.myFoo_setFoo]
class = myCompany.myFoo
method = setFoo
signature = !.*
layer = FOO/BAR
```

The following point captures all "set" methods in the MyCompany.MyFoo class:

```
[myCompany.myFoo_AllSets]
class = myCompany.myFoo
method = !set.*
signature = !.*
layer = FOO/BAR
```

The following point captures all methods in the MyCompany package/namespace:

```
[myCompany_All_Methods]
class = !myCompany\..*
method = !.*
signature = !.*
layer = FOO/BAR
```

Capture a Specific Method That Returns a String

The following point captures the getFoo method with no arguments that returns a java.lang.String in the MyCompany.MyFoo class:

```
[myCompany.myFoo_GetFoo_String]
class = myCompany.myFoo
method = getFoo
signature = ()Ljava\lang\String
layer = FOO/BAR
```

Capture with a Controlled Scope

The following point captures all methods in the MyCompany package/namespace that are called from the MyCompany.logging class. For more details see "Using Caller Side Instrumentation" on page 128.

```
[myCompany_All_Methods_from_MyCompany_Logging]
class = !myCompany\..*
method = !.*
signature = !.*
scope = MyCompany.logging
layer = FOO/BAR
```

The ignoreScope argument is used to exclude certain packages, classes, and methods from those included in the scope specified in scope argument. The following point captures all methods in the MyCompany package/namespace that are called from the MyCompany.logging class except for those called from the myMethod method. For more details see "Using Caller Side Instrumentation" on page 128.

```
[myCompany_All_Methods_except_from_MyCompany_Logging]
class = !myCompany\..*
method = !.*
signature = !.*
scope = MyCompany.logging
ignoreScope = MyCompany.logging\myMethod
layer = FOO/BAR
```

Hard and Soft deep_mode

The following interface definition is used for both soft and hard deep_mode examples:

```
public interface Interface1 {

        public void callerMethod();
}
```

The following class is used for both soft and hard deep_mode examples:

```
public class Class1 implements Interface1 {
  public void callerMethod(){
    calleeMethod();
    calleeMethod2();
  }
 public void calleeMethod(){
  System.out.println("hello world");
  //more code lines here…
  }
 public void calleeMethod2(){
  System.out.println("hello world 2");
 }
}
```

The following point captures the "callerMethod" in the Class1 class:

```
[Training-1]
class = Interface1
method = !.*
signature = !.*
deep_mode = soft
layer = Training
```

The following point captures all methods in Class 1 (for example, "callerMethod", "calleeMethod1" and "calleeMethod2):

```
[Training-1]
class = Interface1
method = !.*
signature = !.*
deep_mode = hard
layer = Training
```

Argument Capture

The argument displayed in Diagnostics is the final string left on the stack by the code snippet. Code snippets must end with a string or an object where toString() can be left on the stack of statements to be parsed to the bytecode.

Caution: Extreme caution has to be exercised when using argument capture. Unless the set of all possible values of the captured argument is finite, the agent will run out of Java heap space.

Suppose that you instrument a method called myCompany.myFoo.myMethod(), and myFoo has another method called getComponentName() that returns a String. The following example shows the result of getComponentName() as the argument in Diagnostics (#callee refers to the callee object for an instance method, in this case).

```
[myCompany_componentName_as_argument]
class = myCompany.myFoo
method = myMethod
signature = !.*
detail = before:code: 8d2509eb
layer = FOO/BAR
```

The code snippet in the **custom_code.properties** file is entered as follows:

```
8d2509eb = #callee.getComponentName()
```

The following point captures the first argument to myMethod and shows it as the captured argument in Diagnostics. It also uses it as the sublayer name. This is achieved by including ${ARG} in the layer. In this example, if the captured argument—in this case, the first argument of myMethod—has the value myArg, the layer is FOO/myArg.

```
[myCompany_capture_firstArg_and_also_show_as_layer]
class = myCompany.myFoo
method = myMethod
signature = !.*
detail = before:code: 358f05d6
layer = FOO/${ARG}
```

The code snippet in the **custom_code.properties** file is entered as follows. If you use #arg2, you would capture the second argument instead.

```
358f05d6 = #arg1.toString()
```

Inbound and Outbound Web Services

When the detail argument in a point contains the "outbound" or "ws-operation" keyword, Diagnostics attempts to parse the final string on the Code Snippet stack for additional information to display about the method call.

For inbound Web Services (“ws-operation” detail must be used), the string looks like the following:

```
"DIAG_ARG:type=ws&ws_name=”+<WebServiceName>+"&ws_op=”+
<OperationName>+”&ws_ns=”+<TargetNameSpace>+”&wsOport=”+<wsPort>
```

For outbound Web Services (“outbound” detail must be used), the string looks like the following:

```
"DIAG_ARG:type=ws&ws_name=”+<WebServiceName>+"&ws_op=”+
<OperationName>+”&target=”+<TargetName>
```

Here is an example:

```
class = weblogic.wsee.ws.WsStub
method = invoke
signature = (Ljava/lang/String;Ljava/lang/String;Ljava/util/Map;Ljava/util/Map;)
Ljava/
lang/Object;
layer = Web Services
detail = outbound,before:code:edd75e36
```

The code snippet in the **custom_code.properties** file is entered as follows:

```
edd75e36 = #service = #callee.getService().getWsdlService();\
#qname = #service.getName();\
"DIAG_ARG:type=ws&ws_name="+ #qname.getLocalPart() +"&ws_op="+ \
#callee.getMethod(#arg1).getOperationName().getLocalPart() +"&target="+ \
#callee.getProperty("javax.xml.rpc.service.endpoint.address");
```

Renaming Root Methods

Consider the following point:

```
class = Statement
method = execute
layer = Database/JDBC/Execute
detail = when-root-rename
rootRenameTo = mySuffix
```

If such a method ends up being the root method, the name of such a server request is Background-mySuffix, and the type of the server request is RootRename.

Consider the following point instead:

```
class = Statement
method = execute
layer = Database/JDBC/Execute
detail = when-root-rename
```

Notice that the rootRenameTo property is skipped. The name of such a server request is Background–Database (because Database is the first sublayer) and the server request type is RootRename again.

Adding a Field to a Class

Consider the following point:

```
class = com.corp.Foo
method = bar
detail = add-field:protected:Object:serviceName
```

The detail causes the following one field and two public setter/getter methods to be added to the class com.corp.Foo:

```
protected transient Object serviceName
public void _diag_set_serviceName(Object arg)
public Object _diag_get_serviceName()
```

Passing Attributes to Instance Trees

The following example attaches my_attribute name to every captured instance of com.corp.Foo.bar().

The name prefixed with display_ and its corresponding value is shown in the call profile.

```
class = com.corp.Foo
method = bar
detail = add-field:protected:Object:serviceName
```

Code snippet:

```
f59f0c5c = ##my_attribute="value-of-my-attribute";"";
```

Filtering Out Methods by Their Access Flag

The following example instruments all methods in class com.corp.Foo (but not static methods).

```
class = com.corp.Foo
method = !.*
signature = !.*
method_access_filter = static
```

Not Recording Direct Recursion

In the following example, if method com.corp.Foo.bar calls itself (or anything in the same layer), the second call is not recorded. This is caused by the **detail = no-layer-recurse**.

This, however, is only for direct recursion. If com.corp.Foo.bar calls an instrumented method from another layer that calls this method again, all methods are recorded.

```
class = com.corp.Foo
method = bar
layer = Example/MyBar
detail = no-layer-recurse
```

Performing Caller Side Instrumentation

The following point causes caller side instrumentation to be performed (as opposed to the default callee instrumentation). This is caused by the **detail = caller**.

Another way to do caller side instrumentation is to use the “scope” property as described in "Using Caller Side Instrumentation" on page 128.

```
class = com.corp.Foo
method = bar
detail = caller
```

Configuring Allocation Analysis

Both of the following examples track allocations of java.lang.Integer in the package com.mycompany.mycomponent. There are, however, two differences:

- In the first example **(detail = leak)**, tracking is managed. It starts when the user clicks **start** in the profiler and stops when the user clicks **stop**. In the second example **(detail = deallocation)**, tracking starts with application startup.
- In the first example, the point is disabled when it comes to regular instrumentation. This means you will not see “new Integer” show up on an instance tree. In the second example, you will.

Example 1 – Managed. Tracking starts when the user clicks **start** and stops when the user clicks **stop** in the profiler:

```
[Leak]
scope = !com\.mycompany\.mycomponent\..*
class = java.lang.Integer
keyword = allocation
detail = leak
active = true
```

Example 2 – Unmanaged. Tracking starts with application startup:

```
[Leak]
scope = !com\.mycompany\.mycomponent\..*
class = java.lang.Integer
keyword = allocation
detail = deallocation
active = true
```

Neither of these points captures reflected allocation. To enable reflected allocation capture, simply append the detail “reflection” to the point (**detail = leak,reflection**).

Configuring Lightweight Memory Diagnostics (LWMD)

The following example turns on collection diagnostics for collections that happened inside of the com.mercury.mycomponent package. You can find this example in the **auto_detect.points** file. It is set to active = false by default.

```
[Light-Weight Memory Diagnostics]
scope = !com\.mycompany\.mycomponent\..*
class = java.lang.Integer
```

```
keyword = lwmd
active = true
```

You also need to set the property **lwm.diagnostics.capture=true** in the **dynamic.properties** file. For more information, see the Diagnostics User Guide chapter on the "Collections and Resources View."

Configuring Collection Leak Pinpointing

Regardless of JRE version, you must run the JRE Instrumenter using the appropriate mode for your application server if you want to use the collection leak pinpointing (CLP) feature in the Java Agent. "Preparing Application Servers for Monitoring with the Java Agent" on page 30 for details on instrumenting the JRE.

In the **dynamic.properties** file you can set the following properties to configure collection leak reporting. These same values can also be set in the Java Profiler Configuration tab UI (see "Enabling and Configuring Collection Leak Reporting" on page 189).

clp.diagnostics.reporting=true

Enable reporting in the Diagnostics UI. You can disable reporting in the UI for this feature by unchecking the checkbox.

clp.diagnostics.growth.time.threshold.flag = 60m

The threshold of time duration in which the collection has size growth. If a collection's size growth period exceeds this threshold, it will be flagged as a memory leak by the probe. To avoid false positives, this value should be larger than the time required by your application to fully initialize all its caches.

clp.diagnostics.nongrowth.time.threshold.unflag = 60m

For an already flagged leaking collection, if its size stops growing continually for this threshold time period, the probe will unflag it as a leak.

Enabling Object Lifecycle Monitoring for JDBC Result Set

A few preconfigured instrumentation points allow object lifecycle monitoring but are disabled by default. Two of them are shown in the following example.

The example shows how to enable object lifecycle monitoring for JDBC Result Sets. For a more detailed discussion on object lifecycle monitoring, see "Object Lifecycle Monitoring" in the Diagnostics User Guide.

For this example, two actions are required:

1. Go to **inst.properties** and find details.conditional.properties. Set mercury.enable.resourcemonitor.jdbcResultSet=true
2. Specify the scope in the corresponding open instrumentation points (shown below).

   In the following, the probe performs object lifecycle monitoring for JDBC Result Sets inside package com.mycompany.mycomponent.

   ```
   [Lifecycle-JDBC-ResultSet-Open]
   scope = !com\.mycompany\.mycomponent\..*
   class = java.sql.Statement
   method = !(getResultSet.*)|(executeQuery.*)
   signature = !.*\)Ljava/sql/.*ResultSet;
   ```

```
detail = condition:mercury.enable.resourcemonitor.jdbcResultSet,lifecycle,caller
```

```
[Lifecycle-JDBC-ResultSet-Close]
class =
!(java\.sql\.ResultSet)|(weblogic\.jdbc\.wrapper\.ResultSet)|
(com\.ibm\.ws\.rsadapter\.jd
bc\.WSJdbcResultSet)
method = !(close)|(closeWrapper)
signature = !.*
deep_mode = soft
detail =
condition:mercury.enable.resourcemonitor.jdbcResultSet,before:code:513a2b36,metho
d-trim
```

Adding a Disabled Point and Enabling it at Runtime

In the following example, the point is disabled. This does not mean that instrumentation does not happen. Instrumentation happened but did collect any data. This significantly lowers the runtime overhead of such a point.

To enable data collection while the application is running, go to the Layer page in the (http://<probe-host>:<probe-port>/inst/layer or from the Profiler select the Configuration tab and then select View instrumentation), look for layer **myLayer**, and click **Enable**.

```
[My Example]
class = Example
method = !.*
layer = myLayer
detail = disabled
```

If you do not want instrumentation to happen at all, use **active=false**. However, such a point cannot be enabled at runtime.

Specifying that a Method Should Never be Trimmed

In the following example, latency trimming does not apply to Example.myMethod().

```
My Example]
class = Example
method = myMethod
detail = method-no-trim
```

Specifying that a Method Should Always be Trimmed

In the following example, the method Example.myMethod() is not reported. However, any code snippets associated with the point will always be executed.

```
[My Example]
class = Example
method = myMethod
detail = method-trim, before:code:...
```

Enabling Collection of CPU Time for a Method

In the following example, the detail “method-cpu-time” causes the CPU time to be collected for method Example.myMethod().

```
[My Example]
class = Example
method = myMethod
detail = method-cpu-time
```

Changing SAP RFC Instrumentation Based on SAP JCO Library Version

In the **<agent_install_directory>/etc/inst.properties** file there are two points defined depending on the version of SAP JCO used. Comment out the version you are not using. Starting with version 2.1.10 or later use com.mercury.opal.capture.inst.SapRfcinstrumentationPoint2_1_10. Otherwise the default setting will work for version 2.1.9 and earlier.

Printing Instrumentation and Runtime Information for a Point (Debugging Only)

The following example prints several pieces of debug information in standard out and probe.log.

- The **gen-instrument-trace** detail causes printing to stdout the thread stack trace whenever this point is used to instrument a method.
- The **gen-runtime-trace** causes printing to stdout the thread stack trace whenever Example.myMethod() is run.
- The **trace** detail causes printing in the probe.log verbose instrumentation information whenever Example.myMethod() is run.

```
[My Example]
class = Example
method = myMethod
detail = gen-instrument-trace, gen-runtime-trace, trace
```

Understanding the Overhead of Custom Instrumentation

When you are creating custom instrumentation, beware of over-instrumenting the application because it can introduce excessive latency into the probed application. Excessive latency arises from an increase in the classloader latency as more and more classes are instrumented. The custom instrumentation does not have the same impact on the method latency or the CPU overhead because the overhead of instrumentation is nearly fixed for every method because the amount of bytecode is almost always the same. This means that the physical percentages of the CPU and latency overhead will vary in direct proportion to the length of time the method takes to run.

For example, if a method takes 100ms, and instrumentation makes it run in 101ms, overhead is 1%. If a method takes 10ms and instrumentation changes its response to 11ms, overhead is 10%. If this method is not called very often, its overall latency effect on the application is minimal. However, the overall latency effect of an instrumented method that is called more frequently can affect the latency of the application’s response even though its overhead percentage is much smaller.

Unlike a traditional profiler, Diagnostics uses bytecode instrumentation. This allows the default instrumentation to be selective to minimize the overhead caused by instrumentation to an average of 3-5%. Methods with higher latency overhead introduced by instrumentation are only instrumented when they are called infrequently in relation to other components in the application and when the instrumentation provides specific information needed for triage activities (for example, JNDI lookups).

You should also consider Diagnostics data overhead when you are customizing the instrumentation for an application. The more methods you instrument, the more data the probe must serialize and pass over the network to the Diagnostics Server. You can tune the Java probe’s default configuration so that it can adjust the volume of Diagnostics data to avoid any unnecessary effect on the performance of the system being monitored. Improper tuning of a probe can cause CPU, Memory and Network overhead on the physical machine where the Java Agent is installed. For more information about managing Latency, CPU, Memory and Network overhead, see "Advanced Java Agent and Application Server Configuration" on page 158

Instrumentation Control on a Per Layer Basis

By default, the layers defined in the capture points file are enabled. If you include the `details=disabled` argument in a point, the layer is disabled when the probe is started.

The classmap provides the capability to dynamically instrument methods and classes using the JVMTI interface without restarting the JVM instance. All other virtual machines require that the JVM instance be restarted to apply changes you make to the capture points files.

Once instrumentation is placed within a method, its data collection and running CPU and method latency overhead can be controlled on a per layer basis (see the Instrumented Layers page below).

You can access the Instrumented Layers page from the URL:

http://<probe-host>:<probe-port>/inst/layer.

Diagnostics

Instrumented layers (no particular sorting)

Layer	Hits	Active Locations	Actions
(Other)	78	8 / 8	[Disable] [Clear # Hits]
(keyword) ejb30	0	0 / 328	[Enable] [Clear # Hits]
(keyword) http	61	13 / 13	[Disable] [Clear # Hits]
(keyword) remote-http	4	44 / 44	[Disable] [Clear # Hits]
Business Tier/EJB/Session Bean	0	0 / 11	[Enable] [Clear # Hits]
ClientSideMonitor/Instrumentation	0	0 / 3	[Enable] [Clear # Hits]
Database/JDBC	0	0 / 40	[Enable] [Clear # Hits]
Database/JDBC/Connection	31575	36 / 92	[Enable] [Disable] [Clear # Hits]
Database/JDBC/Execute	31579	47 / 47	[Disable] [Clear # Hits]
Directory Service/JNDI	0	0 / 5	[Enable] [Clear # Hits]
HttpStatus	50	16 / 16	[Disable] [Clear # Hits]
Java Server Faces/Lifecycle/Execute	0	0 / 2	[Enable] [Clear # Hits]
Java Server Faces/Lifecycle/Render	0	0 / 2	[Enable] [Clear # Hits]
Legacy/JCA/CCI	0	0 / 2	[Enable] [Clear # Hits]
Legacy/JCA/Connection	0	0 / 1	[Enable] [Clear # Hits]

To disable a layer from the Instrumented Layers page, click the **Disable** link associated with the layer on the page. The layer is disabled and the link toggles to **Enabled** so that you can enable the layer again when necessary.

Instrumented Location Throughput Throttling

In some cases, an instrumentation point instruments a method which is executed very frequently. This may significantly increase the probe overhead for the application thread and can also overload the probe by generating large amounts of data to process.

You can limit the number of events (instrumented method calls) per second that the probe monitors. The threshold, in events per second, is configurable, but when set applies to all instrumented points. The event counters are shared by all threads.

For instrumented points that reach the configured threshold, the probe attempts to provide the real throughput, in events/second, by recording this number in the probe.log. In the Diagnostics Enterprise or Profiler UI, the displayed metrics are for the number of method calls up to, but no higher than, the configured threshold.

To set the threshold:

1. Configure the required number (which must be a non-negative value) in the **location.maximum.throughput** parameter in the **<agent_install_directory>/etc/capture.properties** file.
2. Ensure that the **settings.override.authorization** parameter in the **<agent_install_directory>/etc/probe.properties** file is set to **true**.

For example, if the **location.maximum.throughput** parameter is set at 1000, when an instrumented method has been called 1000 times in a second, the probe stops collecting metrics for this method, although it does

keep counting the number of method calls in that second. The UI displays metrics for the first 1000 calls only and an entry may be written to the probe.log with the actual number of the method calls for that second.

Advanced Instrumentation Examples

This section includes:

- "Using Caller Side Instrumentation" below
- "Capturing HTTP Server Requests Based on Query Parameters" on the next page
- "CORBA Cross VM Instrumentation" on page 130
- "Using RMI Instrumentation" on page 131
- "Using Thread Local Storage to Store the SOAP Payload" on page 131
- "Performing Correlation Across Multiple Threads" on page 132
- "Using Fragment Local Storage to Store Web Service Field" on page 133
- "Using Annotations for Custom Instrumentation" on page 136

Using Caller Side Instrumentation

By default, all instrumentation in Diagnostics is called side instrumentation where the bytecode is placed within the method call. Caller side instrumentation refers to the process of placing the bytecode for measurement around the call to the method to be instrumented instead of within.

Caller side instrumentation allows finer control of instrumentation placement, but can increase application classloader time because each class specified in the scope must be checked for references to the class/method specified in the points.

A common use for caller side instrumentation is to instrument calls to methods in **rt.jar**. Classes loaded into the virtual machine using the bootclassloader and not from a conventional class loader cannot be directly instrumented. To instrument calls to these methods, you must use caller side instrumentation.

In the following example, the parse methods for the **javax.xml.parsers.SAXParser** and **javax.xml.parsers.DocumentBuilder** are instrumented by placing bytecode around the calls to parse in any (!.*) method from any class. Caller side instrumentation is required because both the **javax.xml.parsers.SAXParser** and **javax.xml.parsers.DocumentBuilder** classes are contained in the rt.jar and loaded into the virtual machine by the bootclassloader.

```
[XML-DOM-JDK14]
;---------- Interface --------------
Class = !javax\.xml\.parsers\.(SAXParser|DocumentBuilder)
method = parse
signature = !.*
scope = !.*
layer = XML
```

In the following example, instruments calls to javax.naming.Context's "lookup" method that are called from the com.myCompany.myFoo classes and places them in the JNDI sublayer in the FOO layer.

```
[JNDI-lookup-FOO]
```

```
;------------- Server side JNDI hook --------------
class = javax.naming.Context
method = lookup
signature = (Ljava/lang/String;)Ljava/lang/Object;
scope = !com\.myCompany\.myFoo\..*
deep_mode = soft
layer = FOO/JNDI
```

Note: To verify that the point has caused the bytecode to be properly placed, check the **<agent_Install_dir>/log/<probeName>/detailReport.txt** file for the entries Unique Header Name (that is, [JNDI-lookup-FOO]).

During final triage steps for a performance issue, it can be impractical to use the classmap and individual build points for every method called by a suspect area of the application. It is very common to use one or more levels of caller side instrumentation to identify the time spent within an individual method or methods that have a suspected bottleneck. This is a useful way to fill in the next level to a Call Profile in Diagnostics.

The following example instruments any call to a method that is performed within the **com.myCompany.myFoo** class by the "myMethod" method:

```
[MethodsCalledByFoo.myMethod]
class = !.*
method = !.*
scope = !com\.myCompany\.myFoo\.myMethod.*
layer = FOO/other
```

The following example also captures the arguments to any "set" method called in **com.myCompany.myFoo** class by the "myMethod" method:

```
[SetMethodsCalledByFoo.myMethod]
class = !.*
method = !set.*
scope = !com\.myCompany\.myFoo\.myMethod.*
detail = args:1
layer = FOO/other
```

Capturing HTTP Server Requests Based on Query Parameters

Applications typically use the same URL to access different workflow. If the application uses a URI argument (for example, `http://<myserver>/myApplication/Browse?Genre=metal`) to differentiate between the workflow, Diagnostics can be configured to parse and treat the different URIs as different server requests.

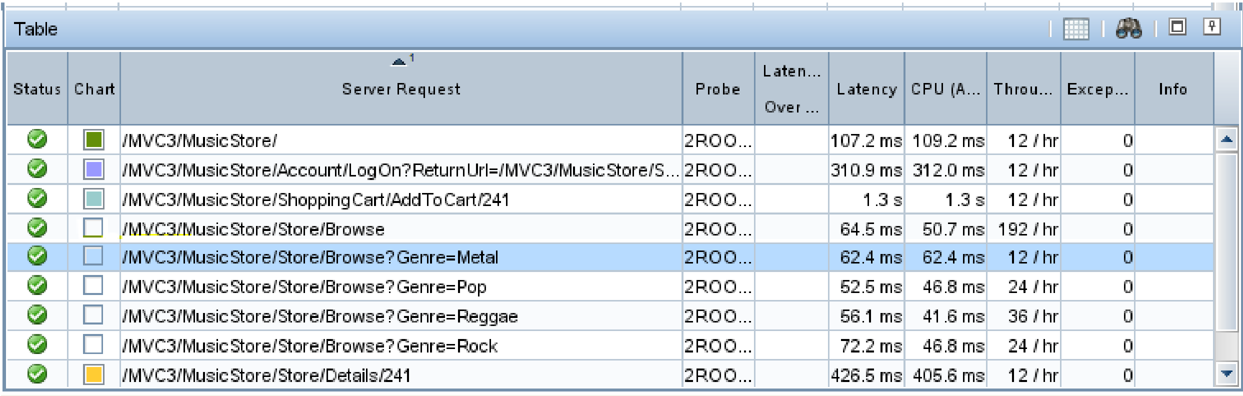
URI aggregation is controlled from the **[HttpCorrelation]** point. A valid regular expression entry for **args_by_class** should be created for each URI pattern.

For example, setting args_by_class as follows:

```
[HttpCorrelation]
args_by_class=!.*&Genre
```

results in the following ServerRequests appearing uniquely in the Diagnostics console:

```
http://<myserver>/myApplication/Browse?Genre=Metal
http://<myserver>/myApplication/Browse?Genre=Pop
http://<myserver>/myApplication/Browse?Genre=Reggae
http://<myserver>/myApplication/Browse?Genre=Rock
```



Table

Status	Chart	Server Request	Probe	Laten... Over ...	Latency	CPU (A...	Throu...	Excep...	Info
		/MVC3/MusicStore/	2ROO...		107.2 ms	109.2 ms	12 / hr	0	
		/MVC3/MusicStore/Account/LogOn?ReturnUrl=/MVC3/MusicStore/S...	2ROO...		310.9 ms	312.0 ms	12 / hr	0	
		/MVC3/MusicStore/ShoppingCart/AddToCart/241	2ROO...		1.3 s	1.3 s	12 / hr	0	
		/MVC3/MusicStore/Store/Browse	2ROO...		64.5 ms	50.7 ms	192 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Metal	2ROO...		62.4 ms	62.4 ms	12 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Pop	2ROO...		52.5 ms	46.8 ms	24 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Reggae	2ROO...		56.1 ms	41.6 ms	36 / hr	0	
		/MVC3/MusicStore/Store/Browse?Genre=Rock	2ROO...		72.2 ms	46.8 ms	24 / hr	0	
		/MVC3/MusicStore/Store/Details/241	2ROO...		426.5 ms	405.6 ms	12 / hr	0	

You can configure more than one URI parameter to be used for URI parsing in the args_by_class setting. For example:

```
args_by_class=!.*&Genre&Category
```

Note: Avoid using a session parameter or highly unique URI value because of the impact to overhead and data storage.

In a WebLogic environment, set the **use.weblogic.get.parameter=true** in **<agent_install_directory>/etc/inst.properties** when using URI aggregation to prevent URI aggregation from consuming the ServletRequest's inputstream.

CORBA Cross VM Instrumentation

The Common Object Requesting Broker Architecture (CORBA) standard enables components written in different computer languages and running on different systems to work together.

Instrumentation for correlating CORBA cross VM instance trees is provided in the **<agent_install_directory>\etc\auto_detect.points** file.

Follow these steps in to enable cross-VM instance trees for CORBA:

1. Uncomment the Corba cross-VM points in the **auto_detect.points** file.
2. Specify the following JVM argument at Application Server startup:

```
-
```

```
Dorg.omg.PortableInterceptor.ORBInitializerClass=com.mercury.opal.javaprobe.handl
er.corba.CorbaORBInitializer
```

3. Put the following jar file in the classpath:

```
<java-agent-install-dir>/lib/probeCorbaInterceptors.jar
```

Using RMI Instrumentation

The RMI (Cross-VM) point in the capture points file is inactive by default. You must activate this point to capture the cross-vm processing in the application. If you have Java probes with this point activated on both sides of an RMI call, Diagnostics can correlate the call tree data from both virtual machines.

```
[RMI]
keyword = rmi
layer = CrossVM
active = false
```

RMI Instrumentation In a Clustered Environment

The **weblogic.t3.rmi** property in the **<agent_install_directory>/etc/inst.properties** file controls how the RMI instrumentation captures Cross-VM RMI performance metrics. By default, **weblogic.t3.rmi** is set to `false`, which causes the performance metrics to be gathered using the generic RMI instrumentation. In a clustered environment, all servers in a cluster must have RMI instrumentation turned on to avoid application failure when **weblogic.t3.rmi** is set to `false`.

When **weblogic.t3.rmi** is set to `true`, the generic RMI instrumentation is disabled, and the RMI Cross VM is captured using only WebLogic's T3 protocol. This allows the Java probe to function with only some of the servers in a cluster probed with RMI instrumentation enabled.

Using Thread Local Storage to Store the SOAP Payload

The following example demonstrates usage of thread local storage. In particular, it shows how to store (and clean) the SOAP payload from thread local storage. SOAP payload is captured by default only for certain application servers. For more information on the support matrix, see "Configuring SOAP Fault Payload Data" on page 182.

The following example is applicable only for application servers where Diagnostics does not capture payload out of the box.

First, it is necessary to identify where to access the payload from. Assume that the payload is the second argument of a method called DispatchController.dispatch().

The keyword store-thread causes the Java probe to store the special fields in the corresponding code snippet (in this case, My_Inbound_Payload) into thread local storage. This can be referenced from a different code snippet provided both points are hit on the same thread. Looking up the payload is demonstrated in the next example ("Using Fragment Local Storage to Store Web Service Field" on page 133).

```
[MyAppServer-SoapPayload-Capture]
```

```
class = com.myCompany.DispatchController
method = dispatch
signature = !\(Ljava/lang/Object;Ljava/lang/Object;\).*
layer = Web Services
detail = before:code: ae7f0a58,store-thread

# Used by [MyAppServer-SoapPayload-Capture]
ae7f0a58 = ##My_Inbound_Payload=#arg2;"";
```

Performing Correlation Across Multiple Threads

Asynchronous Server Requests are server requests that switch threads between server request start and end events. In the most simple case, one thread receives the request, partially processes it, and then hands it off to another thread to complete processing and to send the response back to the requesting party.

Diagnostics offers two operations, available through code snippets, to allow the Java agent to perform correlation across multiple threads:

- parkFragment(Object anchor)

  This operation is executed to indicate that the current thread will no longer run the current server request. All method invocations, as recorded by the Java Agent, are artificially terminated at this point. This is to indicate that even though some of these methods will continue execution, their activity will have nothing to do with the current server request. Furthermore, even if the current thread will invoke some instrumented methods after calling parkFragment, these calls will not be reported. The server request is no longer considered running, and the specified object (anchor) is used by the application as a unique identification of the server request to be resumed later (presumably, by another thread).

- resumeFragment(Object anchor)

  This operation is executed to indicate that the current thread resumes execution of previously parked server request. The argument (anchor) is used to identify the server request. All active method invocations will have their start time artificially reset to the current time. This is to indicate that even though some time may have elapsed while these method were executing, their execution had nothing to do with the server request being resumed. If the current thread was already running a server request, it will be ignored (dropped).

When using these operations, it is essential that the correct anchor object, as well as the correct thread switching points are identified by the application specialist.

Beware of race conditions: if the fragment is reported "parked" too late, after the corresponding resume operation is performed, the fragment will get lost (and a warning will appear in probe.log). Two useful techniques to avoid the race condition are: first, calling parkFragment slightly before the current thread really abandons the server request, and second, try to piggyback the application built-in synchronization which is often used to hand off an object from one thread to another.

A "parked" fragment can be seen using the pending-fragment servlet, as "PARKED SERVER REQUEST" displayed in place of the currently running method.

The feature usually requires you to identify the thread switching points in the monitored application, and to provide the corresponding instrumentation points with code snippets. Out of the box support is provided for BEA AquaLogic.

Examples of two instrumentation points with the corresponding code snippets are presented below. They are a part of the AquaLogic support.

The first point presented below is executed whenever AquaLogic sends a sub-request to another server. The instrumented method, PipelineContextImpl.dispatch(...) returns true if the sub-request was successfully sent. The thread sending the sub-request does not wait for a response, but proceeds to pick up the next server request from a pipeline.

Therefore, the code snippet examines the return value, and if it is true, signals to the probe that the current server request will be suspended. The server request is identified by a MessageContext object, which AquaLogic creates for every incoming server request.

```
[BEA_ALSB_AsyncDispatch]
; instrumentation point for AquaLogic Service Bus asynchronous dispatch
class = com.bea.wli.sb.pipeline.PipelineContextImpl
method = dispatch
signature = !\(Lcom/bea/wli/sb/context/MessageContext;.*
detail = after:code:549b1b59
layer = Service Bus/AquaLogic

# Used by [BEA_ALSB_AsyncDispatch]
# Asynchronously dispatches a subrequest for a service, the response will be
# processed on another thread
549b1b59 = (#return == true ?
@ThreadContextProxy@.parkFragment(#location,#arg1) : void);
```

Upon receiving a response from the sub-request, AquaLogic executes RouterCallback.onReceiveResponse (...), possibly on another thread. The processing of the original server request resumes, and this is signaled to the probe by the code snippet.

In this case, the MessageContext object representing the server request is not available as an argument of the instrumented method and needs to be extracted from the RouterCallback object.

```
[BEA_ALSB_ProxyService_Callback_Response]
; instrumentation point for AquaLogic Service Bus callback function
class = com.bea.wli.sb.pipeline.RouterCallback
method = !(onError)|(onReceiveResponse)
signature = !.*
layer = Service Bus/AquaLogic
detail = before:code:dba72078

# Used by [BEA_ALSB_ProxyService_Callback_Response]
# Resume processing of a server request when the reply for a subservice comes back
# (or when the server request was moved to the response pipeline internally)
dba72078 =
@ThreadContextProxy@.resumeFragment(#location,#callee._context.getMessageCon
text());"";
```

Using Fragment Local Storage to Store Web Service Field

The following example demonstrates several features of points and code snippets:

- How to use fragment local storage to store web service-specific fields (ws_name, ws_op, and so on). This is an alternative to specifying the “DIAG_ARG” string.

- How to retrieve (and remove) the stored payload from thread local storage (which was stored in the previous example).
- How to extract the consumer ID out of the SOAP payload.
- How to use fragment local storage to store the consumer ID.

Because web services are treated in a special way, several fields must be captured. These fields are described in "Code Snippet Grammar" on page 104.

The first step is to find the instrumentation points that will give access to the required fields (Web Service name, operation, namespace, port name). Suppose that there is a single class in the instrumented application that has access to all these fields. Assume that this class is called com.myCompany.MyWSContext. We need to access an instance of this class when all the above fields are set. There can be many options. Suppose that one such option is when MyWSContext is passed as the first argument of a method MyWSFactory.create(). This is the method we want to instrument.

Here is our instrumentation point (each line is explained below):

```
class = com.myCompany.MyWSFactory
method = create
signature = !\(Lcom/myCompany/MyWSContext;.*
layer = Web Services
detail = ws-operation, before:code: f334f0b4,store-fragment
```

The first three lines of the point shown above cause the probe to instrument anything that matches com.myCompany.MyWSFactory.create(MyWSContext, *).

The fourth line specifies the layer for this point.

The fifth line provides the probe with additional information about this point (details):

- The first detail (ws-operation) is important because it causes the probe to treat this as an inbound Web Service.
- The second detail (before:code: f334f0b4) causes the probe to insert the corresponding code snippet at the start of the methods that comply with this point. The actual code snippet is shown below. The number f334f0b4 was generated by going to

  `http://<probe-host>:<probe-port>/inst/code-key` and pasting the code snippet in the text box.
- The third detail (store-fragment) causes the probe to store all special fields (##) found in the corresponding code snippet as attributes of the server request.

Here is the corresponding code snippet (each line of the below code snippet is explained below).

```
f334f0b4 = #wsContext=#arg1;\
##WS_inbound_service_name=#wsContext.getServiceName();\
##WS_inbound_operation_name=#wsContext.getOperationName();\
##WS_inbound_target_namespace=#wsContext.getNamespaceURI();\
##WS_inbound_port_name=#wsContext.getEndpoint();\
#soap_payload =
@com.mercury.opal.capture.proxy.ThreadContextProxy@.getThreadContextValue("My
_Inbound_Payload");\
#consumer_id = (#soap_payload == null ? null :
@com.mercury.opal.capture.proxy.ProbeCodeSnippetHelper@.getConsumerIdFromDo
cument(##WS_inbound_service_name<java.lang.String>,#soap_payload<org.w3c.do
```

```
m.Document>));\
##WS_consumer_id = (#consumer_id == null ?
@ProbeCodeSnippetHelper@DO_NOT_STORE : #consumer_id);"";
```

First line: `f334f0b4 = #wsContext=#arg1;\`

As mentioned previously, the number `f334f0b4` was generated by going to http://<probe-host>:<probe-port>/inst/code-key and pasting the code snippet in the text box. The actual code snippet starts after `f334f0b4 =`. The first expression is `#wsContext=#arg1. It` simply assigns the first argument of the method—in this case, a MyWSContext object—to a local variable (wsContext).

Second line: `##WS_inbound_service_name=#wsContext.getServiceName();\`

This expression uses fragment local storage to store the service name. It is important to use the exact variable name (`WS_inbound_service_name`). These variable names are documented in the “Special Fields” section of "Code Snippet Grammar" on page 104.

Third line: `##WS_inbound_operation_name=#wsContext.getOperationName();/`

This expression uses fragment local storage to store the ws operation. It is important to use the exact variable name (`WS_inbound_operation_name`). These variable names are documented in the “Special Fields” section of "Code Snippet Grammar" on page 104.

Fourth line: `##WS_inbound_target_namespace=#wsContext.getNamespaceURI();\`

This expression uses fragment local storage to store the ws namespace. It is important to use the exact variable name (`WS_inbound_target_namespace`). These variable names are documented in the “Special Fields” section of "Code Snippet Grammar" on page 104.

Fifth line: `##WS_inbound_port_name=#wsContext.getEndpoint();\`

This expression uses fragment local storage to store the ws port name. It is important to use the exact variable name (`WS_inbound_port_name`). These variable names are documented in the “Special Fields” section of "Code Snippet Grammar" on page 104.

The above first five lines are sufficient to successfully capture the inbound Web Service. The remaining of the code snippet deals with capturing the consumer ID out of the SOAP payload. This is optional and only if the instrumented application server is not one of the application servers supported for capturing SOAP payload out of the box. See the previous example for details. In the followings example, we refer to the SOAP payload that was captured in the previous example.

Sixth line: `#soap_payload =
@com.mercury.opal.capture.proxy.ThreadContextProxy@.getAndRemoveThreadContextValue("My_
Inbound_Payload");\`

This expression retrieves and removes the stored payload from thread local storage (see the previous example on how this was stored) and stores it on a local variable (soap_payload).

Seventh line: `#consumer_id = (#soap_payload == null ? null :
@com.mercury.opal.capture.proxy.ProbeCodeSnippetHelper@.getConsumerIdFromDocument(##WS_
inbound_service_name<java.lang.String>,#soap_payload<org.w3c.dom.Document>));\`

This expression sets a consumer_id local variable. If the payload is null, the consumer_id is set to null. Otherwise, we use the service name to perform consumer ID matching based on the consumer.properties entries. For more information on consumer ID matching, see "Configuring Consumer IDs" on page 175.

Eighth line: `##WS_consumer_id = (#consumer_id == null ? @ProbeCodeSnippetHelper@DO_NOT_STORE : #consumer_id);"";`

In this final line, this consumer ID local variable becomes the consumer id for this server request. It is important to use the exact variable name (WS_consumer_id). These variable names are documented in the “Special Fields” section of "Code Snippet Grammar" on page 104.

Using Annotations for Custom Instrumentation

Applications can “force” the instrumentation of methods by simply using a custom annotation (InstrumentationPoint) that is contained in the **annotation.jar** file in the Diagnostics Java Agent lib directory. Put a copy of this file in your classpath when compiling your classes using the InstrumentationPoint annotation. The annotation is defined as follows (InstrumentationPoint.java):

```
/*
* (c) Copyright 2008 Hewlett-Packard Development Company, L.P.
* --------------------------------------------------------------------------
*/
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = ElementType.METHOD)
public @interface InstrumentationPoint {
String layer();
String keyword() default “”;
String layerType() default “method”;
String detail() default “”;
String code() default “”;
Boolean active() default true;
}
```

This feature requires that the **look.for.annotations** property in **inst.properties** is set to true (default).

Development

1. Add the path to the **annotation.jar** (or copy the jar into your application) file found in the Diagnostics Java Agent lib directory to your application build classpath.
2. Import the classes for any methods that need to be monitored:

   ```
   import com.mercury.diagnostics.common.api.InstrumentationPoint;
   ```

3. Identify methods to be monitored and add the annotation:

   ```
   @InstrumentationPoint(ARGS)

   public void launchTest4()
   ```

In this instance, ARGS includes the following (refer to points file documentation for more information about what these arguments mean):

- layer="layer name"
- keyword= "keyword"
- layerType="type"

- detail="details"
- active="true/false"

Example

The following example shows code that uses the InstrumentationPoint annotation.

```
/*
 * (c) Copyright 2008 Hewlett-Packard Development Company, L.P.
 ----------------------------------------------------------------------
 */
 import com.mercury.diagnostics.common.api.InstrumentationPoint;
…
 @InstrumentationPoint(layer=”my_app”,detail=”diag,method-no-trim,method-cpu-tim
e”)
 public void myMethod1(Object x, String y) {
…
}
```

In the example, myMethod1 will get instrumented and be visible as a node in all instance trees. It will not get trimmed, even if its latency goes below the minimum method latency threshold (51 ms by default). The inclusive (including children) CPU consumption by this method will be measured and reported.

Configuring Cross VM Correlations for New or Custom Technologies

Diagnostics can show call profiles that span multiple Java virtual machines (JVM). These Cross VM call profiles and topologies are very useful when a performance issue involves a client and a server. You want to know which application is the source of the problem but looking at the call profile for the client or server individually may not help with intermittent issues since they would not be correlated. The Cross VM call profile will show the client and the server correlated together in a single call tree.

Out-of-the-box the Java Agent provides support for this feature for many different technologies: for example, JMS, HTTP/S (Web Services only), RMI, SAP, TIBCO and Corba. With the latest version of Diagnostics, additional support was added to help you configure cross VM correlation for new or custom technologies.

The Cross VM correlation technique is exposed in code snippets, allowing you to prepare instrumentation points and code snippets to correlate almost any inter-process communication, including home-grown and legacy communication techniques. The only requirement for the communication technique is that its messages be able to carry an additional string, which is referred to as coloring.

The coloring string is created on the client side by the Java Agent, and attached to the outgoing message by a user-written code snippet. After the message is received, a user-written code snippet on the server side extracts the coloring from the message and passes it to the server side agent for parsing and processing.

Thus, your responsibility related to the cross-vm communication technique is primarily limited to embedding the coloring into the outgoing messages, and extracting the coloring from the received messages. This, of course, includes identifying the code locations (instrumentation points) for the client side (the outbound point), and for the server side (the inbound point). Refer to "Tutorial for Configuring Cross VM Correlation for Custom Technologies" on page 140 for a detailed example. And refer to "APIs Used to Facilitate Custom Transport

Cross-VM Correlations" on page 139 for information on the three APIs provided to help you configure custom cross-vm correlation.

Client Side

For the outbound calls (the client side), use the new **outbound:<coloring-type>** detail.

The available coloring types are:

- default
- sap
- none
- snippet

For all coloring types except **none**, there should be an associated code snippet, which will provide a special argument containing the technology type, the call target name and identification.

The argument has the following form:

```
DIAG_ARG:type=<type>&name=<name>&target=<target>
```

where <type> is the technology type used for the remote call, and <name> and <target> are technology dependent values. The technology type should be the same as the one used for the inbound instrumentation point (see "Server Side" on the next page).

For all coloring types except **snippet**, the probe will generate the appropriate coloring and it will report the coloring to the Diagnostics Server for future correlation. However, the outgoing message remains unmarked at this time.

For all coloring types except **none**, a code snippet for another instrumentation point (which is hit after the outbound point, preferably during the outbound method execution) must attach the generated coloring to the outgoing message.

The most recently generated coloring can be obtained by calling **ICorrelationColor RemoteCaptureProxy.getCurrentColor(#location)**.

In developing support for your own cross-vm communication, you may use **snippet**, which means that the coloring will be explicitly created by a direct call from a code snippet. For the snippet coloring the above order is reversed, which means the coloring is generated (and, most often, immediately attached to the message) before the outbound point is hit. Please note that this includes a case where the **before** code snippet for the outbound point creates the coloring, because the code snippet will be executed before the agent is called.

To create the coloring from code snippets:

1. Make a call to
   ICorrelationColor RemoteCaptureProxy.createColoring(#location, <type>, <diag-arg>)

   For type, use

   - RemoteCaptureProxy.ENCODED_COLORING for **default**
   - RemoteCaptureProxy.SAP_R3_COLORING for **sap**

   If in doubt which type to use, use the default.
2. Make a call to **grabCorrelationString()** on the object returned in step 1, and insert the returned string into the outgoing message (using a technology-dependent technique). This is where you can use your creativity.

Tip: If using String messages, use the following helper API to accomplish this step:

```
ProbeCodeSnippetHelper.createDiagEnvelope(String coloring, String
originalMessage)
```

3. Hit an instrumented point with the **outbound:snippet** detail. This will automatically use the most recently created coloring instead of creating a new one. Executing the outbound point informs the probe that the coloring was actually used, and identifies the method which will be considered the connection point for cross-vm call profiles. For synchronous cross-vm communication it is recommended to use **outbound** detail for a method that is used to both send the message and receive an acknowledgment, so the latency of the outbound call can be properly captured.

Server Side

For the inbound calls (the server side), use the **inbound:<technology-type>** detail. Use your own technology type names when supporting new cross-vm technologies. Check to avoid conflicts with existing technology names (server request types). Examples of server request types include: ADO, CICS, Corba, HTTP, JDBC, JMS, MSMQ, RMI, Remoting (.NET), SAP ABAP types, Web Services. In addition, you may see server request types named Pseudo and RootRename.

The **before** code snippet has to perform the following steps:

1. Extract the correlation string from the incoming message, using the technology-dependent technique, corresponding to the one used for the outbound calls.

   Tip: If the ProbeCodeSnippetHelper.createDiagEnvelope() was used previously, use ProbeCodeSnippetHelper.extractColoringFromDiagEnvelope(String envelope) to get the correlation string.

   And use ProbeCodeSnippetHelper.extractOriginalMessageFromDiagEnvelope(String envelope) to get the original message.

2. Leave TWO Strings on the stack: the capture argument (as any **before** code snippet should), and the extracted correlation string.

APIs Used to Facilitate Custom Transport Cross-VM Correlations

Three helper APIs were added to facilitate custom transport cross-VM correlations (see the tips in the sections above and see "Code Snippet Helper" on page 106 for information on their use. There is also a "Tutorial for Configuring Cross VM Correlation for Custom Technologies" on the next page to walk you through an example.

- `ProbeCodeSnippetHelper.createDiagEnvelope(String coloring, String originalMessage)`
- `ProbeCodeSnippetHelper.extractColoringFromDiagEnvelope(String envelope)`
- `ProbeCodeSnippetHelper.extractOriginalMessageFromDiagEnvelope(String envelope)`

HTTP/S Support

The support for the server side HTTP/S is built in and is enabled by default. The Java Agent automatically recognizes standard J2EE implementation of HttpServlet, as well as Jetty and Apache Catalina implementations. No user action is required on the server side, if one of these technologies is used.

For the client side, the Java Agent automatically instruments the openConnection method from the java.net.URL class, to embed the most recently generated coloring (if it exists) into the outgoing HTTP request. One of the HTTP request headers is used to carry the coloring. The header will be recognized by the server side agent.

Therefore, HTTP support on the client side is an exception to the above rules. You still have to provide the outbound point and the corresponding DIAG_ARG, but you do not have to worry about embedding the coloring into the outgoing messages.

For example, Diagnostics mediators use the following point:

```
[RemoteHttpComponent-Outbound-1]
class = com.mercury.diagnostics.common.net.registrar.RemoteHttpComponent
method = getConnection
signature = (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;)Ljava/net/HttpURLConnection;
priority = 1
detail = method-no-trim,outbound:default,before:code:7b1125e2
layer = Network.RemoteHttpComponent
```

The first argument for the getConnection method is a String representing the connection URL. The referred code snippet extracts from it the hostname and port and uses them for the target identification. A special utility method is provided by RemoteCaptureProxy to facilitate this conversion in a way consistent with the built-in part of the HTTP/S support.

```
7b1125e2 = #target=@RemoteCaptureProxy@.getTargetFromUri(#arg1); \
"DIAG_ARG:type=http&name="+#target+"&target="+#target;
```

Tutorial for Configuring Cross VM Correlation for Custom Technologies

This tutorial takes a simplified client-server application that uses a shared blocking queue as its custom transport solution. The client sends a "String" message by adding it to the queue. The server receives a "String" message by removing it from the queue.

Although this example runs in a single JVM (to keep it simple), it uses two threads to simulate an application running in two JVMs. (If your intention is to correlate threads running in a single JVM, there is a simpler solution that will help you do this. See"Performing Correlation Across Multiple Threads" on page 132 for more details).

The sample code is shown below:

```
public class SimulatedCrossVM {

 private static int INTERVAL = 5 * 1000; // 5 seconds
 private static BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
 private static class ReceiverSide extends Thread {

 public ReceiverSide() {
```

```
    super("Receiver");
  }
  public void execute(String receivedString) throws InterruptedException {
   System.out.println("Executing message: " + receivedString);
   sleep(2 * INTERVAL);
  }
  private void receiveAndHandleMessage() throws InterruptedException {
   String message = null;
   message = queue.take();
   // Handle it
   execute(message);
  }

  public void run() {
   try {
   while (true) {
    receiveAndHandleMessage();
    }
   }
   catch (Throwable t) {
    // oops
    t.printStackTrace();
   }
  }
 }
  private static class SenderSide extends Thread {

  // For simulated TCP connection
  private String destHost;
  private int destPort;
  public SenderSide(String host, int port) {
   super(host + ":" + port);
   destHost = host;
   destPort = port;
  }
  public void sendMessage(String origMessage) throws InterruptedException {
   queue.put(origMessage);
  }
  private String generateMessage() {
   String message = "T" + System.currentTimeMillis();
   return message;
  }
  private void generateAndSendMessage() throws InterruptedException {
   sleep(2 * INTERVAL);
   // generate message
   String message = generateMessage();System.out.println("Sender's original message: "
 + message);
   // And send it (outbound call)
   sendMessage(message);
```

```
  sleep(INTERVAL);
 }

 public void run() {
 try {
  while (true) {
   generateAndSendMessage();
 }
  }
   catch (Throwable t) {
    // oops
    t.printStackTrace();
   }
  }
 }

 public static void main(String[] args) {
  SenderSide sender = new SenderSide("fake-host", 12345);
  ReceiverSide receiver = new ReceiverSide();

  sender.start();
  receiver.start();
 }
}
```

Executing this code will have the following output:

```
Sender's original message: T1313785958127
Executing message: T1313785958127
```

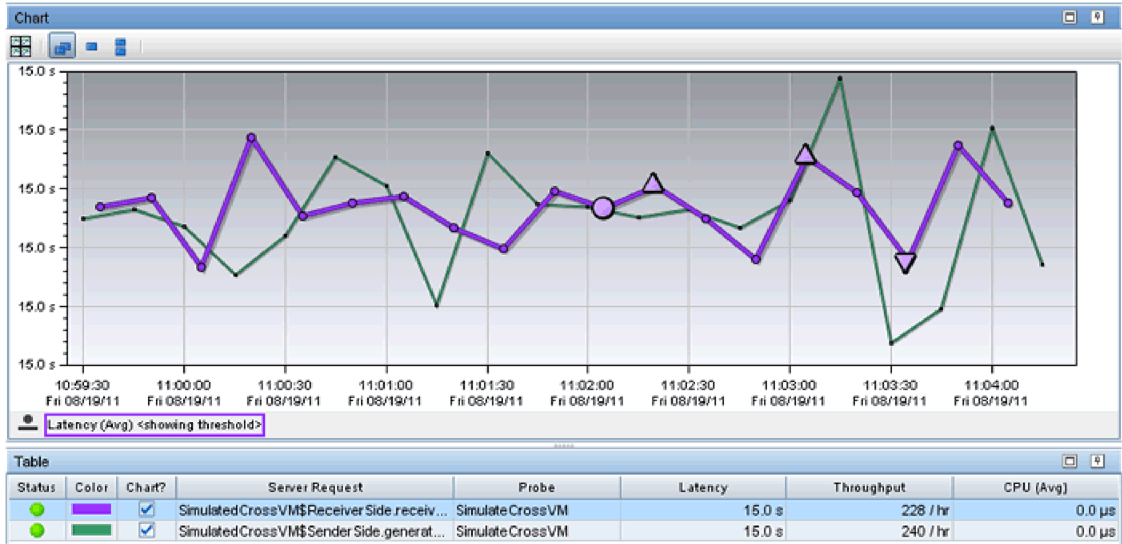
Step 1: Instrument Your Methods

By instrumenting your methods, you let Diagnostics know which methods are important. Since these methods are custom, the out-of-the-box instrumentation points won't do anything. Edit the **etc/autodetect.points** file by adding the following instrumentation points. See "Maintaining Instrumentation from the Java Profiler UI" on page 147 for guidance on defining instrumentation points.

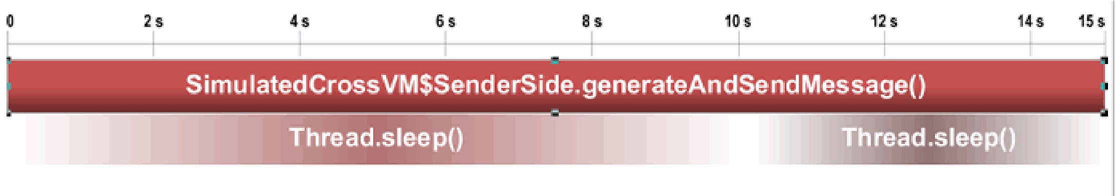
```
[SimCrossVM-Sender]
class = SimulatedCrossVM$SenderSide
method = generateAndSendMessage
signature = !.*
layer = Sending

[SimCrossVM-Outbound]
class = SimulatedCrossVM$SenderSide
method = sendMessage
```

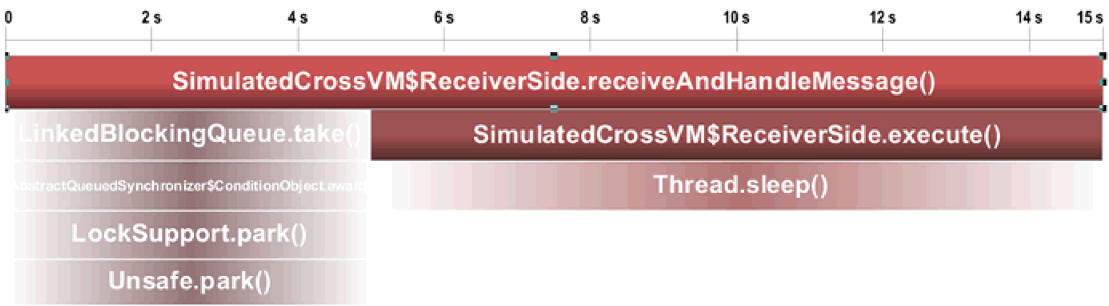
```
signature = !.*
layer = Sending

[SimCrossVM-Receiver]
class = SimulatedCrossVM$ReceiverSide
method = receiveAndHandleMessage
signature = !.*
layer = Receiving

[SimCrossVM-Inbound]
class = SimulatedCrossVM$ReceiverSide
method = execute
signature = !.*
layer = Receiving
```

Result: Running this instrumented test program, you see the following Server Requests:



Here are the call profiles shown for the sender and receiver.





Step 2: Add “Coloring” to the Sender Logic

In this step, we add "coloring" to the messages sent by the client. When the instrumented server receives this colored message, Diagnostics will correlate them. You add code snippets for a point in the **<agent_install_directory>/etc/code/custom_code.properties** file, This part is trickier, if you're not familiar with the code snippet syntax, it is described in "Defining Points With Code Snippets" on page 103.

First, we mark the method as an outbound point that uses a code snippet (outbound:snippet), and identify the code snippet to execute before invoking the method (before:code:5ea4753f). Since we're going to use the first argument, it's a good idea to provide a more specific signature (!\(Ljava/lang/String;.*).

```
[SimCrossVM-Outbound]
class = SimulatedCrossVM$SenderSide
method = sendMessage
signature = !\(Ljava/lang/String;.*
layer = Sending
detail = outbound:snippet,before:code:eb2d751f
```

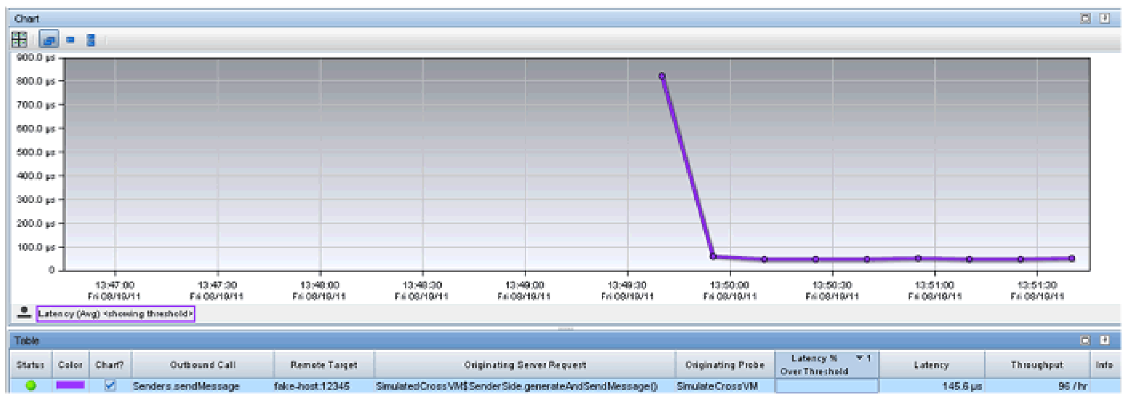
The corresponding code snippet is shown below. Line 1 creates a string (#target) that includes the hostname and destination port of the server. Line 2 defines a new string (#diagArg) that follows a special syntax (DIAG_ARG:type=<type>&name=<name>&target=<target>). The "type" is the technology type and can be any name you choose; it will be used in the next step. The "name" and "target" are technology dependent values that will be shown in the UI; they can also be anything you choose. Line 3 defines a third string (#color) which will be used to identify this specific invocation of the method call from any other. Line 4 updates the method's 1st argument with the colored String, which will cause sendMessage to send a modified String. Finally, line 5, places the coloring on the stack for usage by Diagnostics.

```
1. eb2d751f = #target=#callee.destHost+":"+#callee.destPort; \
2. #diagArg = "DIAG_ARG:type=CB-TCP&name=Senders.sendMessage&target="+#target; \
3. #color = (null == #arg1 ? null : @RemoteCaptureProxy@.createAndGrabColor(#location,
   @RemoteCaptureProxy@ENCODED_COLORING, #diagArg.toString())); \
4. #arg1 = @ProbeCodeSnippetHelper@.createDiagEnvelope(#color, #arg1);\
5. #diagArg;
```

Running the example updates the output as follows. Notice the receiving side did not get the same string message that was sent. This is a result of the code snippet's Line 4. In many cases, the receiving side may not handle this well. It's a good idea to note the receiver's behavior as this can happen "accidentally" if the client and server are not both using the same instrumentation, and in particular, not both instrumented.

```
Sender's original message: T1313786970403
Executing message: MF_DIAG1_!Dhf/
ABAABKrh3Qf0cy7yaLsAAAAAAA9mYWtlLWhvc3Q6MTIzNDUAYTEzMTM3ODY5N
jAzODgmU2ltdWxhdGVDcm9zc1ZNJlNpbXVsYXRlZENyb3NzVk0kU2VuZGVyU2lk
ZS52b2lkIGdlbmVyYXRlQW5kU2VuZE1lc3NhZ2UoKSZcMCZcMCZcMCY=:T131378
6970403
```

At this point, the only change you'll see in the UI is some "Outbound Calls". Notice the values in the columns "Outbound Call" and "Remote Target", these are the values you provided in the code snippet "name" and "target".



Step 3: Remove Coloring from the Receiver Side

The last step is to remove the coloring on the receive side so that the receiver can get the original "uncolored" message from the sender. First we mark the point as an inbound point with the technology type used in the code snippet defined in step 2, and assign a code snippet to run before this method is called. Again, we also specify a more specify signature since that argument will be used in the code snippet.

```
[SimCrossVM-Inbound]
class = SimulatedCrossVM$ReceiverSide
method = execute
signature = !\(Ljava/lang/String;.*
detail = before:code:d2c83d3c,inbound:CB-TCP
layer = Receiving
```

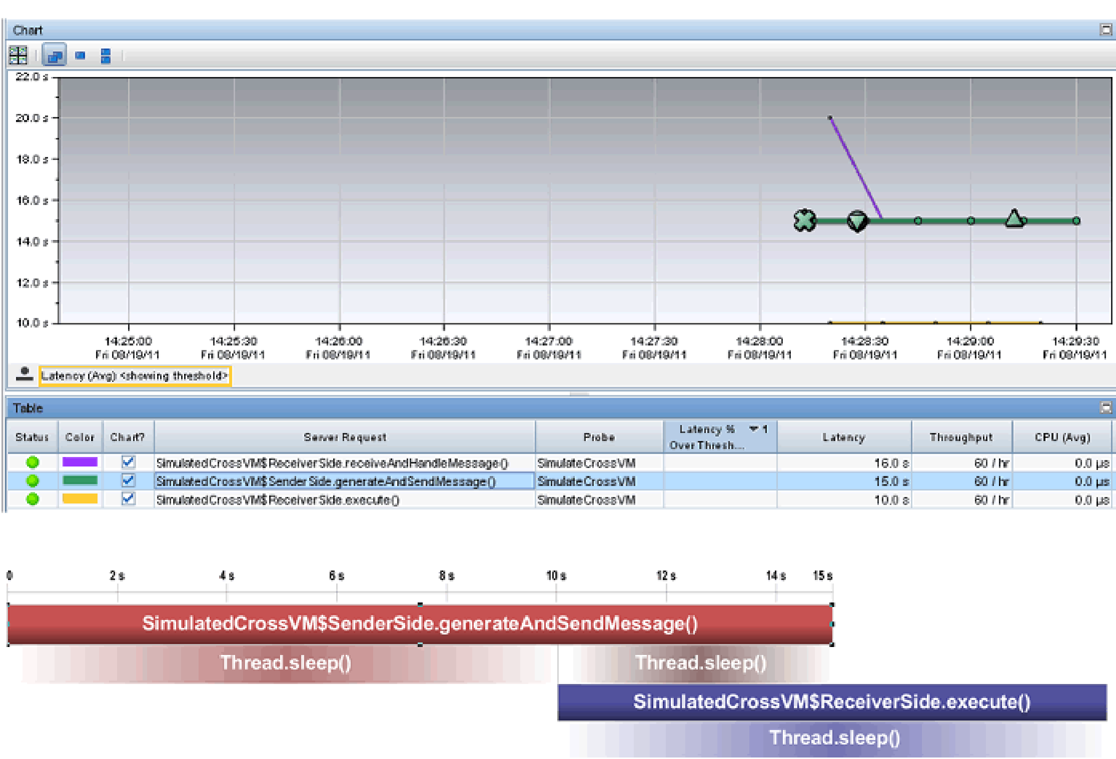
The corresponding code snippet is shown below. Line 1 extracts the coloring from the incoming message. Line 2 updates the method's 1st argument, restoring it to the original message sent by the client. Line 3 puts the coloring on the stack (and an empty String) for use by Diagnostics.

1. d2c83d3c = #coloring=@ProbeCodeSnippetHelper@.extractColoringFromDiagEnvelope(#arg1); \
2. #arg1=@ProbeCodeSnippetHelper@.extractOriginalMessageFromDiagEnvelope(#arg1); \
3. "";#coloring;

The program's output is now restored to the original:

```
Sender's original message: T1313789287234
Executing message: T1313789287234
```

The Server Request view now shows a Cross-VM call profile is available for the Sender's "generateAndSendMessage". Open this call profile and observe the client and server call profiles are now stitched together! They're not doing much in this sample application, but in a real application, you would be able to see if performance issues occur in the client, server, or both.

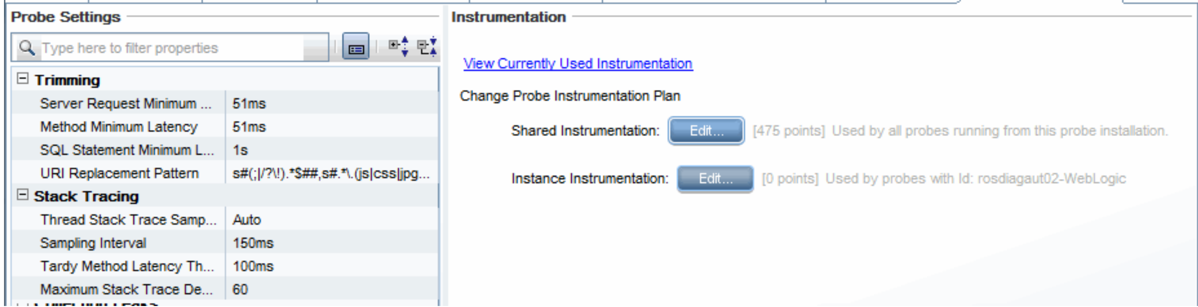


This call profile looks a bit strange but is typical for asynchronous applications. The client does not wait for a response, but does continue to do some processing (err sleeping for 5 seconds). During that time the server is processing the request and completes a few seconds afterwards. You will see the time durations for the methods in the tree as shown below. Notice also the diamonds with the number 2 inside, which represent the JVM depth. If your server made yet another outbound call, you could have 3 or more! In those cases, cross VM correlation because especially useful. Imagine trying to find the source of a performance issue across that many JVMs!

Maintaining Instrumentation from the Java Profiler UI

You can use the Configuration tab in the Java Diagnostics Profiler to maintain the instrumentation points and edit the probe configuration without having to manually edit the Java Agent capture points file or property files. You can access the Configuration tab from the Java Diagnostics Profiler whether profiling has been started or not.

The Instrumentation section of the Java Diagnostics Profiler gives you access to view and update the instrumentation for the application the probe is monitoring. The edit dialogs enable you to view and edit the instrumentation points as defined in the capture points file that Diagnostics uses to instrument your applications.



When you click **Edit...** for Shared Instrumentation, you are editing and changing the capture points files shared among all probes on the hosts. By default this is **<agent_install_directory>\etc\auto_detect.points**, however the probe may be using a custom capture points file. In that case you are editing the shared custom capture points file. For more information about custom capture points files, see "About Instrumentation and Capture Points Files" on page 96.

When you click **Edit...** for Instance Instrumentation, you are editing and changing the capture points file for this session of the profiler on this probe only.

Reviewing the Current Instrumentation

To review the layers, classes, and methods that were instrumented as a result of the points in the current capture points file, click **View Currently Used Instrumentation** in the Instrumentation section of the Configuration tab. The Profiler displays the Instrumented Layers page:

Diagnostics

Instrumented layers (no particular sorting)

Layer	Hits	Active Locations	Actions
(Other)	78	8 / 8	[Disable] [Clear # Hits]
(keyword) ejb30	0	0 / 328	[Enable] [Clear # Hits]
(keyword) http	61	13 / 13	[Disable] [Clear # Hits]
(keyword) remote-http	4	44 / 44	[Disable] [Clear # Hits]
Business Tier/EJB/Session Bean	0	0 / 11	[Enable] [Clear # Hits]
ClientSideMonitor/Instrumentation	0	0 / 3	[Enable] [Clear # Hits]
Database/JDBC	0	0 / 40	[Enable] [Clear # Hits]
Database/JDBC/Connection	31575	36 / 92	[Enable] [Disable] [Clear # Hits]
Database/JDBC/Execute	31579	47 / 47	[Disable] [Clear # Hits]
Directory Service/JNDI	0	0 / 5	[Enable] [Clear # Hits]
HttpStatus	50	16 / 16	[Disable] [Clear # Hits]
Java Server Faces/Lifecycle/Execute	0	0 / 2	[Enable] [Clear # Hits]
Java Server Faces/Lifecycle/Render	0	0 / 2	[Enable] [Clear # Hits]
Legacy/JCA/CCI	0	0 / 2	[Enable] [Clear # Hits]
Legacy/JCA/Connection	0	0 / 1	[Enable] [Clear # Hits]

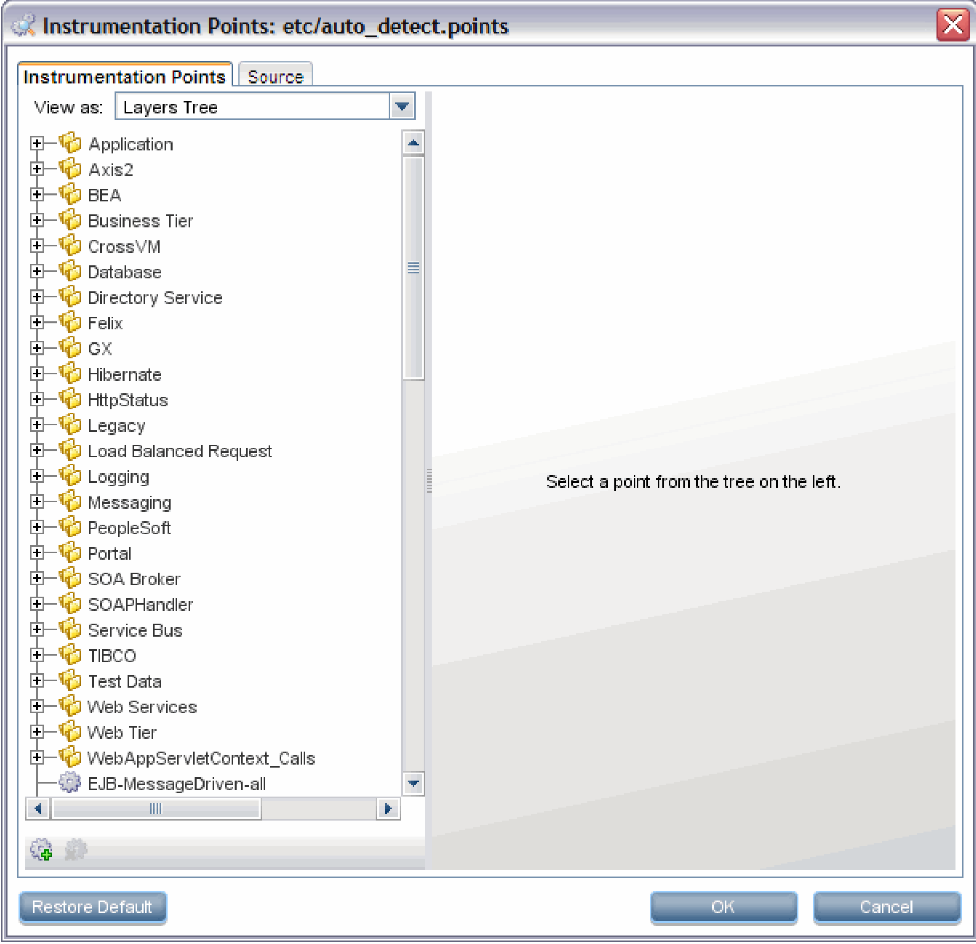
The Instrumented Layers page lists the layers that were instrumented, the number of times the instrumentation points in the layer were triggered, and the number of points currently active in the layer. The following columns are provided:

Column	Description
Layer	Lists the layers that were instrumented. The layer names in this column are links to a page that provides details about the processing in the layer that was monitored by the probe. **Note:** Only the layers defined in points that were actually instrumented are listed.
Hits	Contains a count of the number of times that the classes and methods that are monitored by the points in the listed layer were invoked. You can reset the count using the **Clear # of Hits** link in the Actions column.
Active Points	Contains the count of the number of points that are currently active as well as the total number of points that were defined for the particular layer.

Column	Description
Actions	Contains links that enable you to manipulate the information for the listed layers. The available action are: **Disable:** Disables all of the points in the selected layer so that they no longer capture data. The instrumentation stays in place and can be enabled again. Enabling or disabling points here is effective only until the next restart of your application. To change the default enabled state for a point, see "Coding Points in the Capture Points File" on page 98. **Clear # Hits**: Resets the hit count displayed in the # Hits column for the selected layer.

Maintaining the Instrumentation Points

To maintain the points that provide the instrumentation instructions that tell the probe what to monitor in your application, navigate to the Configuration tab in the Java Diagnostics Profiler and click **Edit...** for either the Shared Instrumentation or the Instance Instrumentation. The Instrumentation Points dialog opens:

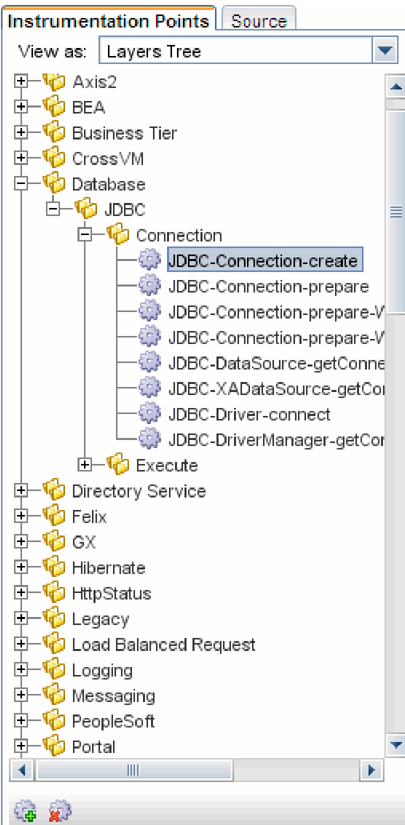


You can edit the instrumentation in two ways: visually, using a list or tree of points on the **Instrumentation Points** tab; or via the source of the capture points file on the **Source** tab.

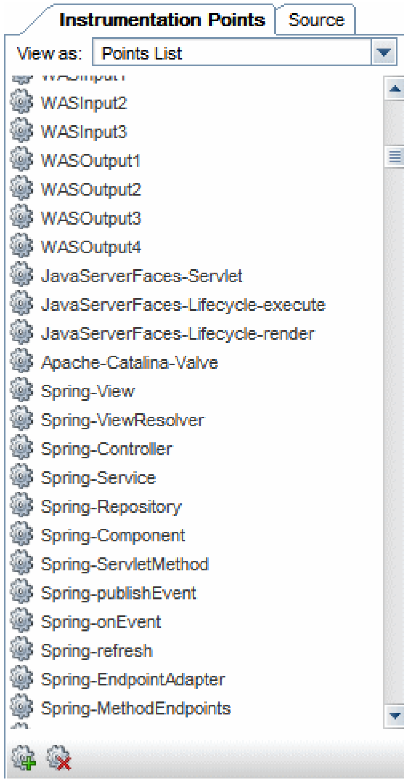
Selecting and Viewing an Existing Point

The navigation bar in the Instrumentation Points dialog helps locate the points in the capture points file that you would like to maintain. By making a selection from the **View as** dropdown, you can choose the format in which the points are listed.

When you select **Layers Tree** from the dropdown, you see a list of the points in the capture points file in a tree structure according to the layers and sublayers you assigned to the point:



When you select **Points List** from the dropdown, you see a list of the points in the capture points file in ascending alphabetical order:

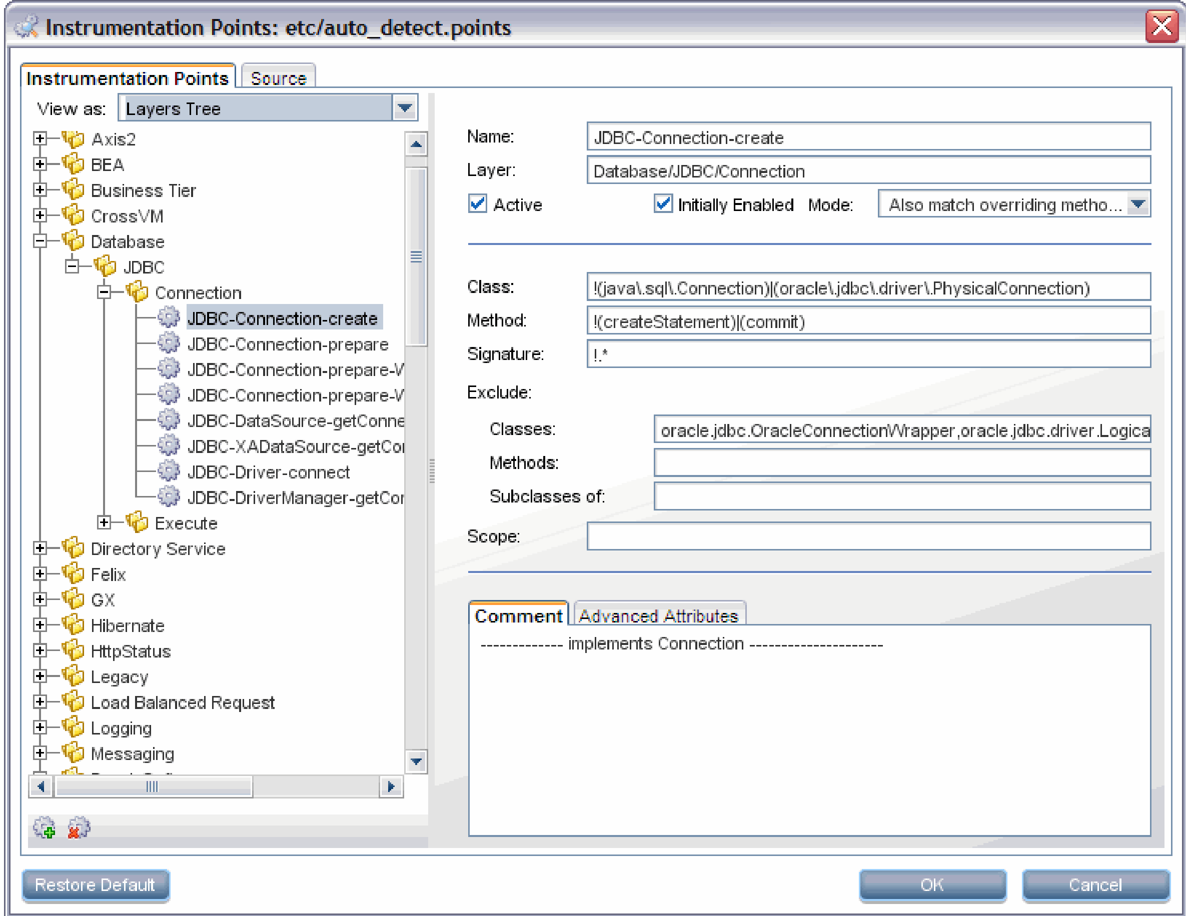


When you locate the point you want to view or maintain, select the point in the navigation bar. Then you see the details of the selected point in the view/edit panel where you can maintain the point.

Updating an Existing Point

When you select a layer or sublayer from the navigation bar, the view/edit panel contains only a prompt to remind you to select a point.

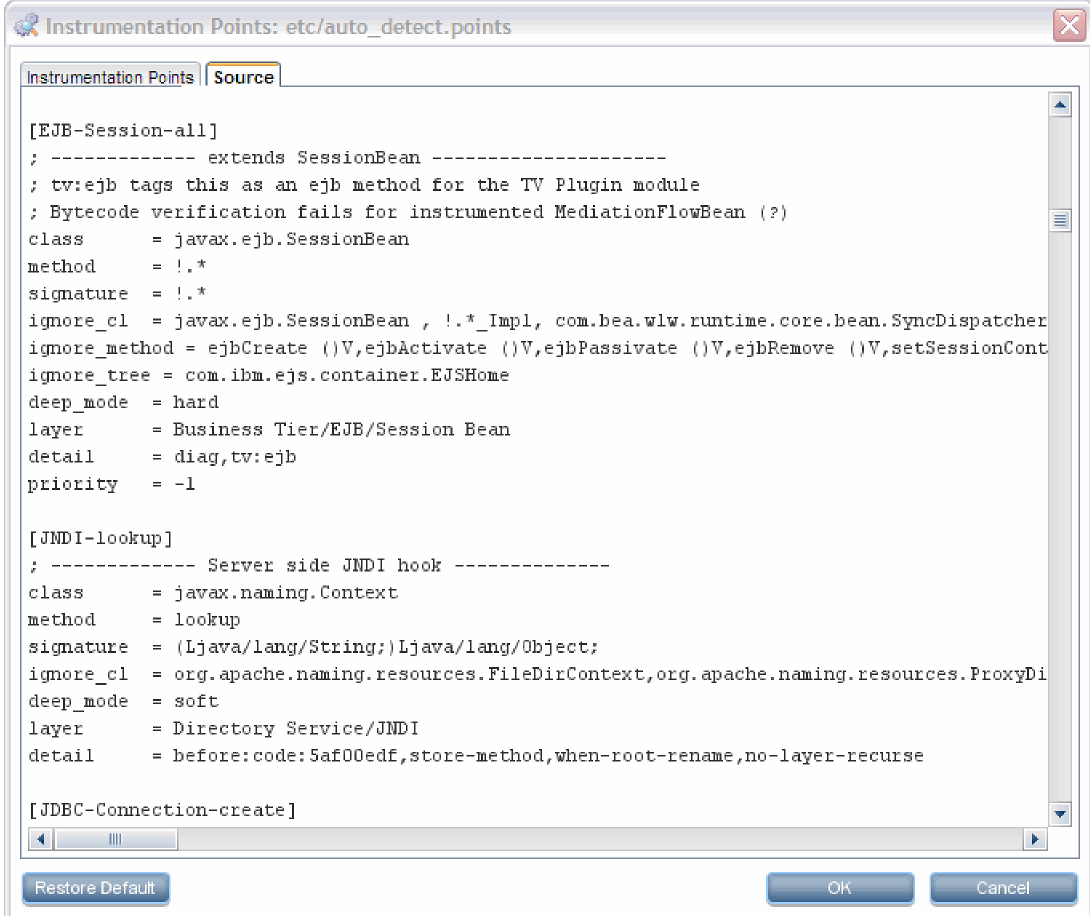
To update an existing point, select the point from the navigation bar so that the Profiler displays the details for the point in the Instrumentation Points tab of the view/edit panel:



The arguments that are commonly used when defining a point in the capture points file are displayed as separate data fields to make it easier for you to make any necessary updates. More advanced arguments are displayed in the **Advanced Attributes** tab at the bottom of the display. Comments for the point are displayed in the **Comment** tab. After making changes click **OK**. And remember to apply all of the changes made using the Configuration tab by clicking Apply Changes.

The arguments that can be used to define a point in the capture points file are documented in "Coding Points in the Capture Points File" on page 98.

The following is an example of the Source tab:



Deleting an Existing Point or Layer

You could delete a point or layer listed in the navigation bar.

To delete a point or layer:

1. Select the point or layer from the Navigation bar on the Instrumentation Points tab.
2. Click Delete Point ( ). The Profiler deletes the selected entity from the list in the navigation bar.

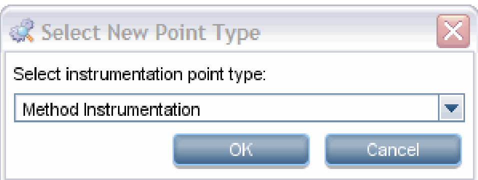
   The selected entity is not actually deleted from the capture points file until you apply all of your instrumentation points updates from the Configuration tab in the Profiler.
3. Close the Instrumentation Points dialog by clicking **OK**.
4. Apply all of the changes made using the Configuration tab by clicking **Apply Changes**.

Adding a New Point

You could add a point to the instrumentation.

To add a point:

1. Click New Point ( ). The Profiler displays the Select New Point Type dialog box:

2. Select the appropriate point type from the dropdown and click **OK.**

   The Profiler displays the Instrumentation Points tab with the view/edit section initialized for creating a new point for the selected point type.

3. Enter the arguments and comments for the new point into the appropriate locations on the tab.

   When you enter the Layer information, the entry for the new point in the navigation bar is updated to show the point in the correct existing layer or, if the layer that you specified does not already exist, with a brand new layer.

   The new point is not actually added to the capture points file until you apply all of your instrumentation points updates from the Configuration tab in the Profiler.

4. Close the Instrumentation Points dialog by clicking **OK**.
5. Apply all of the changes made using the Configuration tab by clicking **Apply Changes**.

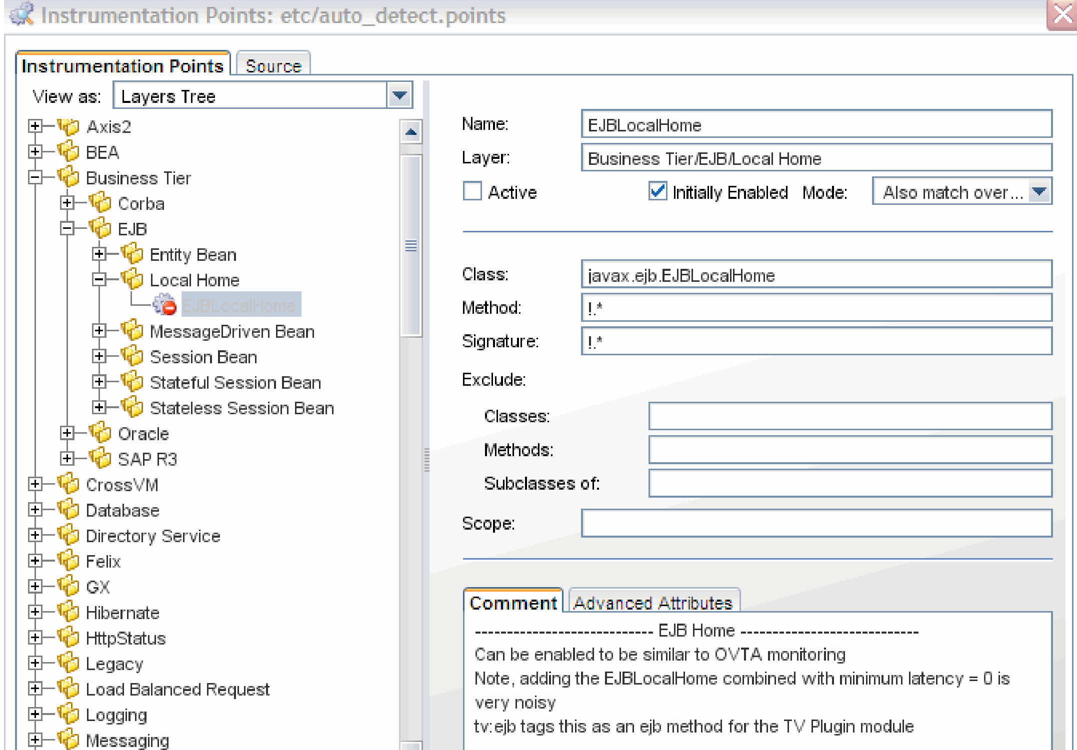
Activating OVTA-like Points

Points are included in the Java probe instrumentation for Servlet Filters and EJB local home methods. These instrumentation points provide additional functionality similar to the OVTA (OpenView Transaction Analyzer) Java Monitor.

The ServletFilter point requires that the HttpCorrelation2 point also be activated for server filters to be monitored correctly. This is because servlet filters sometimes are the first time Diagnostics sees an HTTP server request.

The EJBLocalHome, ServletFilter, and related HttpCorrelation2 instrumentation points are not active by default. Inactive points are indicated by a red symbol on the icon next to the instrumentation point, as shown below. To use these points, set active=true in the **auto_detect.points** file through the UI or by directly editing the file.

Locate these points in the Profiler UI as described in "Selecting and Viewing an Existing Point" on page 150 and navigate to the **Business Tier>EJB>LocalHome>EJBLocalHome** point or the **Web Tier>Servlet>ServletFilter** point and **HttpCorrelation2** point.



To set these points to active:

1. Select the point from the Instrumentation Points navigation bar so that the Profiler displays the details for the point. Check the **active** check box.
2. Close the Instrumentation Points dialog by clicking **OK**.
3. Apply all the changes made using the Configuration tab by clicking **Apply Changes**. Restart your application server (which restarts the probe) for the newly activated points to take effect.

Restoring Default Points

When you finish diagnosing a problem using the Profiler or Diagnostics Enterprise User Interface, you can restore the default instrumentation to avoid incurring the overhead from a more robust instrumentation.

To restore the default settings to the instrumentation:

1. Click **Restore Defaults.**

   The instrumentation points are not actually added to the capture points file until you apply all of your instrumentation points updates from the Configuration tab in the Profiler.
2. Close the Instrumentation Points dialog by clicking **OK**.
3. Apply all of the changes made using the Configuration tab by clicking **Apply Changes**.

Default Layers Defined for Typical Java Classes and Methods

Diagnostics Enterprise User Interface groups the performance metrics for classes and methods into *layers* and *sublayers* according to the instructions provided in the capture points file. The default layers were defined so that the performance metrics for processing in the application that used similar system resources could be reported together. The layers make it easier for you to isolate and identify the areas of the system that could be contributing to performance issues.

The following table lists the default layers and sublayers that are defined for typical Java classes and methods.

Platform-specific layers are also defined in the capture points file. These layers are, for the most part, sublayers of the top-level parent layers defined in the following tables. You can see performance data for layers in the Load View in the Diagnostics UI.

Java EE Layers

Layer	sublayers	Parent Layer
Web Tier	JSP Servlets Struts Session Spring Struts2	
Business Tier	EJB Corba	
Web Services		
EJB	Entity Bean Session Bean Local Home Stateless Session Bean Stateful Session Bean MessageDriven Bean	Business Tier
Directory Service	JNDI	
Database	JDBC	
JDBC	Execute Connection	Database

Layer	sublayers	Parent Layer
Messaging	JMS Spring	
JMS	Producer Listener Consumer	Messaging
Spring	Producer Consumer	Messaging
Hibernate		

Portal Layers

Diagnostics groups the performance metrics for the classes and method calls associated with processing for portals into Portal Component layers. Each Portal Component layer is broken down into layers for the portal lifecycle methods. For more information about portal layers, see the Diagnostics User Guide.

Chapter 11: Advanced Java Agent and Application Server Configuration

This chapter discusses advanced configuration of the Diagnostics Java Agent and the application server environment. Advanced configuration is for experienced users with in-depth knowledge of this product. Use caution when modifying any of the component properties.

This chapter includes:

- "Advanced Configuration Overview" below
- "About Dynamic Configuration" on the next page
- "Disabling the Java Diagnostics Profiler" on page 160
- "Controlling Probe Logging" on page 160
- "Setting the Probe's Host Machine Name" on page 161
- "Specifying a Different Probe IP Address" on page 161
- "Setting the Active Products Mode" on page 162
- "Controlling Automatic Method Trimming on the Agent" on page 163
- "Configuring URI and Parameter Capture" on page 164
- "Capturing Non-Sequential Server Requests" on page 167
- "Configuring an Agent for a Proxy Server" on page 167
- "Time Synchronization for Probes Running on VMware" on page 168
- "Limiting Exception Tree Data" on page 168
- "Diagnostics Probe Administration Page" on page 170
- "Authentication and Authorization for Diagnostics Java Profilers" on page 172
- "Configuring Collection of CPU Time Metrics" on page 174
- "Configuring Consumer IDs" on page 175
- "Configuring SOAP Fault Payload Data" on page 182
- "Configuring REST Services" on page 183
- "Customizing Grouping JMS Temporary Queue/Topics" on page 183
- "Configuring SQL Query Parsing" on page 183
- "Configuring Display of Application Name for Server Requests" on page 185
- "Maintaining Probe Settings from the Java Profiler UI" on page 186
- "Generating Performance Reports for JUnit Tests" on page 189

Advanced Configuration Overview

The following bullet points list the probe configuration sources of information to consult to configure your environment.

- If you have a probe that you want to prevent others from using in Profiler mode, see "Disabling the Java Diagnostics Profiler" on page 160.

- To have log messages posted to the probe logs for lower level messages, adjust the log level as described in "Controlling Probe Logging" on the next page.
- If you have more than one agent installed on the same host, make sure the log messages for each agent are stored in a different file, as explained in "Changing the Log Directory for a Probe" on the next page.
- To examine the performance of processing that would normally be trimmed from the metrics reported in Diagnostics, you can reduce the level of trimming or turn off trimming completely as described in "Controlling Automatic Method Trimming on the Agent" on page 163.
- If there is a proxy between the agent and the Diagnostics Server Commander, you must set the correct property to tell the agent the URL of the Diagnostics Server Commander, see "Configuring an Agent for a Proxy Server" on page 167.
- If you installed a Java Agent in an Software as a Service (SaaS) environment, disable the reverse http (rhttp) communication between the agent and the Diagnostics Server Mediator, see "Time Synchronization for Probes Running on VMware" on page 168.
- If you are running in a virtual environment, see "Time Synchronization for Probes Running on VMware" on page 168.
- If you need to limit the amount of exception data, see "Limiting Exception Tree Data" on page 168.
- If you want to use some of the collection options that require property file changes, see the other topics in this section such as "Configuring Consumer IDs" on page 175.

About Dynamic Configuration

The advanced configuration of the Java Agent is managed by property settings in several property and configuration files. You can view and modify these files in **<agent_install_directory>/etc/**.

Some property settings are picked up dynamically–they take effect a few minutes after the changes are saved to the file. The dynamic properties are as follows:

- Any property in the **dynamic.properties** file.
- Any property (or metric definition) in the **metrics.config** file.
- Any property in another property or configuration file that has a comment indicating its changes are picked up dynamically. For example, in **<agent_install_directory>/etc/dispatcher.properties**:

```
20 # Server configuration
21
22 # The URL of the server
23 # Commander in single server environment or distributed server in multi-
24 # environment.  If it ends with "/commander/registrar/", the mediator wi
25 # registration and keepalive requests to the commander.  If it ends with
26 # "/registrar/", then the mediator will send keepalive events to the ser
27 # all probes at once at a well-defined interval.  If this property is en
28 # set to its default value, then the commander.registrar.url property wi
29 # used to auto assign a registrar.url to this probe.
30 # (This property can be dynamically changed)
31 registrar.url=http://rosdiagcmdr01.ovrtest.adapps.hp.com:2006/commander/
32
```

Any property that is not in the categories above is non-dynamic. Changes to non-dynamic properties require an application server restart for the new settings to take effect. For example, all of the settings in **<agent_install_directory>/etc/auto_detect.points** are non-dynamic.

Disabling the Java Diagnostics Profiler

You can disable the Diagnostics Profiler for Java on a Java Agent so that it cannot be accessed accidently. When the Java Diagnostics Profiler is disabled, the user interface cannot be accessed from the Java Diagnostics Profiler URL: `http://<probe_host>:<probeport>/profiler`.

To disable the Java Diagnostics Profiler, set the **disable.profiler** property in **<agent_install_directory>/etc/probe.properties** to `true`.

The default value for **disable.profiler** is **false**. To enable the Java Diagnostics Profiler once it is disabled, change the value of the **disable.profiler** property from **true** to **false**.

Controlling Probe Logging

You can control the level of the messages the probe logs and change the location where the log messages are posted using the probe properties.

Controlling the Log Message Level

The level of messages from the probe that are logged to the standard output is controlled by the **lowest_printing_level** property in the property file **<agent_install_directory>/etc/logging.properties**. The default setting for this property is **OFF**. This prevents almost all agent messages from being logged to the console.

You can adjust the logging level dynamically by changing the value assigned to the **lowest_printing_level** property. The level of messages logged changes as soon as you save the property file.

The valid values for the **lowest_printing_level** property are:

Property Value	Description
OFF	No messages are logged.
DEBUG	All messages are logged.
INFO	Info, Severe, and Warning messages are logged.
WARN	Warning and Severe messages are logged.
SEVERE	Severe messages are logged.

Changing the Log Directory for a Probe

The default location for the log directory for a probe is **<agent_install_directory>/log.** When you have more than one probe on the same host, you can change the location of the log directory for each probe using the **log.dir** property. This property can be set in two ways:

- The value of the **log.dir** property can be set in the property file **<agent_install_directory>/etc/probe.property.**
- The value of the **log.dir** property can be specified on the startup command line for the application server as a JAVA system property as in the following example:

```
-Dprobe.log.dir=/path/to/log
```

For more information on specifying the **log.dir** property on the startup command line, see "Configuring an Agent for a Proxy Server" on page 167.

Setting the Probe's Host Machine Name

The probe's host name registers the probe with the Diagnostics commander server. The Diagnostics commander server uses the probe's host name to communicate with the probe and displays it along with the system metrics for the server that the probe is monitoring in the Diagnostics views.

The probe normally can detect the host name of the machine that is its host. In some situations, the server configuration is faulty and the probe cannot detect the correct host name. In situations where a firewall or NAT is in place or where your agent host machine was configured as a multi-homed device, it might not be possible for the agent to properly detect its host.

If the probe cannot detect its host name, you can instruct the probe to get the host name via a reverse DNS lookup based on the socket connection, or you can specify the host name using a probe property.

Instructing the Probe to Use Reverse DNS Lookup

If the configuration of the probe's host prevents the probe from detecting the host name, you can instruct the probe to detect the host name using a reverse-DNS lookup by setting the **server.host.name.lookup** property. This property is located in the **<agent_install_directory>/etc/dispatcher.properties** file.

The default value for the **server.host.name.lookup** property is 'false'. This tells the probe to do the lookup without using reverse-DNS. Set this property to 'true' instruct the probe to use reverse-DNS lookup.

Manually Specifying the Probe Host Name

The **probe.host.name.override** property enables you to manually set a host machine name for the probe and stop the probe from doing the automatic lookup.

To set a default host machine name for a probe, set the **probe.host.name.override** property (located in the property file **<agent_install_directory>/etc/dispatcher.properties**) to a machine name or IP address.

When you set the **probe.host.name.override** property, automatic lookup of the host name is disabled.

Note: Setting the **probe.host.name.override** property because of a NAT or firewall is only an issue for a test environment where you are using LoadRunner, Performance Center, or Diagnostics Standalone.

When you set the **probe.host.name.override** in a production environment where you are using BSM/APM or Diagnostics Standalone, the name you specify is shown as the host name in System Health.

Specifying a Different Probe IP Address

The **probe.host.ip.address.override** property (located in the property file **<agent_install_directory>/etc/dispatcher.properties**) enables you to override the Probe's IP address. You can use this property when you want the probe to provide the server with a different IP address, for example, a Virtual IP that would allow the server to communicate to the probe through a tunnel.

Setting the Active Products Mode

The Java Agent mode is typically set for you based on the options you enter in the setup program. But you can set the mode manually as described in this section.

The Java Agent can be set in different modes to do the following:

- Monitor applications from development through pre-production testing and into production
- Work with other Software products
- Be used as a standalone Diagnostics Java Profiler not reporting to a server or to other Software products

The mode the Java Agent works in is determined by the modes value of the **active.products** property located in the property file **<agent_install_directory>/etc/probe.properties**.

The modes value in the **active.products** property is also used in determining usage against the license capacity (see the chapter on Licensing in the Diagnostics Server Installation and Administration Guide). For Diagnostics there are two types of LTUs (License to use):

- AM - When using of the product in an enterprise mode, typically in a production environment.
- AD - When using the product in a pre-production load testing environment with probes in LoadRunner or Performance Center runs.

The value of the **active.products** property is initially set at the time you install the Java Agent.

- If you select Diagnostics Profiler Mode the value of the **active.products** property in the **etc/probe.properties** file is set to **PRO** mode at the time you install the Java Agent.
- With the Application Management/Enterprise Mode (AM License) option, the value of the **active.products property** in the **etc/probe.properties** file is set to **Enterprise** mode if you select the Diagnostics Server.
- If you select this AD License option, the value of the **active.products property** in the **etc/probe.properties** file is set to **AD** mode at the time you install the Java Agent.

To change the value of the **active.products** property you can edit the property file and restart the application server. Or you can re-run the Java Agent Setup and use the Change option to set the mode to Diagnostics Profiler Mode (PRO), Application Management/Enterprise Mode for Diagnostics (Enterprise) or Diagnostics Mode for LoadRunner/Performance Center (AD).

Note: To use the standalone Diagnostics Profiler for Java trial copy in enterprise mode or integrated with other Software products, contact Software Customer Support to purchase Diagnostics.

To see Diagnostics data in the user interface of the interfacing Software products, you must perform additional configuration steps. See the APM-Diagnostics Integration Guide or LoadRunner/Peformance Center-Diagnostics Integration Guide for details.

The sections that follow provide instructions for configuring each product mode of the **active.products** property.

PRO Product Mode – Diagnostics Profiler for Java

When PRO mode is set, the agent gathers performance metrics and presents them in the standalone Diagnostics Java Profiler user interface which is made available through a URL on the agent host.

If you are running the Java Agent as part of the Java Diagnostics Profiler trial copy, restrictions are placed on the agent to limit the load it can handle.

If you are running the Java Agent as part of the full Diagnostics enterprise product, or along with another Software product, the Profiler is enabled without the load restrictions.

PRO mode is not used in determine usage against license capacity.

Enterprise Product Mode

When configured in Enterprise mode, the agent works with Software products such as BSM/APM, LoadRunner, Performance Center, and as the full Diagnostics enterprise product. Although you must also do additional configure to enable these integrations (see theAPM-Diagnostics Integration Guide or LoadRunner/Peformance Center-Diagnostics Integration Guide for details).

In Enterprise mode data will also be sent to the Diagnostics Java Profiler.

Enterprise mode is the default for Java Agents (if you don't specify AD or AM mode). In Enterprise mode the agents are counted against the AM license capacity.

AM Product Mode

In AM mode the Java agent will capture all instrumentation data. You can set AM mode to protect an agent in a production BSM/APM deployment from accidentally being included in a LoadRunner or Performance Center run. In AM mode, the agent is not listed as an available agent in LoadRunner or Performance Center.

Agents in AM mode will always be counted against the AM license capacity.

AM mode supersedes all other modes except for AD.

AD Product Mode

In AD mode the Java agent will only capture data during a LoadRunner or Performance Center run and the results will be stored in a specific Diagnostics database for that run, for example, Default Client:21.

When the agent is in AD mode it will not use resources or send any data to the server unless the probe is part of a LoadRunner/Performance Center run.

Use this mode to prevent an agent in a QA environment from using additional resources and continually report data to the Diagnostics server when a load test is not running.

Another advantage of running a probe in AD mode is that probes in AD mode are only counted against the AD license capacity when in a LoadRunner or Performance Center run. For example if you have 20 probes installed in LoadRunner/Performance Center AD mode but only 5 are in a run, then only 5 are counted against AD license capacity.

See the LoadRunner/Peformance Center-Diagnostics Integration Guide for more information.

Controlling Automatic Method Trimming on the Agent

Default configuration for the agent includes settings that control the trimming of methods. Trimming can be controlled according to how long the method takes to execute, which is known as *latency*, and by the *stack depth* of the method call. The default configuration instructs the probe to trim both by latency and depth.

You could reduce the level of trimming, or turn off trimming completely. You can control trimming using the **minimum.method.latency** and **maximum.stack.depth** properties in **<agent_install_directory>/etc/capture.properties**.

Controlling Latency Trimming

Methods that complete with latency greater than or equal to the value of the **minimum.method.latency** property are captured, and those that complete with latency less than this limit are trimmed to avoid incurring the overhead for less interesting methods.

Note: In the following situations, latency is not trimmed when its latency is less than the trimming property:

- Methods that are the root for a call tree.
- Methods that threw an exception.

If the information for all methods must be captured, lower the value of the **minimum.method.latency** property or set it to zero.

Consider the following when setting the **minimum.method.latency** property:

- The lower the value of the **minimum.method.latency** property, the greater the chance that the performance of your application will be adversely impacted.
- Depending on your platform, and whether native timestamps are being used (**use.native.timestamps** = false), it might not be useful to specify this value in increments of less than 10 ms.

Controlling Depth Trimming

Methods that are called at a stack depth less than or equal to the value of the **maximum.stack.depth** property are captured. Those called at a stack depth greater than this limit are trimmed to avoid incurring overhead for less interesting methods.

Here is an example:

If **maximum.stack.depth** is `3 and /login.do calls a() calls b() calls c() then o`nly `/login.do`, `a`, and `b` are captured.

Note that setting a low **maximum.stack.depth** can significantly reduce the overhead of capture.

Configuring URI and Parameter Capture

Any HTTP/S server request URI, or HTTP parameter, can be transformed before being reported by the probe. Some of the transformations are based on regular expression matching and replacement and are controlled by properties in the **<agent_install_directory>\etc\dynamic.properties** file. The values of the properties controlling such replacement must use the **s/pattern/replace/** syntax. To perform multiple operations, use a comma-separated list. The operations are performed in order.

The URI or HTTP parameter transformations can be used when you are seeing too many server requests and you want to replace many server request URIs with one simplified server request URI that aggregates them.

For example, the following URIs may be accepted by a particular banking application:

```
/banking/account/00283117/status

/banking/account/02089003/check_balance
```

```
/banking/account/00082453/transfer/amount/250000/to/account/02089003
...
```

If the server requests are identified by the URIs as shown above, the number of different server requests can be very large. This can create storage problems on the Diagnostics server, but more importantly, it can make reported data very poorly suited for performance analysis. The reported server requests must be mapped to a manageable set, using the options below.

URI Truncation and Mapping

The regular expression matching and replacement for any HTTP/S server request URI is controlled by the **uri.pattern.replace** property in the **dynamic.properties** file.

In the banking application example shown above, you may want to eliminate the numbers following the "account/" and "amount/" parts in the URI. To do this, you set the **uri.pattern.replace** value as follows:

```
uri.pattern.replace=s'account/\\d*'account/*',s'amount/\\d*'amount/\\$'
```

This results in the server requests being reported as follows:

```
/banking/account/*/status
/banking/account/*/check_balance
/banking/account/*/transfer/amount/$/to/account/*
...
```

Caution: Overuse of this feature can impact performance.

You can see details and more examples as comments in the **dynamic.properties** file under URI Truncation and Mapping.

Automatic Detection and Trimming of REST-ful Server Requests

By default, the probe attempts to automatically detect the URI path elements that demonstrate high variability, as shown in the banking application example above. This behavior is controlled by the **automatic.uri.collapsing** property in the **<agent_install_directory>/etc/capture.properties** file. The value of the property is an expression indicating the maximum number of path segments allowed for each segment position, provided all the preceding path segments are the same. Whenever the number of the different values for the path segment exceeds the configured threshold, this path segment (in the given context) is replaced by an asterisk (*****). For example, in the banking application example shown above, after seeing a sufficiently large number of different account numbers and transfer amounts (equal to or greater than the value configured in the **automatic.uri.collapsing** property), the probe reports the server requests as:

```
/banking/account/*/status
/banking/account/*/check_balance
```

```
/banking/account/*/transfer/amount/*/to/account/*
...
```

Automatic detection makes manual configuration as described in the previous section unnecessary, but it may require a relatively large set of different URIs to be seen by the probe before it is activated.

Tip: The probe stores its internal data related to this feature in the **log/<probe-id>/<probe-id>_sr.templates** text file. You can "train" the Diagnostics probe in a test environment to determine the correct set of server requests to report, and then copy this file to a production environment before starting the production probe.

URI Trimming

If you have too many server requests, you can also use the **maximum.uri.pathsegments** property in the **capture.properties** file to trim server requests to a configured number of path segments.

The default for this setting is **-1**, which disables the property. For probes reporting in a SaaS environment (SaaS selected in the Java Agent setup) **maximum.uri.pathsegments** is automatically set to 2 to ensure the volume of server request data sent to Micro Focushosted servers is not too large.

For example, a setting of 2 results in no more than two path segments, so the URI `/banking/account/00082453/transfer/amount/250000/to/account/02089003` is trimmed to `/banking/account`.

The probe applies the URI transformations in the following order:

1. URI mapping (configured by the **uri.pattern.replace** property).
2. Static content replacement (configured by the **uri.static_content.suffixes** property).
3. URI trimming (configured by the **maximum.uri.pathsegments** property).
4. Automatic URI transformation (configured by the **automatic.uri.collapsing** property).

Caution: While each of the above transformation steps can be disabled, we do not recommend disabling all of them.

HTTP Parameter Truncation and Mapping

You can transform any captured HTTP parameter. This can be useful when a parameter value is too complex to be used in server request classification without causing symbol table explosion.

The regular expression matching and replacement works in the same manner as for URI Truncation and Mapping explained above, and it is controlled by the **parameter.pattern.replace.<property-key>** property in the **dynamic.properties** file, where **<property-key>** is the HTTP parameter name (key).

You must enable HTTP parameter capture in the **[HttpCorrelation]** point in the **auto_detect.points** file using the **args_by_class** keyword. Also, if your HTTP requests use the POST method, you must specify **ignore.post.parameters=false** in the **inst.properties** file.

For example, if you want to capture the HTTP parameter **eventSource**, which takes values like

```
FNOLVehicleIncidentPopup:FNOLVehicleIncidentScreen:VIPS:VehicleDamageDescription
```

and you only want to keep the part up to the first colon (**:**), you can add the following line to the **[HttpCorrelation]** point definition in the **auto_detect.points** file:

```
args_by_class = !.*&eventSource
```

and add the following line to the **dynamic.properties** file:

```
parameter.pattern.replace.eventSource=s':.*''
```

Caution: Overuse of this feature can impact performance.

You can see details and more examples as comments in the **dynamic.properties** file under HTTP Parameter Truncation and Mapping.

Capturing Non-Sequential Server Requests

Some non-J2EE applications split the work to be performed by a single server request into components that are executed by multiple threads. Server requests such as these are termed as non-sequential because the components can be run concurrently. There are a number of frameworks that applications can use to organize concurrent execution of the code. Some of the frameworks are limited to concurrent I/O, while others, like the standard J2SE package java.util.concurrent, can be used for almost any purpose.

By default, support for non-sequential server requests is disabled. To enable non-sequential server requests, set the parameter `mercury.enable.non_sequential.fragments = true` for the property `details.conditional.properties` in `etc/inst.properties`. Support for non-sequential server requests causes slight increase in the probe overhead. Therefore, it is recommended to leave it disabled for applications that do not need it.

Similar to the traditional frameworks, Java Agent offers out-of-box support for a limited number of concurrent frameworks. Additional support can be added by enhancing the Java Agent configuration in `auto_detect.points`.

Configuring an Agent for a Proxy Server

Note: This section only applies if you are using the Java Agent with a Diagnostics Server.

Two properties are used to specify for the Java Agent, the URL of the Diagnostics Commander Server. The property you set depends on whether or not there is a proxy.

- **registrar.url** in **dispatcher.properties**

  The **registrar.url** property in **<agent_install_directory>\etc\dispatcher.properties** is set when you install the agent. When there is a direct connection between the agent and the URL of the Diagnostics Commander Server, the agent uses the value of this property.

- **registrar.url** in **webserver.properties**

  In the presence of a proxy, you must set the **registrar.url** property in the **<agent_install_directory>\etc\webserver.properties** file to indicate the URL of the Diagnostics Commander Server.

Time Synchronization for Probes Running on VMware

For probes running in a VMware guest, time must be synchronized between the VMware guest and the underlying VMware host. If time is not synchronized properly, the Diagnostics UI could display inaccurate metrics or no metrics at all from a probe running in a VMware guest.

Time should be synchronized according to the recommendations given in the VMware whitepaper on timekeeping (http://www.vmware.com/pdf/vmware_timekeeping.pdf) in a section on "Synchronizing Hosts and Virtual Machines with Real Time." VMware Tools must be installed in each VMware guest operating system that hosts a Diagnostics probe. The time synchronization option in VMWare Tools must be turned on.

This option in VMware Tools works only if the guest operating system time is initially set earlier than that of the VMware host. For instructions on how to install VMware Tools, see the "Basic System Administration" document for VMware ESX Server. If any non-VMware time synchronization software (such as Network Time Protocol) is used, it should be run in the VMware ESX server service console.

If you encounter negative latency issues when running the probe on VMware guest with the probe property attempt.vmware.timestamp.adjustments set to true, you should set the following property in the probe **etc/capture.properties** file:

```
use.vmware.timestamp.workaround=true
```

When use.vmware.timestamp.workaround is set to true, the probe will use the alternative call to get the VMware host timestamps, so as to workaround the negative latency issue.

Limiting Exception Tree Data

The agent collects exception information and uses it to build exception instance trees. Exception instance trees provide the data for the exception information that appears in the Diagnostics UI such as a stack trace.

By default, every exception that occurs in the monitored application is a candidate for the exception instance trees. Collecting all exception information is usually undesirable, however, because exceptions that are not of interest overload the display as well as the data collection and transfer operations. You can, therefore, limit the exception types for which data is collected. For example, filtering application server-based errors such as **javax.naming.AuthenticationException** allow the exception trees to contain more application-specific errors.

The exception tree data collected is controlled by limiting specific exception types or limiting the number of exception types.

Limit Specific Exception Types

You can control which specific exception types are excluded and included from collection by setting the **exception.types.to.exclude** and **exception.types.to.include** properties in the **<agent_install_ directory>\etc\dispatcher.properties** file as follows:

- **exception.types.to.exclude**

  Set this property to ignore exceptions of one or more specified types. All subtypes of each specified type are also ignored unless the subtype is specified by the **exception.types.to.include** property.
- **exception.types.to.include**

  Set this property to specify which, if any, of the specified excluded exceptions (or their subtypes) are to be included. Subtypes of any exception type specified to be included are also included.

Both properties take lists of fully-qualified exception type names, separated by commas. Changes to the **dispatcher.properties** file take effect immediately. It is not necessary to restart the application.

Limit the Number of Exception Types

You can limit the exception tree data collected by specifying the number of different types of exceptions by setting the **exception.instance.tree.count** property in **server.properties**. By default, this property is set to 4, which indicates that only the first four exceptions types encountered during the probe's data collection cycle are used in building the exception trees. You can raise or lower this setting.

Examples

The following example causes exceptions of type ClassNotFoundException and all its subtypes to be ignored.

```
...
exception.types.to.exclude=javax.naming.AuthenticationException
```

The following example causes some subtypes of the java.lang.IOException class to be excluded, as indicated by the diagram that follows:

```
...
exception.types.to.exclude=java.io.IOException,java.io.InvalidClassException
exception.types.to.include=java.io.ObjectStreamException
```

The following diagram shows the excluded and included exception types on the java.io class hierarchy:

- java.lang.Throwable } Included by default
  - java.lang.Error } Included by default
    - java.io.IOError } Included by default
  - java.lang.Exception } Included by default
    - java.io.IOException
      - java.io.CharConversionException } Excluded
      - java.io.EOFException } Excluded
      - java.io.FileNotFoundException } Excluded
      - java.io.InterruptedIOException } Excluded
      - java.io.ObjectStreamException — Included
        - java.io.InvalidClassException — Excluded
        - java.io.InvalidObjectException } Included
        - java.io.NotActiveException } Included
        - java.io.NotSerializableException } Included
        - java.io.OptionalDataException } Included
        - java.io.StreamCorruptedException } Included
        - java.io.WriteAbortedException } Included
      - java.io.SyncFailedException } Excluded
      - java.io.UnsupportedEncodingException } Excluded
      - java.io.UTFDataFormatException } Excluded

Diagnostics Probe Administration Page

You can use the Diagnostics Probe Administration page to configure Java Agent and Profiler settings. Access the Diagnostics Probe Administration page directly from your browser.

Accessing the Diagnostics Probe Administration Page

Open the Diagnostics Probe Administration page inside your browser.

To access the Diagnostics Probe Administration page:

1. In your browser, navigate to `http://<probe_host>:<probeport>`.

   A probe is assigned to the first available port, beginning at **35000**.

   The Administration page opens.

2. Select the menu option for the activity you want to perform.

   - **Open Diagnostics Profiler:** Opens the Java Diagnostics Profiler.

   - **Advanced Options:** Opens the Components pages. For more information, see "Diagnostics Probe Components Page" on the next page.

     - If your probe is configured to work with a Diagnostics Server, the probe (Profiler) authorization and authentication settings are managed from the Diagnostics Commander Server to which this probe is connected. When you click this option, you are redirected to that Diagnostics Commander Server. For more information, see “User Authentication and Authorization” in the *Diagnostics Server Installation and Administration Guide*
     - If your probe is configured to work as a Profiler only and is not connected to any Diagnostics Server, this option opens the User Administration page, where you can create, edit and delete users and change their privileges. For more information, see "Authentication and Authorization for Diagnostics Java Profilers" on page 172.

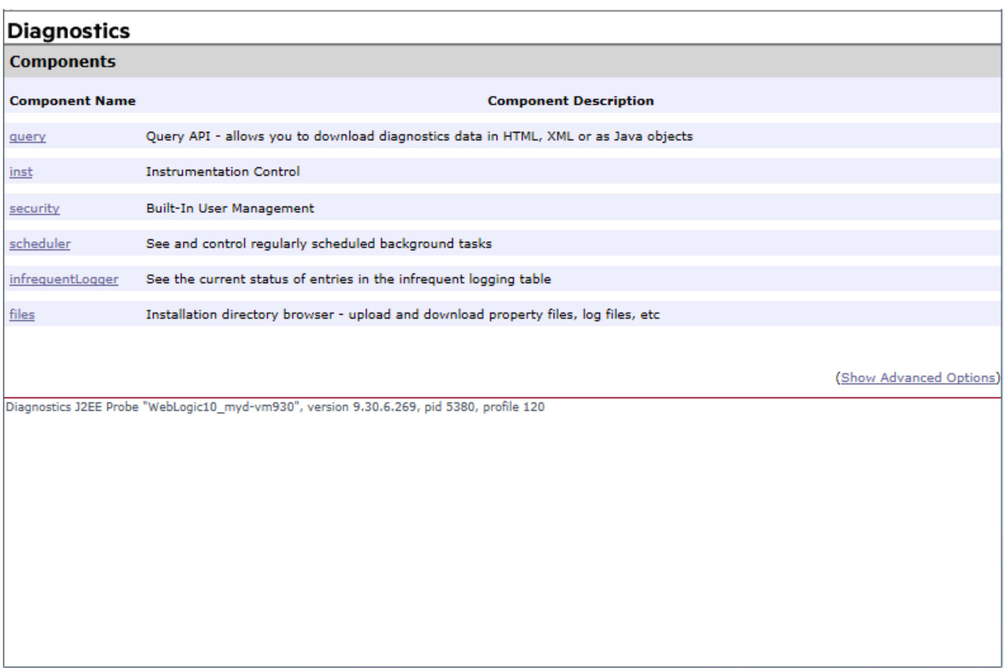
Diagnostics Probe Components Page

From the Components page you can open the Java Diagnostics Profiler, and access the User Administration page.

To access the Components page:

1. Open the Diagnostics Probe Administration page as described in "Accessing the Diagnostics Probe Administration Page" on the previous page.
2. Click **Advanced Options**.
3. If prompted, enter your user name and password.

   The Components page opens.

   Diagnostics

   Components

Component Name	Component Description
query	Query API - allows you to download diagnostics data in HTML, XML or as Java objects
inst	Instrumentation Control
security	Built-In User Management
scheduler	See and control regularly scheduled background tasks
infrequentLogger	See the current status of entries in the infrequent logging table
files	Installation directory browser - upload and download property files, log files, etc

   (Show Advanced Options)

   Diagnostics J2EE Probe "WebLogic10_myd-vm930", version 9.30.6.269, pid 5380, profile 120

4. Click one of the following options:
   - **query.** For internal use by developers.
   - **inst.** Includes various instrumentation options. For more information about probe instrumentation, see "Custom Instrumentation for Java Applications" on page 96.
   - **security.** Depending on how your probe is configured, you access a different page from this option.
     - If your probe is configured to work with a Diagnostics Server, the probe (Profiler) authorization and authentication settings are managed from the Diagnostics commander server to which this probe is connected. When you click this option, you are redirected to that Diagnostics commander server . For more information, see “User Authentication and Authorization” in the Diagnostics Server Installation and Administration Guide.
     - If your probe is configured to work as a Profiler only and is not connected to any Diagnostics Server, this option opens the User Administration page, where you can create, edit, and delete users and change their privileges. For more information, see "Authentication and Authorization for Diagnostics Java Profilers" on the next page.

- **scheduler.** Enables you to see and control regularly scheduled background tasks. For the ServerCommunication scheduler or the sharedInfrequentEventScheduler, you can see the state and the number of tasks inside each. For each task, you can select an action such as RUN NOW or DELETE.
- **infrequentLogger.** See the current status of entries in the infrequent logging table.
- **files.** Installation directory browser – upload and download property files, log files, etc.

  Note: By default, the upload button on this page is disabled. To enable it, in the **<agent_install_directory>/etc/common.properties** file, change the value of the **enable.file.uploadFromUI** parameter to **true** (enable.file.uploadFromUI=true).

  Caution: Enabling this feature may lead to security issues.

Authentication and Authorization for Diagnostics Java Profilers

When you install the Java Agent as a Profiler only (not connected to any Diagnostics Server), you can manage the authentication and authorization of users of the Profiler from the Diagnostics Probe User Administration page.

Note: If the Java Agent is configured to work with a Diagnostics Server, the probe (Profiler) authorization and authentication settings are managed from the Diagnostics Commander Server to which this probe is connected. For more information, see “User Authentication and Authorization” on page 787 in the Diagnostics Server Installation and Administration Guide.

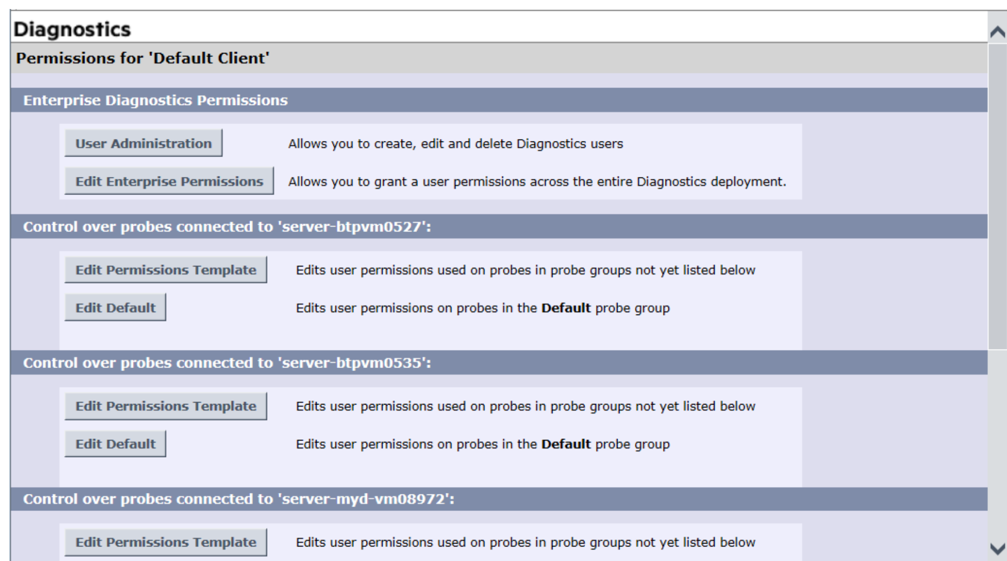
To manage authentication and authorization for users of the standalone Java Diagnostics Profiler:

1. Access the Diagnostics Probe Administration page

   In your browser, navigate to http://<probe_host>:<probeport>. A probe is assigned to the first available port, beginning at `35000`.

   The Diagnostics Probe administration page opens.

2. Select **Advanced Options > Security** to open the User Administration page.



On the User Administration page, you can create new users, assign privileges to users, change passwords of existing users, and delete users.

To create a new user:

1. Click **Create User**, enter a user name in the **New User Name** box, and click **OK.** The new user appears in the list of user names.
2. In the row representing the new user, type a password in the **Password** box and confirm it by retyping it in the **Confirm Password** box.
3. Type the password of the user currently logged on, in the **Password for <current user>** box and click **Save Changes**.

To assign privileges to a user:

1. Go to the row representing the relevant user and select the appropriate check boxes representing the different privileges.

   The following privilege levels can be assigned to Java Diagnostics Profiler users:

Privilege	Description
View	The user can view Profiler data from the UI.
Execute	The user can perform garbage collection and clear the performance data held by the Profiler.
Change	The user can run potentially risky operations, such as taking a heap-dump or changing instrumentation.

   The privilege levels, **rhttpout** and **system** are for internal purposes only.

   Each privilege level stands alone. There is no inheritance of privileges from one level to the next. You must grant a user all of the privilege levels that are necessary to perform the functions they need to perform.

   a. Type the password of the user currently logged on, in the **Password for <current user>** box and click **Save Changes**.

   > **Note:** If login fails, the user profiler is checked for authentication in the Command Server

> etc/.htaccess authentication
>
> If login is successful, the user profiler is checked for authentication in the Local etc/.htaccess authentication
>
> ```
> enable.local.htaccess.authentication=false
> ```
>
> Reference Path: <java_agent_home>/etc/probe.properties file

To change the password of an existing user:

1. Go to the row representing the relevant user, type a password in the **Password** box, and confirm it by retyping it in the **Confirm Password** box.
2. Type the password of the user currently logged on, in the **Password for <current user>** box and click **Save Changes**.

To delete a user:

1. Type the password of the user currently logged on, in the **Password for <current user>** box.
2. Click **Delete user** ( ) corresponding to the user you want to delete.

   A message box opens asking if you want to delete the selected user.
3. Click **OK** to delete the user.

Configuring Collection of CPU Time Metrics

The CPU Time metrics appear in the Details pane for the Transaction view, the Probes view, the Call Profile view, and the Portal Components view. You can enable, disable, and configure the collection of CPU time metrics. The CPU time metrics are **CPU (Avg)** and **CPU (Total)**. If collection of CPU time metrics is disabled or not configured for methods, you will see N/A for these metrics.

The CPU Time metrics rely on CPU timestamping which is generally supported on the following platforms: Windows, Solaris, AIX, and Linux kernels 2.6.10 or later (for example RedHat 5.x, SUSE 10.x).

> **Note:** Support for CPU timestamping can vary, however, not only by operating system, but also by platform architecture (for example SPARC versus x86).
> For the most recent information on support for CPU Time on specific platform versions and architecture, see the Diagnostics Support Matrix at Diagnostics_System_Requirements Guide.

> **Note:** In VMware, the CPU time metric is from the perspective of the guest operating system and is affected by the VMware virtual timer. See the VMware whitepaper on timekeeping at http://www.vmware.com/pdf/vmware_timekeeping.pdf and "Time Synchronization for Probes Running on VMware" on page 168.

By default, collection of CPU time metrics is enabled for server requests. You can disable CPU time metric collection and configure collection of CPU time metrics in property files or using the Java Diagnostics Profiler UI. You can configure collection of the following CPU Time metrics:

- Server Requests only
- Server Requests and Portlet Methods
- Server Requests and All Methods

For a Java Agent, the collection of CPU Time metrics is controlled by two properties:

- **use.cpu.timestamps** property in **<agent_install_directory>\etc\capture.properties**.

  This property is set to **true** by default, which enables collection of CPU time metrics. Collection of any CPU timestamps is controlled by a second property listed below. If you set the use.cpu.timestamps property to false, the CPU time metrics are not collected for any server request or method reported by the probe
- **cpu.timestamp.collection.method** property in **<agent_install_directory>\etc\dynamic.properties**.

Caution: Use caution when configuring the collection of CPU timestamps because of the increase in Diagnostics overhead. The increased overhead is caused by an additional call for each method that is needed to collect the timestamp.

Cpu.timestamp.collection.method can be set to one of the following:

- **0 –** No CPU timestamping.
- **1 –** CPU timestamps collected only for server requests.

  The default value is **1,** which means CPU times can be reported at the server request level but not the transaction level. However, if the setting is removed or commented out of the properties file, the default is 0.
- **2 –** CPU timestamps collected for All server requests and ALL methods.
- **3** – CPU timestamps collected for ALL server requests and the lifecycle methods instrumented for Portal Components.

Another way to set the **cpu.timestamp.collection.method** property is using the Configuration tab in the Java Diagnostics Profiler as follows:

1. In the **Profiler** UI, select the **Configuration** tab. The profiler does not need to be started to make this probe configuration change.
2. In the Configuration screen, select a **Collect CPU Timestamps** option from the dropdown list.

CPU Timestamp Collection Method	Description
None	No CPU Timestamps.
For Server Requests Only	CPU timestamps are only collected for server requests.
For Server Requests and Portlet Methods	CPU timestamps are collected for ALL server requests and the lifecycle methods instrumented for portal components.
For Server Requests and All Methods	CPU timestamps are collected for ALL server requests and ALL methods.

3. When you complete your changes, click **Apply Changes.**

Note: Your changes take effect immediately. You do not need to restart the application (or probe).

Configuring Consumer IDs

Web service metrics can be grouped by particular consumers of the Web service. The metrics are then aggregated for that consumer and displayed in SOA Services views such Services by Consumer ID or Operations by Consumer ID. There are several ways of defining the consumer ID:

- "A Value in a SOAP Header"
- "A Value in a SOAP Envelope"
- "A Value in the SOAP Body"
- "A Value in an HTTP Header"
- "A JMS Queue Name" (or topic name) for SOAP over JMS web services
- "A JMS Message Property" for SOAP over JMS web services
- "A JMS Message Header" for SOAP over JMS web services
- "A specific IP Address "
- "A Range of IP Addresses"

> **Note:** Defining consumer ID based on SOAP header, envelope, or body requires the Diagnostics SOAP message handler for Java probes. For some application servers, special instrumentation is provided in Diagnostics to automatically load the Diagnostics SOAP message handler.
>
> However, some manual configuration is required for WebSphere 5.1 JAX-RPC and Oracle 10g JAX-RPC, see "Loading the Diagnostics SOAP Message Handler " on page 70 for details.
>
> The Diagnostics SOAP message handler is not available for all application servers. Custom instrumentation is not available to capture SOAP faults or consumer IDs from SOAP payloads. Therefore, this feature is not available on all versions of all application servers. For the most recent information on Diagnostics SOAP message handler support, see the Diagnostics Support Matrix at Diagnostics_System_Requirements Guide.

Aggregating the data by consumer ID is useful if you want to determine who is using a particular service and how frequently they are using it. Consumer IDs are also useful for BSM/APM. BSM/APM users can look at the performance of the same application based on consumers to compare their performance characteristics.

Configuring Consumer IDs is optional. By default, IP address is used as consumer ID for SOAP over HTTP/S web services and inbound queue name (or topic name) is used by default as consumer ID for SOAP over JMS web services.

This section includes:

- "Basic Procedure for Consumer ID Configuration" below
- "About Consumer ID Rules" on the next page
- "Consumer ID Rules Syntax and Examples for Java Agents" on page 178

Basic Procedure for Consumer ID Configuration

The basic procedure to configure consumer IDs is as follows:

1. (Optional). Specify ***dump-payload** in the **consumer.properties** file to print the entire SOAP payload out to the **consumer.log** file. Use this output to plan how to create the specific rules to configure consumer IDs for SOAP payload capture.

   Before you configure consumer IDs, familiarize yourself with the SOAP payload data to determine how best to create the specific rules Diagnostics will use to find the value for consumer IDs.

   The dump-payload option should only be used when help is required to locate the element that contains the Consumer Id.

This option should be the only value on the right side of the equal(=)sign when used: `DumpTest;HTTP_WS;TraderService = *dump-payload`

> **Note:** Do not try to use the same service name to extract a value AND dump the payload at the same time.

For example, to use this feature, enter:

```
SoapTest1;HTTP_WS;TraderService = *dump-payload
```

This results in printing the SOAP Payload for a rule that matches TraderService. The content of the consumer.log file is:

```
2009-01-15 14:42:13,653 INFO consumer [[ACTIVE] ExecuteThread: '0' for queue:
'weblogic.kernel.Default (self-tuning)'] [PAYLOAD:] <?xml version="1.0"
encoding="UTF-8"
standalone="yes"?><soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:trad="http://
www.bea.com/examples/Trader" xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/">
 <soapenv:Header>
  <CallerA>customerA</CallerA>
 </soapenv:Header>
 <soapenv:Body>
  <trad:buy soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <string xsi:type="xsd:string">hpq</string>
   <intVal xsi:type="xsd:int">11</intVal>
  </trad:buy>
 </soapenv:Body>
</soapenv:Envelope>
```

2. For each Java Agent you want metrics grouped by consumer, update the **consumer.properties** file as described in "Consumer ID Rules Syntax and Examples for Java Agents" on the next page.
3. To track more than five consumer types, update the **max.tracked.ids.per.probe** setting in the **dispatcher.properties** file.
4. Review the **<probe_name>_id.properties** file located in the **probe/files/log** directory. The **<probe_name>_id.properties** file might need to be completely deleted or modified to match the **consumer.properties** changes made in the previous steps. The file goes together with the max.tracked.ids.per.probe (dispatcher.properties) setting, once the limit is reached, per probe, all other consumers are classified as "Other".

About Consumer ID Rules

The assignment of consumer IDs is controlled by consumer ID rules in a configuration file, **consumer.properties**.

Each category of consumer IDs has its own rules: SOAP rules, HTTP header rules, JMS web service rules, and IP rules. The rules are not applied according to how the rules are defined. The SOAP header rules are

applied first; the HTTP headers rules are applied next; then the JMS rules are applied; and lastly the IP rules are applied.

> **Note:** ALL configuration items in the rules are case sensitive. For example, if you enter a <pattern-name> of TraderService, the Web service name must have a capital T and a capital S for the pattern to match.

All rule types do not need to be used. There might be SOAP rules, no HTTP rules, and IP rules. If there is no match on any of these rules, the original IP address or queue name for JMS is used as the consumer ID.

The SOAP rules allow for the consumer ID to be obtained from an XML element in the SOAP header, SOAP envelope, or body as well. The rule specifies a regular expression that is used to match against the web service name being called by the consumer.

If there is a match, the probe attempts to find the text element also specified in the rule. If the text element is not found in the SOAP header, this rule is skipped and the probe goes on to the next rule that is defined.

The HTTP header rules allow for the consumer ID to be obtained from a header in the collection of HTTP headers in a HTTP request.

The JMS web service rules allow for the consumer ID to be JMS queue/topic name, and JMS Message properties or Message Header (JMSReplyTo only).

The IP rules allow for the consumer ID to be obtained from the mapping of IP addresses to a consumer ID. The rule is used to define an IP address, or a range of addresses, to be assigned to a consumer ID.

Consumer ID Rules Syntax and Examples for Java Agents

The assignment of consumer IDs is controlled by specifying rules in the **consumer.properties** file.

> **Note:** ALL configuration items are case sensitive. For example, if you enter a <pattern-name> of TraderService, the Web service name must have a capital T and a capital S for the pattern to match.

A Value in a SOAP Header

To assign a consumer ID based on a value in a SOAP header, use the following format:

```
<rule-name>;HTTP_WS;<pattern-name> = soap-header;<element-value>
```

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<pattern-name> is a regular expression to match on the Web service name or you can use the exact Web service name.

<element-value> the element in the SOAP envelope whose value you want to use as the Consumer ID.

For example, the following rule matches on a Web service with service name TraderService and uses the CallerA element’s value as the consumer IDs:

```
SoapRule1;HTTP_WS;TraderService = soap-header;CallerA
```

When the callers of the TraderService Web service have a value defined for CallerA, the metrics are grouped by the different values for CallerA. The following excerpt from the soap header maps to a consumer ID of "Customer2" for this caller of the TraderService:

```
SoapTest1;WS<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/
envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <env:Header>
  <CallerA>Customer2</CallerA> <---- The consumer id returned would be
 "Customer2"
 </env:Header>
 <env:Body env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <m:sell xmlns:m="http://www.bea.com/examples/Trader">
   <string xsi:type="xsd:string">sample string</string>
   <intVal xsi:type="xsd:int">100</intVal>
  </m:sell>
 </env:Body>
</env:Envelope>
```

By default, Diagnostics looks for CallerA in the first-level element (the element directly under the SOAP env:Header). You can configure Diagnostics to look into a deeper-level xml element for consumer ID. The dynamic property **max.search.level.depth** in the **consumer.properties** file controls the depth at which to search for consumer ID (default value is 1 level deep). For example, max.search.level.depth = 2 would find consumer ID:

```
<env:Header>
 <test:id>
  <test:CallerA>consumerA</test:CallerA>
 </test:id>
</env:Header>
```

A Value in a SOAP Envelope

To assign a consumer ID based on a value in a SOAP envelope, use the following format:

`<rule-name>;HTTP_WS;<pattern-name> = soap-envelope;<element-value>`

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<pattern-name> is a regular expression to match on the Web service name or you can use the exact Web service name.

<element-value> the element in the SOAP envelope whose value you want to use as the Consumer ID.

A Value in the SOAP Body

To assign a consumer ID based on a value in the SOAP body, use the following format:

`<rule-name>;HTTP_WS;<pattern-name> = soap-body;<element-value>`

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<pattern-name> is a regular expression to match in the Web service name or you can use the exact Web service name.

<element-value> the element in the SOAP body whose value you want to use as the Consumer ID.

A Value in an HTTP Header

To assign a consumer ID based on a value in an HTTP header, use the following format:

```
<rule-name>;HTTP_WS;<pattern-name> = attribute;<header-value>
```

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<pattern-name> is a regular expression to match on, in the URI.

<header-value> is the HTTP header whose value you want to use as the Consumer ID.

For example, the following rule matches on a web service with a URI of "/webservice/.*" and uses the "User-Agent" header's value as the consumer ID:

```
WsRule1;HTTP_WS;/webservice/.* = attribute;User-Agent
```

When the callers of the Web service have a value defined for User-Agent, the metrics are grouped by the different values for User-Agent. The following excerpt from the HTTP header maps to a consumer ID in the heading:

```
GET /service/call HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 2000)
Host: ovrntt1
Caller: ovrntt1
Connection: Keep-Alive
```

A JMS Queue Name

To assign a consumer ID based on the matching the JMS queue/topic name, use the following format:

```
<rule-name>;JMS_WS;<queue-name>=<consumerID-string>
```

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<queue-name> is a regular expression to match on, in the JMS queue/topic name.

<consumerID-string> is a literal string to use as the Consumer ID.

For example, the following rule matches on a JMS queue name that starts with queue://sca_soapjms.* and uses the string "myJMSConsumer" as the consumer ID:

```
JMSTest3;JMS_WS;queue\://sca_soapjms.*=myJMSConsumer
```

You must use a backslash "\:" to escape the ":" after queue or topic.

The priority used in matching is determined by the order specified in the consumer.properties file. JMS_WS queue matching takes priority over IP matching; JMS_WS property matching takes priority over JMS_WS Header matching; and JMS_WS Header matching takes priority over JMS_WS queue name matching.

A JMS Message Property

To assign a consumer ID based on matching a JMS queue/topic name and use the value from the JMS message property as the consumer ID, use the following format:

```
<rule-name>;JMS_WS;<queue-name>=jms-property;<property-value>
```

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<queue-name> is a regular expression to match on in the JMS queue/topic name.

<property-value> is the JMS property whose value you want to use as the Consumer ID.

For example, the following rule matches on a JMS queue name that starts with queue://MedRec.* and uses the value from the JMSXDeliveryCount property as the consumer ID:

```
JMSTest1;JMS_WS;queue\://MedRec.*=jms-property;JMSXDeliveryCount
```

You must use a backslash "\:" to escape the ":" after queue or topic.

A JMS Message Header

To assign a consumer ID based on matching the JMS queue/topic name and JMS message header, use the following format:

```
<rule-name>;JMS_WS;<queue-name>=jms-header;<header-value>
```

Where:

<rule-name> is a String that identifies the rule. The name must be unique to the consumer.properties file.

<queue-name> is a regular expression to match in the JMS queue/topic name.

<header-value> must be JMSReplyTo.

For example, the following rule matches on a JMS queue name that starts with queue://MedRec.* and uses the value from the JMSReplyTo header as the consumer ID:

```
JMSTest1;JMS_WS;queue\://MedRec.*=jms-header;JMSReplyTo
```

You must use a backslash "\:" to escape the ":" after queue or topic.

A specific IP Address

To assign a consumer ID based on an IP Address, use the following format:

```
<rule-name>; IP; <IP-address> = <consumerID-string>
```

For example, the following rule matches on IP address 123.456.567.8 and uses the name "CustomerA_IP" as the consumer ID:

```
IPRule1;IP;123.456.567.8 = CustomerA_IP
```

A Range of IP Addresses

To assign a consumer ID based on a range of IP addresses, use the following format:

```
<rule-name>; IP; <IP address range> = <consumerID-string>
```

where <IP address range> can be defined with integers, wildcards specified with *, integer range specified with -.

For example, the following rule matches all IP addresses whose first octet is 15 and uses the name "mySuperCluster" as the consumer ID:

```
IPRule2;IP;15.*.*.* = mySuperCluster
```

The following rule matches all IP addresses whose first octet is 15 and whose second octet is between 200 and 300; it uses the name "Customer_IP" as the consumer ID:

```
IPRule3;IP;15.200-300.*.* = Customer_IP
```

Configuring SOAP Fault Payload Data

If a SOAP fault is detected, the SOAP payload can be included with the SOAP fault data. SOAP payload is only captured when there is a SOAP fault.

In the Diagnostics UI, you can view the payload information as part of the instance tree. Both JAX-WS and JAX-RPC web services are supported.

Because payloads can contain sensitive information such as credit card numbers, payload capture on SOAP faults is *disabled* by default.

To enable payload capture on SOAP fault set **max.soap.payload.bytes** to a value greater than zero , 5000 is recommended, in the **dispatcher.properties** file on the Java agent.This is the number of bytes captured, so if the payload you see in the UI indicates it is too small you can increase this number. By default the value is set to zero to disable payload capture.

Capturing SOAP payload requires the Diagnostics SOAP message handler for Java probes. For some application servers, special instrumentation is provided in Diagnostics to automatically load the Diagnostics SOAP message handler. Manual configuration is required for WebSphere 5.1 JAX-RPC and Oracle 10g JAX-RPC. See "Loading the Diagnostics SOAP Message Handler " on page 70 for details.

The Diagnostics SOAP message handler is not available for all application servers, nor is custom instrumentation available to capture SOAP faults or consumer IDs from SOAP payloads. Therefore, this feature is not available on all versions of all application servers. For the most recent information on Diagnostics SOAP message handler support, see the Diagnostics Support Matrix at Diagnostics_System_ Requirements Guide.

For a Java Agent, define the limit for the payload size by modifying the **<agent_install_ directory>\etc\dispatcher.properties** file. Payloads larger than the specified size are truncated.

For example, the following entry increases the SOAP payload length to 10000 from its default of 5000:

```
max.soap.payload.bytes = 10000
```

Set this property to 0 to disable this feature.

Configuring REST Services

You can configure REST style Web services to show up as regular Web Services in the Diagnostics UI. See the comments in the following file for configuration details: **<agent_install_directory>\etc\rest.properties**.

Currently, only HTTP is supported (no JMS).

Customizing Grouping JMS Temporary Queue/Topics

For reporting in Diagnostics, SOAP over JMS temporary queues are grouped into a single node. Diagnostics matches the queue/topic name to a list of regular expressions to find the temporary queue/topic names. The ones that match are replaced with either **queue:<probe-id>\TEMPORARY** or **topic:<probe-id>\TEMPORARY** according to the type.

The list of regular expressions used for this matching is in the **<agent_install_ directory>/etc/capture.properties** file. You can customize the list of regular expressions under the property **grouped.temporary.jms.names.**

Configuring SQL Query Parsing

If there are a large number of SQL queries using literals it can overwhelm the server symbol table. In these situations you can configure the **sql.parsing.mode** property in the **dispatcher.properties** file on the Java Agent. The possible mode settings are as follows:

1 - just methods, no SQL queries.

2 - main categories for SQL queries (select/update/insert/delete/...).

3 - (default) a measurement per whole SQL query aggregating similar statements into a single measurement (ignore literals, keyword case...).

4 - a measurement per whole SQL query aggregating only identical statements.

```
sql.parsing.mode = 3
```

Another property in the **dispatcher.properties** file can be used to limit the number of different SQL statements collected in case of temporary database tables, allowing you to fold down the table names using an SQL statement regular expression substitution. The property is **sql.pattern.replace (**see the comments in the **dispatcher.properties** file for more information).

Capturing SQL Parameters

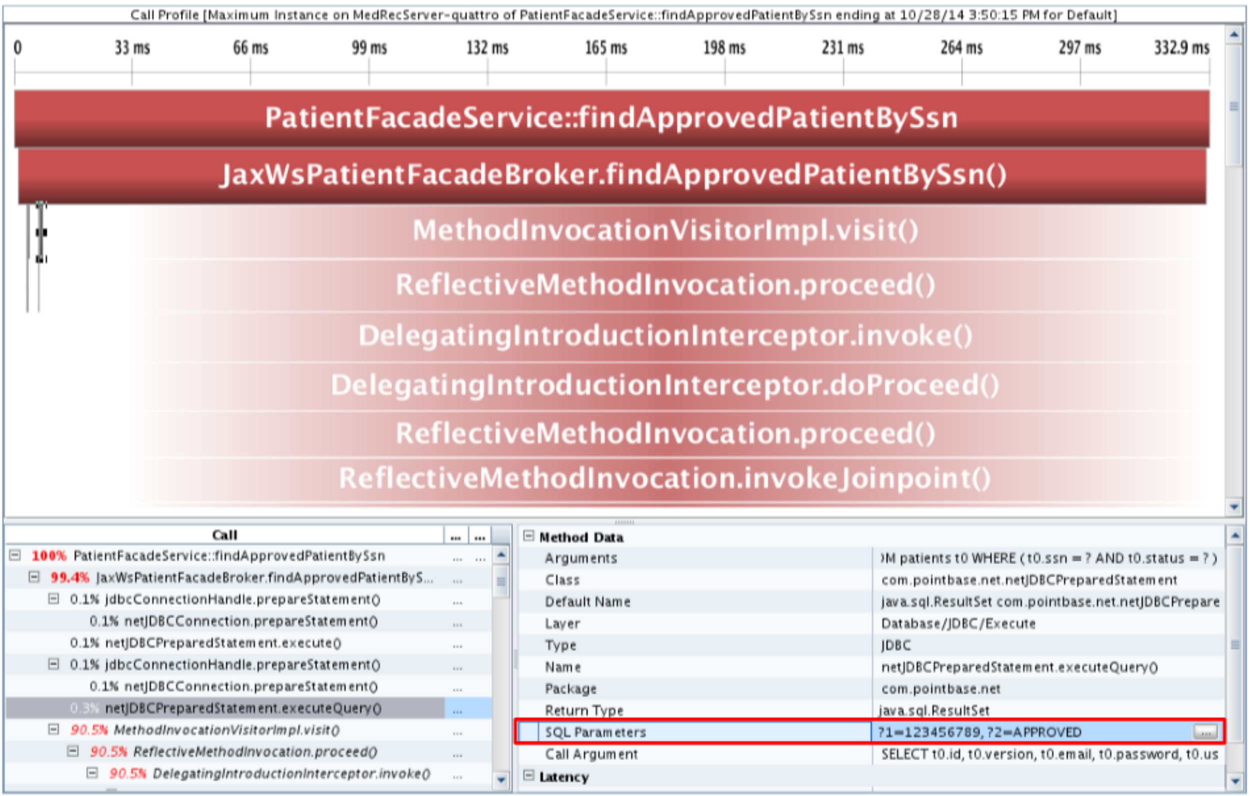
For increased efficiency, applications use prepared statements when repeatedly issuing the same query to a database. Such prepared statements can contain parameters, the value of which are set by the application before the query is actually executed. By default, for the predefined monitoring profile 120, these parameter values are not captured and cannot be viewed by Diagnostics.

You can change the default setting so that parameter values in an SQL query are captured and displayed in the Call Profile view. To change the default setting, edit the **/etc/capture.properties** file and set the value of the **sql.parameters.capture.enabled** property to **true**. Note that this is a dynamic property that you can change at any time.

Note:

- By default, this property is set to **false** for the predefined monitoring profile 120 and **true** for other predefined monitoring profiles.
- You can also enable SQL parameter capturing in the UI. To do this:
  a. Click **View Probe Configuration in New Window**.
  b. Select the **Enable SQL Parameter Capture** check box.
  c. Click **Apply Changes**.

When captured, you can view the parameter values in the Call Profile view. The parameters values are displayed in the **SQL Parameters** row (part of the Method Data section in the Details pane). For example:



Troubleshooting

- Only the first 32 parameters for each prepared statement are captured,
- If setting **sql.parameters.capture.enabled** to **true** does not capture or display parameter values, check that:
  - the prepared statement uses parameters.
  - SQL parameter capture has not been disabled (by setting the **mercury.enable.prepared_ statement.parameter.capture** setting in the **etc/inst.properties** file to **false**).
  - the current monitoring profile for the probe is at least 120.
  - the type of the argument has a natural String representation (binary data cannot be captured).

Configuring Display of Application Name for Server Requests

The **Deployed Into** value displayed in the Diagnostics UI in the Server Requests details pane can show the application name of the server request for most application servers. Prior to Diagnostics 9.0 this information was only available for WebLogic application servers so only a WebLogic probe could fill in the application name identifier on a server request.

To ensure backward compatibility with the server request trend lines, by default the application name is not filled in for the server request, except in WebLogic server requests.

This is configurable using the **fragment.use.application.name** property in the **capture.properties** file and you can set the following values for this property:

- **none.** The discovered application name is never used for identification purposes.
- **default.** Only WebLogic application server probes use the discovered application name.
- **all.** All application server probes use the discovered application name.

Note: Regardless of the value of this property, if a server request's J2EE application name is discovered, it will be used to populate the non-identifying property Topology Information.

Maintaining Probe Settings from the Java Profiler UI

You can use the Configuration tab in the Java Diagnostics Profiler to maintain the instrumentation points and edit the probe configuration without having to manually edit the Java Agent capture points file or property files. You can access the Configuration tab from the Java Diagnostics Profiler whether profiling has been started or not. For details, see "Configuration Tab Description" on page 260.

The Probe Settings section of the Java Diagnostics Profiler Configuration tab enables you to configure probe settings for thread stack trace sampling, collection of CPU time metrics (using timestamping) and reporting collection leaks.

When you click **Apply Changes** on the Java Diagnostics Profiler Configuration tab, all the updates you made in the Probe Settings sections of the Configuration tab are applied to the capture points file and the property files.

Note: Your changes take effect immediately. There is no need to restart the application (or probe).

The following sections describe each of the Probe Settings sections:

"Configuring Thread Stack Trace Sampling" below

"Controlling CPU Timestamp Collection" on page 188

"Enabling and Configuring Collection Leak Reporting" on page 189

Configuring Thread Stack Trace Sampling

When asynchronous thread sampling is enabled, you can see, in the Call Profile view, which methods were executed during long running fragments even if no instrumented methods were hit during this time. See the Diagnostics User Guide chapter on Call Profiles for a screen shot showing the additional nodes added based on thread sampling.

Several properties enable and configure thread stack trace sampling.

The following properties are in **dynamic.properties**:

- **enable.stack.trace.sampling** – enables asynchronous thread stack trace sampling; possible values are false, auto (the default), and true.

  When the dynamic property enable.stack.trace.sampling is set to auto, stack trace sampling is enabled IF the probe is running on selected (certified) platforms and JVMs. For other JVMs, the setting must be set to true explicitly. Use caution because the JVM could generate errors or abort. See the Diagnostics Release Notes.
- **tardy.method.latency.threshold** – the minimum time that an instrumented method must run without hitting any instrumentation points before stack trace sampling is attempted for this method. The purpose of this property is mainly to control the overhead of sampling by limiting the stack trace collection to only the most interesting cases.

- **stack.trace.sampling.rate** – the time that must elapse before the next consecutive sampling attempt is made.

  Small values for **stack.trace.sampling.rate** cause frequent sampling and provide rich data but at the cost of increased overhead.

  The overhead caused by frequent sampling affects primarily the latency of server requests. The overall CPU usage by the probe can go up as well, but this effect is not as profound as the latency increase. For systems with many CPUs, the process CPU consumption can actually go down (not a good thing).
- **stack.trace.depth.max** – the limit for the depth of stack traces obtained from the JVM. You will most likely not need to adjust this value.

The following properties are in **dispatcher.properties**:

- **enable.stack.trace.aggregation** – a boolean property allowing the correlation thread to merge together nodes observed on more than one consecutive stack trace collected, unless there is proof that the nodes must not represent a single method invocation. When set to true, it could decrease the number of additional call tree nodes created, but could create a false impression that the number of calls to the additional nodes is known and is small. When set to false, it creates a node for each method and each stack trace it was visible on, creating a false impression that the number of calls to the nodes is known and is large. In fact, stack trace sampling cannot reveal the number of calls at all.
- **aggregated.stack.trace.validity.threshold** – if the enable.stack.trace.aggregation property is set to true, only the call tree nodes that stem from more than the **aggregated.stack.trace.validity.threshold** number of individual stack traces are reported. This setting controls noise elimination and memory footprint, especially on the server side.

  All of the properties can be dynamically changed so no restart of the application is required.

  You can change the first four properties (from dynamic.properties) remotely, using the Configuration tab in the Diagnostics Java Profiler. After making changes remember to apply all of the changes made using the Configuration tab by clicking **Apply Changes**. For details, see "Configuration Tab Description" on page 260.

Example Thread Sampling Configurations

Use Case 1: A particular method has average latency of about 170 milliseconds, but from time to time it takes 1.4 seconds for this method to complete. Most of the methods visible in Call Profiles for any fragment execute in 550 milliseconds or less. Because the method in question makes multiple calls to its callees, you do not want to instrument them.

Instead you enable stack trace sampling to find out what the cause for long execution times. To minimize overhead, set tardy.method.latency.threshold to 600 milliseconds. This ensures that most of the methods will not get sampled at all because they are likely to complete before this time elapses. However, any method running longer than this value, including our long running method, will get sampled, once the method runs for 600 milliseconds (or longer) without making any calls to any of the instrumented methods.

If you also set the value of **stack.trace.sampling.rate** to 100 milliseconds, this should theoretically give up to eight samples for each method invocation that lasts 1.4 seconds ((1400–600) / 100). Because you know that the method makes many calls to its callees, you could also set **aggregated.stack.trace.validity.threshold** to zero. This ensures that even if each collected stack trace is completely different, they will all be reported.

If you examine the Call Profile for long running instances of the server request, you would see additional nodes revealed by stack trace sampling.

Use Case 2: You prepare a custom application for deployment and see that the default instrumentation provided with the Diagnostics agent does not work very well because many Call Profiles contain very few methods, which does not give any insight about the application specific behavior. You are reluctant to add

additional instrumentation for all classes and methods belonging to the custom application because of the performance and memory consumption concerns.

You enable stack trace sampling. Assuming that a typical server request that does not have sufficiently detailed call tree information runs in about 2 seconds, you select a **stack.trace.sampling.rate** of 200 milliseconds. This can give up to 10 stack traces per typical server request. However, you do not want all the stack traces to be reported because some of the methods visible in the stack traces can be very fast, and they do not substantially contribute to the server request's overall latency. Therefore, you set **aggregated.stack.trace.validity.threshold** to 2. This ensures that only methods visible in three or more consecutive stack traces, or methods with estimated latency of 600 milliseconds or more, will be reported.

After viewing the Call Profiles with the additional nodes obtained from sampling, you can make informed decision about adding additional instrumentation points to the probe configuration in deployment.

Troubleshooting Stack Trace Thread Sampling

Why do I not see any new nodes in my Call Profile after I enabled stack trace sampling?

See if any of the following applies to your case:

- Was the last method visible in the Call Profile an outbound call? Methods marked as outbound do not get sampled. (To reliably check if a method is marked as outbound, find this method in detailReport.txt file and check its corresponding instrumentation point detail for the "outbound" keyword).
- Was the last method visible in the Call Profile marked as no-layer-recurse Such methods do not get sampled. (Use the same procedure as in the previous point to check if a method is no-layer-recurse.)
- Did you try reducing tardy.method.latency.threshold or minimum.method.latency? It is possible that the last method visible in Call Profile makes calls that get trimmed, but they prohibit the sampling to kick in because there is never an inactive period of tardy.method.latency.threshold for the caller.
- Did you try reducing aggregated.stack.trace.validity.threshold or check if there are warnings in the probe.log file about the stack depth being too shallow? Possibly, the observed stack traces changed too quickly to get reported.
- Did you try reducing the stack.trace.sampling.rate? Perhaps your methods simply miss the opportunities to get sampled.
- Did you verify that the latency of the last visible method in Call Profile is not caused by having run garbage collector? Java code, including the stack trace sampling code, does not run during garbage collection.

What is the minimum value of stack.trace.sampling.rate that can be used?

You can use any positive value, but remember that each platform will refuse to sample more frequently that it possibly can. The three determining factors are the minimum granularity of sleep() available, the timer resolution, and the time it actually takes to collect one set of samples.

What is the maximum value of stack.trace.sampling.rate that can be used?

There is no limit. The usefulness of a high setting depends entirely on the latency of the server requests for the application. To get any results, plan for at least a few samples for each server request you are concerned with. Even that could require tuning other sampling parameters as well.

Controlling CPU Timestamp Collection

The CPU timestamps calculate the amount of exclusive CPU time that a method uses. You can view this information on the **Hotspots** tab in the Java Diagnostics Profiler.

Note: In VMware, the CPU time metric is from the perspective of the guest operating system and is

affected by the VMware virtual timer. See the VMware whitepaper on timekeeping at http://www.vmware.com/pdf/vmware_timekeeping.pdf and "Time Synchronization for Probes Running on VMware" on page 168.

By default, collection of CPU time metrics is enabled for server requests.

Collection of CPU time metrics can be configured in property files (see "Configuring Collection of CPU Time Metrics" on page 174) or using the Java Diagnostics Profiler UI (see "Configuration Tab Description" on page 260).

Enabling and Configuring Collection Leak Reporting

Note: You must run the JRE Instrumenter using the appropriate mode for your application server if you want to use the collection leaks pinpointing (CLP) feature in the Java Agent.

You can set the following configuration items for collection leak reporting using the Collection Leaks section in the Java Profiler Configuration tab (for details, see "Configuration Tab Description" on page 260).

These same values can also be set in the **dynamic.properties** file for the probe: **clp.diagnostics.reporting**, **clp.diagnostics.growth.time** and **clp.diagnostics.nongrowth.time**.

Generating Performance Reports for JUnit Tests

When you run JUnit tests, you can enable and configure the Java Agent so that it generates a performance report for all of your unit tests. This is useful for finding out if the performance (latency/CPU) of a particular test has changed over time.

When the unit test finishes, the Java Agent creates a CSV file for each test method (represented as a server request). This CSV file contains a complete listing of all test methods that were executed in each JVM instance, usually per test class. The CSV file can be opened in a spreadsheet program to analyze and visualize performance characteristics (the Filter function in Excel is very helpful for selecting specific methods).

Following is an example of a CSV file:

```
Date,Server Request,Avg Latency,Count,Min Latency,Max Latency,Cpu
Time,Exceptions
Fri Sep 23 12:55:22 PDT
2011,UT_SiSXmlDataReader.testDataSample(),1068.81,1,1068.81,1068.81,374.403,0
Fri Sep 23 12:55:40 PDT
2011,UT_SiSXmlDataReader.testDataSample(),1064.845,1,1064.845,1064.845,405.60
2,0
Fri Sep 23 12:55:57 PDT
2011,UT_SiSXmlDataReader.testDataSample(),1141.689,1,1141.689,1141.689,358.80
2,0
Fri Sep 23 12:56:27 PDT
2011,UT_SiSXmlDataReader.testDataSample(),1474.81,1,1474.81,1474.81,468.003,0
```

The latency times are in milliseconds (ms).

By default the data for each test execution is appended to the CSV files. This is especially useful when tests are run as part of a Continuous Integration cycle which allows you to capture results over time.

To use this functionality, enable the Java Agent in the JUnit test execution by specifying the following JVM parameters:

JVM Parameter	Description
-javaagent:<Java_Agent_Home>/DiagnosticsAgent/lib/probeagent.jar (UNIX) or -javaagent:<Java_Agent_Home>\DiagnosticsAgent\lib\probeagent.jar (Windows)	Enables the agent by specifying the path to the agent JAR file.
-Ddispatcher.ac.autostart=true	Tells the agent to start profiling immediately.
-Dcapture.exit_report=dir=perftest:append	Instructs the agent to produce a performance report to the specified directory and to append the results. (To override the file, replace **append** with **override**.)
-Ddispatcher.minimum.fragment.latency=1ms	Collects only server requests (such as execution of JUnit test methods) that have latency above 1ms.

The following example shows an integration into ANT:

```
<junit dir="${build}" fork="yes" forkmode="perTest" printsummary="yes"
jvm="${env.JAVA_HOME}/bin/java">
 ...
 <jvmarg value="-javaagent:C:/MercuryDiagnostics/JavaAgent/DiagnosticsAgent/lib/
probeagent.jar"/>
 <jvmarg value="-Ddispatcher.ac.autostart=true"/>
 <jvmarg value="-Dcapture.exit_report=dir=<dir_name>:append"/>
 <jvmarg value="-Ddispatcher.minimum.fragment.latency=1ms"/>
 ...
</junit>
```

In addition to the above settings, the JUnit point needs to be activated (set **active=true**) in **<Java_Agent_Home>/DiagnosticsAgent/etc/auto_detect.points**:

```
[JUnit]
class = junit.framework.TestCase
method = !test.*
signature = !.*
deep_mode = hard
layer = JUnit
active = true
```

Note: If you use JUnit 4.x and your unit test classes are not a subclass of junit.framework.TestCase, you need to change the class definition in the above JUnit point to match your unit test classes.

Chapter 12: Java Agent Metrics Collectors

This chapter describes Java Agent metrics capture and how to configure the metric collectors.

This chapter includes:

- "About Metrics Capture" below
- "What Metrics are Being Collected by the Java Agent" on the next page
- "Understanding Metric Collector Entries" on the next page
- "About Collecting Additional Probe Metrics" on page 194
- "Modifying Probe Metrics Already Being Captured" on page 194
- "Stopping Capture of a Metric" on page 194
- "Using Customized metrics.config Files for Multiple JVM Applications on a System" on page 194

About Metrics Capture

With the Java Agent you can configure metrics collectors by modifying the entries in the metrics configuration file, **<agent_install_directory>/etc/metrics.config.**

> **Note:** There is a different metrics.config file included with the .NET Agent .

The system and JMX metric collectors for your agent installation are defined in the metrics configuration file. The properties and entries in the metrics configuration file, **<agent_install_directory>/etc/metrics.config**, enable you to control the metric collectors.

> **Note:** If you update the metrics configuration file, the metric collectors automatically restarts so that your changes can take effect.

What Metrics are Being Collected by the Java Agent

In the **metrics.config** file you can see what metrics are being collected by the Java Agent.

```
metrics.config - Notepad
File  Edit  Format  View  Help

################################
### Collector configuration ###
################################


################################
####  Basic JVM collector
JVM/jvmHeapUsed = HeapUsed|bytes|Probe
JVM/jvmHeapFree = HeapFree|bytes|Probe
JVM/jvmHeapTotal = HeapTotal|bytes|Probe
JVM/jvmheapusedpct = HeapUsedPct|percent|Probe
################################

################################
####  Process Metrics Collector
ProcessMetrics/processCpuUtil=ProcessCpuUtil|percent|Probe
ProcessMetrics/processCpuUtilAbs = ProcessCpuUtilAbs|percent|Probe

################################
####  System metric collector

system/CPU = CPU|percent|System
system/MemoryUsage = MemoryUsage|percent|System
system/VirtualMemoryUsage = VirtualMemoryUsage|percent|System
system/ContextSwitchesPerSec = ContextSwitchesPerSec|count|System
system/DiskBytesPerSec = DiskBytesPerSec|bytes|Disk
system/DiskIOPerSec = DiskIOPerSec|count|Disk
system/NetworkBytesPerSec = NetworkBytesPerSec|bytes|Network
system/NetworkIOPerSec = NetworkIOPerSec|count|Network
system/PageInsPerSec = PageInsPerSec|count|System
system/PageOutsPerSec = PageOutsPerSec|count|System
```

Listing Available Metrics

The Java Agent metrics.config file has a feature to write a list of all the available metrics for each JMX collector into a file. When the **default.dump.available.metrics** property in the metrics.config file is set to true, the probe will write this list of available metrics to text files in the probe log directory. The files are named as follows: **<agent_install_directory>/log/<probe-id>/jmx_metrics_<collector-name>.txt**. See "Getting a List of Available JMX or WebSphere PMI Metrics" on page 204 for details and examples of how to use this information as a template for configuring additional metrics capture.

Understanding Metric Collector Entries

Metric Collector entries instruct the Java Agent metric collectors to gather specific metrics. The parameters on the left hand side of the entry control how the probe gathers the metric from the host or the JVM, and the parameters on the right hand side of the entry define how the collected metrics are processed in Diagnostics and displayed in the user interface.

The entries can have one of the following layouts:

```
<collector_name>/<metric_config>=<metric_id>|<metric_units>|<category_id>
```

or

```
<collector_name>/<metric_config>=
RATE<rate_multiplier>(<metric_id>|<metric_units>|<category_id>)
```

where:

- **<collector_name>** indicates the name of the Diagnostics metric collector. The collectors are defined in **metrics.config**.

  For system metrics the value of this parameter is `system`. For JMX metrics the value of this parameter is usually defined as the name of the application server type and the version, such as `WebSphere5`.

  The collector-name along with metric names can also be found on the Advanced Query page in the Diagnostics UI (http://<diagnostics_sever>:2006/query).

- **<metric_config>** identifies the metric that is to be monitored on the host system or on the JVM for the application server. The format of this parameter varies depending on whether you are creating an entry for a system metric or a JMX metric. For information on formatting the **metric_config** property for the system metric collector, see "Capturing Additional Custom System Metrics" on page 198. For information on formatting the **metric_config** property for JMX metrics, see "Creating New JMX or WebSphere PMI Metrics Entries" on page 206.

- **RATE(...)** indicates that metric values are converted to a rate (units per second) during sampling.

  For example, when the **Rate** parameter is used with the metric **total servlet requests since startup,** the value of the collected metric is converted from a *count of servlet requests* to the *number of servlet requests per second*.

  When **Rate** is not used, omit the parenthesis as shown in the first example above.

  Note: This parameter should only be used for metrics with non-decreasing values.

- **<rate_multiplier>** is an optional parameter that indicates that the rate is to be adjusted by multiplying it by the **<rate_multiplier>**.

  For example, when the **Rate** parameter and the **rate_multiplier** are used with the metric **total gc time** (in ms), the value of the metric collected is converted from *the total time for gc* to the *percent time spent in gc.*

- **<metric_id>** indicates the name that represents the metric in the UI. The metric_id must be unique in the **metrics.config** file. If the value of the metric_id is the same as one of the default metrics, Diagnostics replaces the metric_id in the entry with a standard name to be used to reference the metric in the UI. If the value of the metric_id is not the same as one of the default metrics, the metric_id is used as the name of the metric in the UI exactly as shown in the entry.

- **<metric_units>** indicates the units of measure in which the metric is reported. This is a required parameter and it must contain one of the following units of measure:

  - microseconds, milliseconds, seconds, minutes, hours, days
  - bytes, kilobytes, megabytes, gigabytes
  - percent, fraction_percent
  - count
  - load

- **<category_id>** groups a set of metrics together under the same heading in the tree in the side bar of the Metrics tab in the Java Diagnostics Profiler. This parameter has no impact on the data displayed in the Details pane in the Diagnostics UI views.

Note: After you create the metric collector entry, add the escape character "\" before each occurrence of a back-slash '\', space ' ', or colon ':'. This is a requirement for Java properties loaded from a file.

About Collecting Additional Probe Metrics

To gather information for an additional metric, add an entry for the metric to the appropriate metric collector in the **metrics.config** file using the syntax described in "Understanding Metric Collector Entries" on page 192.

See "Capturing Additional Custom System Metrics" on page 198 for details on capturing additional system metrics.

See "Additional Custom JMX Metrics" on page 204 for details on capture addition JMX metrics.

Modifying Probe Metrics Already Being Captured

You can update both the default and the custom metric entries in the metric collectors in the **metrics.config** file.

Stopping Capture of a Metric

To stop a metric collector from collecting a metric listed in **metrics.config,** you can either delete the metric entry or make the metric entry a comment line by adding a '#' to the beginning.

Using Customized metrics.config Files for Multiple JVM Applications on a System

There may be times when you only need to collect certain metrics, or customize the metric collector properties for select JVM applications running on a system with multiple JVMs, and such changes would negatively impact the other instrumented JVMs running on the system. In these cases, you can create and customize different **metrics.config** configuration files and configure those JVM applications to use the customized settings by following these steps:

> **Note:** You only need to configure the JVM applications that need customized **metrics.config** files. The other JVM applications can use the out-of-the-box **metrics.config** configuration.

1. Copy the **etc/metrics.config** file for each JVM application requiring special customization and name the file, such as **metrics_<app_name>.config**. This file must be in the same **<agent_install_directory>/etc** folder as the original **metrics.config** file. Customize this file as needed.
2. Create a copy of the **lib/modules.properties** file for each **metrics_<app_name>.config** file created, and name the file, such as **modules_<app_name>.properties**. This file must be in the same **<agent_install_directory>/lib** folder as the original **modules.properties** file.

   Change the **metrics.properties** property of this new file to point to the new **metrics_<app_name>.config** file as shown in the following example:

   ```
   #####################################################################
   ## Metrics capture module
   #####################################################################
   metrics.class.name=com.mercury.diagnostics.capture.metrics.MetricsModule
   metrics.class.loader=probeLoader
   ```

```
metrics.properties=metrics_<app_name>.config
```

3. Update each JVM start script that needs customized metrics collection to use the new corresponding **lib/modules_<app_name>.properties** file by adding the following to the JVM property definition:

   ```
   -Dmodules.properties.file=module_<app_name>.properties
   ```

Chapter 13: Java Agent - System Metrics Capture

Information is provided on the process for capturing system metrics and how to configure the Java Agent system metric collector to capture them.

This chapter includes:

- "About System Metrics" below
- "System Metrics Captured by Default" below
- "Configuring the System Metrics Collector" on the next page
- "Capturing Additional Custom System Metrics" on page 198

About System Metrics

The system metric collector is installed with the Java Agent. The system metric collector gathers system level metrics, such as CPU usage and memory usage, from the agent's host. The system metric collector is configurable so you can control which system metrics are collected.

Only one instance of the system metric collector is run on a given host, no matter how many instances of the probe were started on the host. When an instance of the probe is started, it attempts to connect to the UDP port specified in the metrics properties. If a connection is established, the system metric collector instance is started. If a connection cannot be made, a system metric collector instance has already been started on the host by another instance of the probe and a new instance cannot be started.

Each probe periodically attempts to connect to the port to make sure that a system metric collector is always running. If the probe that started the systems metric collector is stopped, one of the other instances of the probe will start a new instance of the systems metric collector when it finds that the port is available.

System Metrics Captured by Default

The following are the system metrics that the metric collector collects by default for all supported platforms:

- CPU
- MemoryUsage
- VirtualMemoryUsage
- ContextSwitchesPerSec
- DiskBytesPerSec
- DiskIOPerSec
- NetworkBytesPerSec
- NetworkIOPerSec
- PageInsPerSec
- PageOutsPerSec

You can control which of the default system metrics the system metric collector gathers and you can add other platform specific metrics so that the collector gathers the information for them as well. See "Configuring

the System Metrics Collector" on the next page for more information. For certain platforms, such as Windows, Solaris, and Linux, you can create custom system metrics that can be gathered by the system metric collector. For details, see "Capturing Additional Custom System Metrics" on the next page.

Configuring the System Metrics Collector

You can configure the system metrics capture process to run in your environment, and to collect and report the system metrics that are of interest to you, by modifying the entries in the metrics configuration file, **<agent_install_directory>/etc/metrics.config**. See "Java Agent Metrics Collectors" on page 191 for general information on the metrics collector and see "Understanding Metric Collector Entries" on page 192 for an explanation of the metrics collector entries and syntax.

> **Note:** If you update the metrics configuration file, the systems metric collector automatically restarts so that your changes can take effect.

Example System Metrics Collector Entry

The following example shows how to create the metric collector entry for a system metric. To create an entry for a system metric called CPU on a host platform, you would enter the following:

```
system/CPU = CPU|percent
```

where:

- **system** indicates that the metric is to be collected by the system metric collector
- the first **CPU** indicates that the metric known as **CPU** on the platform, is being monitored
- the second **CPU** is the name that is to be used in the UI to label the metric
- **percent** indicates the units in which the metric is measured on the host, and reported in the UI

Modifying the Default Port

The default port for the metric collector is **35000**. This value can be modified using the **system.udp.port** property if the configuration for your agent host requires that another port be used.

To modify the default port:

1. Locate the **system.udp.port** property in **metrics.config**.
2. Change the value of the **system.udp.port** property to the number of the port that you want to be used by the system metric collector. The default port is **35000**.

> **Note:** The port assigned to the system metric collector is not related to the port for the agent's Web server.

Disabling System Metrics Collection

To disable the collection of system metrics so that they will not be collected or displayed in the UI, set the value of the **system.udp.port** property to -1.

Capturing Additional Custom System Metrics

You can capture custom system metrics on Windows, Solaris, and Linux platforms using the Java Agent system metric collector.

The following sections provide instructions for capturing the metrics and updating the entries in the system metric collector so that the custom metrics can be monitored.

This section includes:

"Capturing Custom System Metrics on Windows Hosts" below

"Capturing Custom System Metrics on Solaris Hosts" on page 200

"Capturing Custom System Metrics on Linux Hosts" on page 200

Capturing Custom System Metrics on Windows Hosts

Using the features of Windows System Monitor, you can add counters to represent the performance of specific aspects of a system or service. The counters are tracked and reported in the Windows System Monitor, and can be monitored by the Java Agent system metric collector.

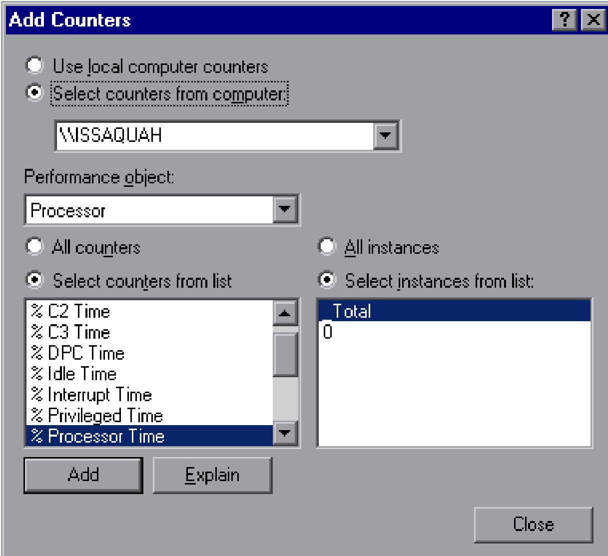
To add counters using the Windows System Monitor:

1. Start the Windows Performance Monitor:
   a. Execute **Run** from the Start menu.
   b. In the **Open** box on the **Run** dialog box type `perfmon`.

      The **Performance** dialog box opens showing the **System Monitor** graph with a table of the current counters beneath the graph.
2. Display the Add Counters dialog box:

   Right-click the **System Monitor** graph and select **Add Counters...** from the pop-up menu.

   Windows displays the **Add Counters** dialog box:



3. Make sure that the host computer is selected from **Select counters from computer** list.

4. In the **Performance object** list, select the object that the counter belongs to.
5. Choose **Select counters from list,** and select a counter from the list of counters that follows.
6. Choose **Select instances from list**, and select an instance from the list of instances that follows.
7. Click **Add**.

Once a counter has been added to the Systems Monitor, the system metric collector can be configured to gather the metrics for the counter. The following instructions will guide you through the steps to create an entry for the **metrics.config** based on the following template:

```
<collector_name>/<metric_config>= <metric_id>|<metric_units>
```

This template is described in "Understanding Metric Collector Entries" on page 192.

To collect metrics for a Windows System Monitor Counter:

1. Open **<agent_install_directory>/etc/metrics.config.**
2. Create the <metric_config> part of the entry using the following template, type the entry for the counter:

   ```
   \<performance_object>(<instance>)\<counter>
   ```

   In the example shown in the preceding screen image:

   - the selected Performance Object is `%Processor`
   - the selected Instance is `_Total`
   - the selected Counter is `Processor Time`

   The <metric_config> portion of the entry that would be created for this example would be:

   ```
   \Processor(_Total)\% Processor Time
   ```

3. Fill in the rest of the system metric entry template as shown in the following example:

   `system/\Processor(_Total)\% Processor Time = ProcessorTime|percent`

4. Format the initial entry by prepending a back-slash '\' before each occurrence of back-slash '\', space ' ', or colon ':' in the initial entry.

   Following this step, the initial entry in the previous step becomes:

   ```
   system/\\Processor(_Total)\\%\ Processor\ Time = ProcessorTime|percent
   ```

   This is the correctly formatted entry for **metrics.config** to enable the system metric collector to gather the metrics for a Windows System Monitor counter.

   ```
   system/\\\\RemoteMachine\\Processor(_TOTAL)\\%\ Processor\ Time=
   Processor\ Time(Remote Machine)|percent
   ```

   Note: Assuming **perfmon** is setup properly on a remote machine, you can use it to get metrics from remote machines by adding **\\MachineName** before the Performance object name as shown in the following example:

```
system/\\\\RemoteMachine\\Processor(_TOTAL)\\%\ Processor\ Time=Processor\ Time
(Remote Machine)|percent
```

Capturing Custom System Metrics on Solaris Hosts

The Solaris system metrics that can be monitored by the system metric collector are found using the **kstat** command. Only a subset of the metrics found using the **kstat** command can be monitored by the system metric collector.

To collect metrics for a Solaris system metric:

1. Execute the **kstat** command and identify the metric that you want to monitor**.**

   A Solaris system metric has the following format:

   ```
   module:instance:name:statistic
   ```

   Here is an example:

   ```
   vmem:35:ptms_minor:free
   ```

2. To cause the metric collector to gather the metrics for an additional system metric, add an entry for the metric to the system metric collector in the **metrics.config** file using the following template:

   ```
   <collector_name>/<metric_config>= <metric_id>|<metric_units>
   ```

   This template is described in "Understanding Metric Collector Entries" on page 192.

   Using this template, the example from the previous step would initially appear as follows:

   system/vmem:35:ptms_minor:free = Virtual Memory (35) Free | count

3. Format the initial entry by prepending a back-slash '\' before every back-slash '\', space ' ', or colon ':'.

   Following this step the initial entry in the previous step becomes:

   ```
   system/vmem\:35\:ptms_minor\:free = Virtual\ Memory\ (35)\ Free | count
   ```

   This is the correctly formatted entry for **metrics.config** to enable the system metric collector to gather the metrics for a Solaris systems metric.

Capturing Custom System Metrics on Linux Hosts

The Linux system metrics that can be monitored by the system metric collector are found in the **/proc** file system. To configure the system metric collector to gather custom Linux metrics, scan the**/proc** file system to locate the desired metric, and then create the system metric collector entry for the metric in **metrics.config** according to the location of the metric information.

To collect metrics for a Linux system metric:

1. Scan the **/proc** file system to locate the metric that you would like the Diagnostics system metric collector to monitor**.**

   To create the system metrics configuration entry in **metrics.config** for the Linux metric, you must explicitly specify where the value for the system metric is located. The location is specified using the following values:

   - **File name.** The name of the file where the metric information is located, including the path from the **/proc** directory.

- **Line offset.** A count of the number of lines in the file to the line where the system metric is located. The first line is counted as line 0.
- **Word offset.** A count of the number of words that the metric value is offset into the line in the file. The first word in the line is counted as line 0. The value at the specified offset must be an unsigned integer.

For example, if you wanted the system metric collector to monitor the SwapFree system metric so that you can see it displayed in the Diagnostics views, you would scan the **/proc** directory to locate the metric, and you would discover that the metric is located in the **meminfo** file. The layout of this file is as follows:

```
MemTotal: 515548 kB
MemFree: 1552 kB
Buffers: 41616 kB
Cached: 152084 kB
SwapCached: 46064 kB
Active: 402720 kB
Inactive: 75328 kB
HighTotal: 0 kB
HighFree: 0 kB
LowTotal: 515548 kB
LowFree: 1552 kB
SwapTotal: 1048568 kB
SwapFree: 779192 kB
Dirty: 4544 kB
Writeback: 0 kB
Mapped: 300056 kB
Slab: 28764 kB
Committed_AS: 801364 kB
PageTables: 3184 kB
VmallocTotal: 499704 kB
VmallocUsed: 2184 kB
VmallocChunk: 497324 kB
HugePages_Total: 0
HugePages_Free: 0
Hugepagesize: 4096 kB
```

The location of the SwapFree metric in this file would lead to the following values:

- **File name:** meminfo
- **Line offset:** 12
- **Word offset:** 1

2. To gather the metrics for an additional system metric, add an entry for the metric to the system metric collector in the **metrics.config** file using the following template:

```
<collector_name>/<line>:<word>:<file>= <metric_id>|<metric_units>
```

This template is a version of the template described in "Understanding Metric Collector Entries" on page 192. The **<metric_config>** property has been replaced with the properties **<line>:<word>:<file>**.

Using this template, the example from the previous step would initially appear as follows:

```
system/12:1:meminfo = Swap Free | kilobytes
```

3. Format the initial entry by prepending a back-slash '\' before every back-slash '\', space ' ', or colon ':'.

   Following this step the initial entry in the previous step becomes:

   ```
   system/12\:1\:meminfo = Swap\ Free | kilobytes
   ```

   This is the correctly formatted entry for **metrics.config** to enable the system metric collector to gather the metrics for a Solaris systems metric.

Chapter 14: Java Agent - JMX Metrics Capture

Information is provided on the process for capturing JMX metrics and how to configure Java Agent metric collectors to capture them.

This chapter includes:

- "About JMX Metrics" below
- "About Configuring JMX Metric Collectors" on the next page
- "Additional Custom JMX Metrics" on the next page
- "Getting a List of Available JMX or WebSphere PMI Metrics" on the next page
- "Creating New JMX or WebSphere PMI Metrics Entries" on page 206

About JMX Metrics

The Java Agent comes with pre-defined JMX metric collectors that access the JMX metrics from the following application servers:

- IBM WebSphere
- BEA WebLogic
- SAP NetWeaver
- Oracle AS
- Apache Tomcat
- JBoss J2EE Server
- TIBCO Business Works

The Java Agent can also collect JMX data from any J2EE server that supports the JMX standard.

The Java Agent runs the JMX metric collectors periodically to collect the metrics from the application server. The collected metrics are displayed on the user interfaces in both Diagnostics Enterprise User Interface and the Diagnostics Java Profiler.

Configuring WebSphere for JMX Metric Collection

For WebSphere JMX metric collection, you might need to configure the Performance Monitoring Infrastructure (PMI) service on the WebSphere server to start receiving JMX metrics.

See "Configuring WebSphere for JMX Metric Collection" on page 55 for information on how to configure WebSphere 5.x, 6.x and 7.0 servers for JMX metrics collection.

Configuring TIBCO for JMX Metric Collection

For TIBCO JMX metric collection you need to enable JMX metric collection; see "Example 5: Configuring TIBCO ActiveMatrix BusinessWorks and Service Bus for Monitoring" on page 44 for instructions.

About Configuring JMX Metric Collectors

The JMX metric collectors are configurable so that you can control which JMX metrics are collected. The JMX metric collectors are defined in the **<agent_install_directory>/etc/metrics.config** file.

Typically a separate collector is defined for each major version of each application server.

See "Java Agent Metrics Collectors" on page 191 for general information on the metrics collector and see "Understanding Metric Collector Entries" on page 192 for an explanation of the metrics collector entries and syntax.

Additional Custom JMX Metrics

The Java Agent is installed with a number of predefined JMX metric collectors for the application servers listed in "About JMX Metrics" on the previous page. You configure these collectors by defining entries in the metrics.config file, see "Understanding Metric Collector Entries" on page 192. You could also create entries in the existing metric collectors and even create new collectors if there are additional JMX metrics that you would like Diagnostics to monitor.

In order to create new entries in the JMX metric collectors you can get a list of the available JMX metrics and WebSphere Performance Monitoring Infrastructure (PMI) metrics. Then you can create new metrics entries in the **metrics.config** file. The following sections provide instructions for creating new entries in the JMX metric collectors so that additional JMX metrics and PMI metrics can be monitored.

Getting a List of Available JMX or WebSphere PMI Metrics

The metric collectors installed with the Java Agent include entries for many of the JMX metrics that are available for each application server. However, there could be other JMX metrics or WebSphere PMI metrics that you could monitor, or new metrics could be exposed by the application server vendor.

In order to make it easier to configure new/additional JMX/PMI metrics for collection the metrics.config file has a feature to write a list of all the available metrics for each JMX collector into a file. When the **default.dump.available.metrics** property in the **metrics.config** file is set to true, the probe will write this list of available metrics to text files in the probe log directory. The files are named as follows: **<agent_install_directory>/log/<probe-id>/jmx_metrics_<collector-name>.txt**.

The **default.dump.available.metrics** property in the probe **metrics.config** file can be changed at runtime. It is recommended that the property is only set to true temporarily to write the list of available JMX/PMI metrics. After the metrics list is written to the file, the property should be set back to false (or commented out) to avoid the overhead of the probe periodically writing the metrics list to file.

Some examples of the metrics list file are shown below. You can use this type of information to configure additional JMX or PMI metrics in the probes' **etc/metrics.config** file.

The following example shows the available MBean ObjectNames and their collectable attributes:

```
======= MBean ObjectNames and Available Attributes =======
MBean ObjectName:
WebSphere:J2EEServer=server1,JDBCProvider=Derby JDBC
```

```
Provider,JDBCResource=Derby JDBC
Provider,Server=server1,cell=yli87Node01Cell,diagnosticProvider=true,j2eeType=JDB
CDataSource,mbe
anIdentifier=cells/yli87Node01Cell/nodes/yli87Node01/servers/server1/
resources.xml#DataSource_12442
31364323,name=WST_PriceGen,node=yli87Node01,platform=dynamicproxy,process=
server1,spec=1.0,
type=DataSource,version=6.1.0.0
Available Attributes:
name: loginTimeout, type: int
name: statementCacheSize, type: int
name: testConnectionInterval, type: java.lang.Integer
........................
```

The following example shows the available MBean ObjectNames and their collectable attributes and fields:

```
======= MBean ObjectNames and Available Attributes and Fields =======
MBean ObjectName:
java.lang:name=PS Old Gen,type=MemoryPool
Available Metrics:
Attribute: CollectionUsage type: javax.management.openmbean.CompositeData
Field: committed, type: java.lang.Long
Field: init, type: java.lang.Long
Field: max, type: java.lang.Long
Field: used, type: java.lang.Long
```

The following example shows the available MBean ObjectNames and their collectable operations and fields:

```
======= MBean ObjectNames and Available Operations and Fields =======
MBean ObjectName:
com.tibco.bw:key=engine,name="MortgageBroker-BrokerService"
Available Metrics:
Operation: java.lang.Integer GetActiveProcessCount()
Operation: javax.management.openmbean.CompositeData GetExecInfo()
Field: Threads, type: java.lang.Integer
Field: Uptime, type: java.lang.Long
Operation: javax.management.openmbean.CompositeData GetMemoryUsage()
Field: FreeBytes, type: java.lang.Long
Field: PercentUsed, type: java.lang.Long
Field: TotalBytes, type: java.lang.Long
Field: UsedBytes, type: java.lang.Long
```

For WebSphere JMX collectors, besides the generic MBean JMX metrics, the available WebSphere specific PMI metrics are also dumped to the WebSphere collector's dump file. This includes the PMI tree instance paths and their available statistics, and the PMI module configuration information as shown in the example below:

```
======= PMI Tree and Available PMI Statistics =======
```

```
connectionPoolModule
Available Statistics:
CreateCount, CloseCount, AllocateCount, ReturnCount, PoolSize, FreePoolSize,
WaitingThreadCount, FaultCount, PercentUsed, PercentMaxed, UseTime, WaitTime,
ManagedConnectionCount, ConnectionHandleCount, PrepStmtCacheDiscardCount,
JDBCTime
connectionPoolModule->Derby JDBC Provider
Available Statistics:
CreateCount, CloseCount, AllocateCount, ReturnCount, PoolSize, FreePoolSize,
WaitingThreadCount, FaultCount, PercentUsed, PercentMaxed, UseTime, WaitTime,
ManagedConnectionCount, ConnectionHandleCount, PrepStmtCacheDiscardCount,
JDBCTime
connectionPoolModule->Derby JDBC Provider->jdbc/ALBUM
Available Statistics:
CreateCount, CloseCount, AllocateCount, ReturnCount, PoolSize, FreePoolSize,
WaitingThreadCount, FaultCount, PercentUsed, PercentMaxed, UseTime, WaitTime,
ManagedConnectionCount, ConnectionHandleCount, PrepStmtCacheDiscardCount,
JDBCTime
```

Creating New JMX or WebSphere PMI Metrics Entries

The following instructions guide you through the process of creating the JMX or PMI metric entries according to the following template:

```
<collector_name>/<metric_config>= <metric_id>|<metric_units>
```

This template is described in "Understanding Metric Collector Entries" on page 192.

To capture JMX or WebSphere PMI metrics:

1. Open **<agent_install_directory>/etc/metrics.config.** and locate the JMX metric collector that is appropriate for the application that is being monitored by the Java Agent.
2. The **<collector_name>** parameter is the same as the rest of the entries in the collector. If you were creating an entry for WebLogic, the value of this parameter would be WebLogic.
3. Create the **<metric_config>** parameter.
   a. For JMX metrics the **<metric_config>** parameter is a pattern that the collector uses to find a matching MBean. The pattern consists of two components, separated by the '**.**' character. See syntax below.

      MBean object and attributes:

      ```
      <MBean object name pattern>.<attribute name>
      ```

      MBean Object, attribute and fields:

      <MBean object name pattern>.<attribute name>#<field name>

      MBean object and operations:

      ```
      <MBean object name pattern>.(<operationname>())
      ```

MBean object, operations and fields:

```
<MBean object name pattern>.(<operationname>()#<field name>)
```

Where

<MBean object name pattern> is the string representation of the object name of an MBean. For an explanation of metric patterns see "Understanding Metric Patterns" on the next page. For an explanation of how to group JMX metrics see "JMX GROUPBY and EXPAND_PMI Modifiers" on page 209.

<attribute name> is the name of the MBean attribute that represents the metric. If <attribute name> has any '.' in it, it should be surrounded by parenthesis: `<MBean object name pattern>.(<attribute name>)`

As an example, for a WebLogic application server, the <metric_config> parameter for the throughput of all **Execute Queues** is configured as:

```
*:Type=ExecuteQueueRuntime,*.ServicedRequestTotalCount
```

See "Getting a List of Available JMX or WebSphere PMI Metrics" on page 204 for an example of a metrics dump showing available attributes.

<attribute name>#<field name> JMX Attributes that return Composite Data can have their numeric fields used as metrics. Simply append the symbol # followed by the name of the field after the MBean name.

For example:

```
Java\ Platform/java.lang\:type\=MemoryPool,name\=Perm\ Gen.Usage#used
```

will track the <used> field of the <Perm Gen> MBean's <Usage> composite data attribute.

(<operationname>()) where the operation name is followed by open and close parentheses. And the entire operation name is enclosed in parentheses.If the operation returns a composite attribute, suffix the composite attribute field after the () as for attributes.

For example:

```
Tibco/com.tibco.bw\:key\=engine,name\=*.(GetActiveProcessCount()) = Active
Process Count|count|Tibco
```

Note that only operations that don’t take arguments are supported.

(<operation name>()#<field name>) JMX Operations that return Composite Data can have their numeric fields used as metrics. Simply append the symbol # followed by the name of the field after the MBean name.

For example:

```
Tibco/com.tibco.bw\:key\=engine,name\=*.(getStatus()#Total\ Errors) = Total
Errors|count|Tibco
```

will track the "Total Errors" field of the Composite data object returned by the getStatus() operation.

b. For WebSphere PMI metrics, the **<metric_config>** parameter is a pattern that the collector uses to find the matching PMI statistics. The pattern consists of two components separated by the '**.**' character.

```
<PMI StatDescriptor>.<statistics name>
```

Where

<PMI StatDescriptor> is used to locate and access particular Stats in the WebSphere PMI tree. It can be either a PMI module name (for example, `webAppModule`), or a PMI module branch (for example, [`webAppModule][AccountManagement#AccountManagementWar.war`]

<statistics name> is the name of the PMI statistics that represent the metric. If statistics name has any '.' in it, it should be surrounded by parenthesis: `[webAppModule][AccountManagement#AccountManagementWar.war].(webAppModule.numLoadedServlets)`

See "Getting a List of Available JMX or WebSphere PMI Metrics" on page 204 for an example of the PMI module and PMI module branches and their available statistics names.

See "JMX GROUPBY and EXPAND_PMI Modifiers" on the next page for an example of how to group PMI metrics.

4. Fill in the rest of the JMX metric entry template as shown in the following example:

```
WebLogic/*:Type=ExecuteQueueRuntime,*.ServicedRequestTotalCount = RATE(Execute
Queues Requests / sec|count|Execute Queues)
```

5. Format the initial entry by prepending a back-slash '\' before every back-slash '\', space ' ', equals (=), or colon ':'.

   Following this step the initial entry in the previous step becomes:

```
WebLogic/*\:Type\=ExecuteQueueRuntime,*.ServicedRequestTotalCount = RATE(Execute
Queues Requests / sec|count|Execute Queues)
```

   This is the correctly formatted entry for a JMX metric collector to enable the collector to gather WebLogic JMX metrics.

Understanding Metric Patterns

For JMX metrics the <metric_config> parameter is a pattern that the collector uses to find a matching MBean; for example:

```
*:Type=ExecuteQueueRuntime,*.ServicedRequestTotalCount
```

In the example above, the object name is ***:Type=ExecuteQueueRuntime,***, which could actually resolves to many MBeans whose names have the **Type** component equal to **ExecuteQueueRuntime**. **ServicedRequestTotalCount** is an attribute name for which metric values will be collected by the JMX metric collector.

> **Note:** Current implementation of the JMX collector only supports attributes that are numeric in type (for example, long, integer, etc.).

The JMX metric collector first uses MBeanServer's query mechanism to find the matching MBeans for each object name provided in the configuration. For JMX metrics the object names are a pattern that the collector uses to find a matching MBean. For more details around the object names, see `http://docs.oracle.com/javase/7/docs/api/javax/management/ObjectName.html.`

Since MBean object names are patterns that can resolve into multiple MBeans, the JMX collector will validate all of the attribute names in the entry against all MBeans that match the pattern, and will aggregate the attribute values over the set of those matching MBeans. Of course, it is not always the case that the object name resolves into multiple MBeans. For example, the following object name resolves to a single MBean (on a WebLogic application server):

```
*\:Name\=weblogic.kernel.Default,Type\=ExecuteQueueRuntime,
*.ServicedRequestTotalCount
```

JMX GROUPBY and EXPAND_PMI Modifiers

You can use the optional GROUPBY modifier to create a separate metric for each matched group of MBean ObjectNames with the same value of the key specified by GROUPBY. In the probe's etc/metrics.config file, for JMX metrics that describe an MBean object name pattern there is an optional modifier GROUPBY that can be added, which tells a JMX-based collector to treat the metric_config as multi-instance expression:

```
collector_name/GROUPBY[oname_key]/metric_config = ...
```

The collector will find all MBeans matching the metric_config and create a corresponding metric for each of them using the object name key oname_key to provide unique naming by appending it to category_id.

```
WebSphere6/GROUPBY[name]/WebSphere\:type\=DataSource,*.statementCacheSize = JDBC
Statement Cache Size|bytes|JDBC DataSource
```

For example:

```
WebSphere6/connectionPoolModule.CreateCount = JDBC Connection Creates|count|JDBC
ConnectionPools

WebSphere6/[connectionPoolModule][Derby\ JDBC\ Provider][jdbc/ALBUM].AllocateCount =
JDBCConnection Allocates|count|JDBC ConnectionPools
```

Or, you may use the optional EXPAND_PMI modifier to group PMI metrics similar to how you group JMX metrics.

For PMI, the EXPAND_PMI modifier is specified to expand the PMI tree from the given module or StatDescriptor branch by the specified level. The expansion level "n" can be 1, 2, ..., or *, with the default level of 1 and * means expand all:

```
collector_name/EXPAND_PMI[n]/metric_config = ...
```

For example:

```
WebSphere6/EXPAND_PMI[*]/connectionPoolModule.AllocateCount = JDBC Connection
Allocates|count|JDBC ConnectionPools
```

creates "JDBC Connection Allocates" metric for each JDBC connection pool provider and for each DataSource of the provider.

Part 4: Using the Diagnostics Profiler for Java

Chapter 15: Diagnostics Profiler for Java

This chapter describes how to use the Diagnostics Profiler for Java:

- "About the Java Diagnostics Profiler" below
- "How the Java Agent Provides Data for the Java Profiler" on the next page
- "Java Diagnostics Profiler UI Navigation and Display Controls" on page 213
- "Analyzing Performance Using the Call Profile Window" on page 215
- "Thread Call Stack Trace Sampling" on page 219
- "Comparison of Collection Leak Pinpointing and LWMD" on page 221
- "Object Lifecycle Monitoring" on page 222
- "Heap Walker Memory Analysis Execution Steps" on page 224
- "Heap Walker Performance Characteristics" on page 227
- "How to Access the Java Diagnostics Profiler" on page 227
- "How to Enable LWMD for Collections Displays" on page 228
- "How to Enable Allocation Capture" on page 228
- "How to Enable Object Lifecycle Monitoring" on page 229
- "How to Analyze Object Allocation" on page 230
- "How to Enable Memory Analysis" on page 230

Diagnostics Profiler for Java UI Description:

- "Summary Tab Description" on page 232
- "Hotspots Tab Description" on page 234
- "Metrics Tab Description" on page 236
- "Threads Tab Description" on page 238
- "All Methods Tab Description" on page 242
- "All SQL Tab Description" on page 244
- "Collection Leaks Tab Description" on page 245
- "Collections Tab Description" on page 247
- "Exceptions Tab Description" on page 250
- "Server Requests Tab Description" on page 252
- "Web Services Tab Description" on page 254
- "Allocation/LifeCycle Analysis Tab Description" on page 256
- "Memory Analysis Tab Description" on page 258
- "Configuration Tab Description" on page 260

About the Java Diagnostics Profiler

The Diagnostics Profiler for Java is installed with the Java Agent. The Profiler runs in a separate UI and provides near real-time data, enabling you to pinpoint application performance bottlenecks.

You can use the different tabs in the Java Profiler to analyze method latency for the selected application. And you can analyze memory problems for the selected application using the memory diagnostics metrics displayed in the Java Profiler.

Special Features Available in the Profiler

Some of the information presented in the Java Profiler is also available in the Diagnostics enterprise UI. However the following features are only available in the Java Profiler. Many of these features are real time and so are enabled and viewed only in the Java Profiler.

- Dynamic instrumentation of a sampled method from the Java Profiler Call Profile (accessible from the Server Requests tab)
- Threads tab
- Allocation/Lifecycle Analysis tab
- Heap Breakdown tab (including the heap walker)
- Probe Configuration tab

How the Java Agent Provides Data for the Java Profiler

This section describes the way in which the Java Agent runs probes to monitor your application and how this data is displayed in the Java Diagnostics Profiler.

Monitoring Method Latency and Call Stacks

The Diagnostics Agent for Java (Java Agent) runs probes to monitor your application and keep track of the metrics for all of the instrumented methods that your application calls. As probes are monitoring, they capture the call stack for the three slowest instances of each server request. The probe also captures a call stack representing all call instances for a type of service request and calculates the aggregated latency

When a server request instance is encountered that is slower than one of the captured instances for the server request, the slower instance replaces one of the previously captured instances.

The Java Diagnostics Profiler displays metrics for all of the instrumented methods. You can drill down to the method instances that are included in the captured call stacks.

While you are analyzing the information displayed on the various tabs of the Java Diagnostics Profiler, you are working with the methods and call stacks captured from the time that the user interface was started. In the meantime, to minimize performance impacts, the probe continues to monitor your application, capture method metrics, and capture call stacks.

Monitoring Application Memory Use

The Java Diagnostics Profiler allows you to monitor your application's memory usage using one of the following methods:

- Collection Leak Pinpointing
- Lightweight Memory Diagnostics
- Heap Breakdown/Heapwalker

Collection Leak Pinpointing allows you to pinpoint Java collection related memory leak locations in Java applications. The data collection for this feature has very low overhead and so it can be used in a production environment.

Lightweight Memory Diagnostics allows you to monitor the collections that your application has created, and to identify the largest collections and the fastest growing collections.

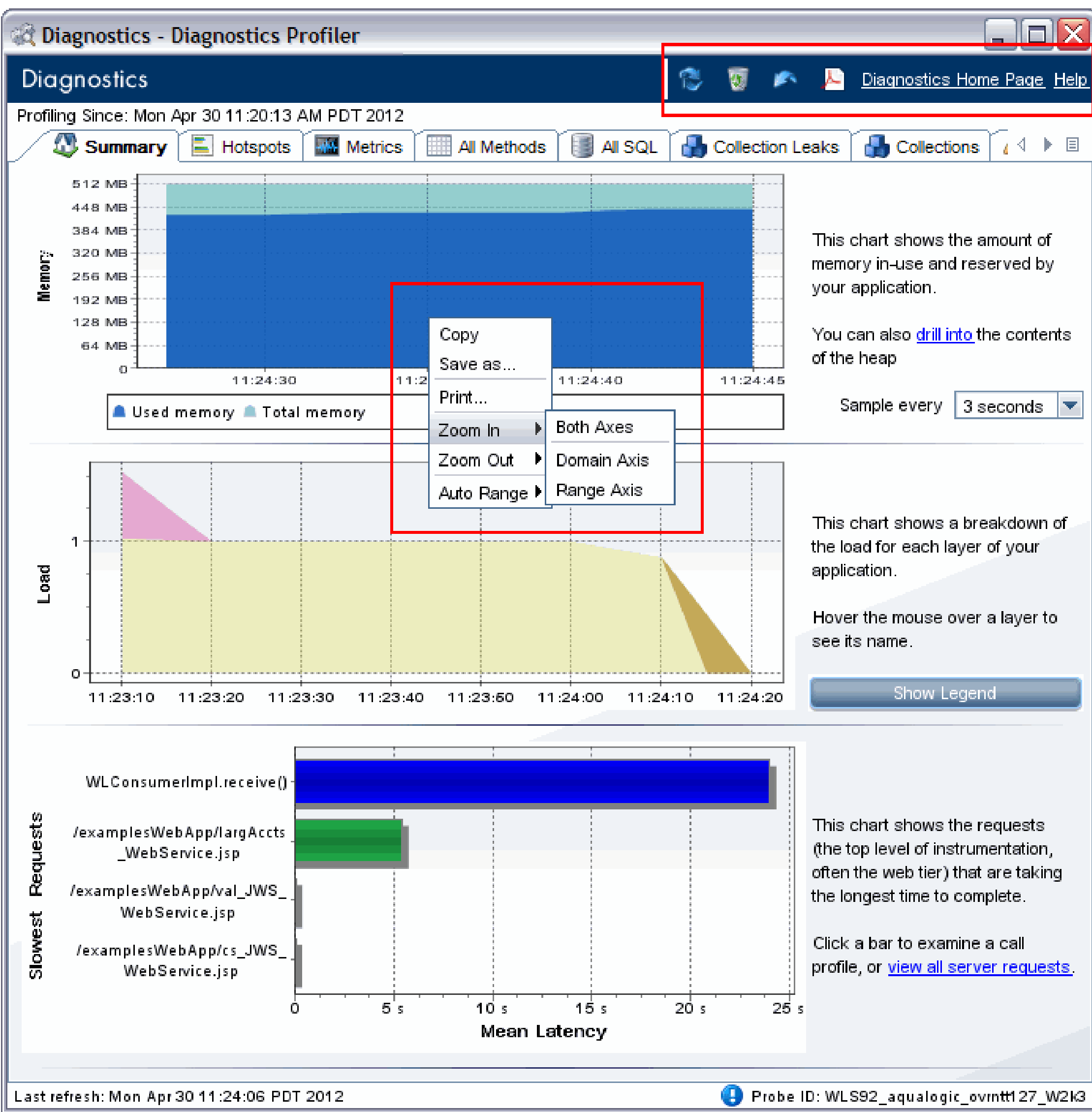
With Heap Breakdown you can monitor the heap generation breakdown and the objects that are stored in heap. This helps you to identify objects that may be leaking. By default, Lightweight Memory Diagnostics and Heap Breakdown are disabled.

For more information see "Comparison of Collection Leak Pinpointing and LWMD" on page 221. Also see "How to Enable LWMD for Collections Displays" on page 228.

For more information on Heap Breakdown/Heapwalker, see "How to Enable Memory Analysis" on page 230.

Java Diagnostics Profiler UI Navigation and Display Controls

This section describes the features and controls that are common within the different tabs of the Java Diagnostics Profiler:



Graph Menu Options (right-click in a graph to access menu)

Right click in a graph to access the graph menu and select an option:

Copy. From a graph, right-click and select Copy to copy the graph and paste it into a document. You can paste into any type of file that allows you to paste an image, such as a Microsoft Word file.

- **Save as.** From a graph, right-click and select Save as... to save the graph as an image (.png file type). Enter a file name in the dialog box displayed. By default the file is saved in My Documents but you can browse to the directory where you want to save the file.
- **Print.** From a graph, right-click and select Print to print the graph.
- **Zoom In.** From a graph, right-click and select Zoom In to zoom in for a closer look. Each time you select zoom in, it uses a multiplier of .5 to give you a magnified view of the data. Note that additional data is not retrieved and the resolution of the data is not changed.
  You can also select portion of the graph for zooming in. Using the mouse, click the graph where you want to begin the zoom and hold the left mouse button. Then drag the mouse to the right to select the zoom range. When you release the mouse the selected portion of the graph is zoomed.
  When zooming in you can select the following:
  Domain Axis - Select this option to zoom in and magnify the domain axis. Typically the domain axis is the time or X-axis.
  Range Axis - Select this option to zoom in and magnify the range axis. Typically the range axis is the axis with the data values or the Y-axis. For horizontal bar charts you only have the Range Axis selection and this zooms the axis with the data values, which in this case is the X-axis.
  Both Areas - Select this option to zoom in on both axes of the graph.
- **Zoom Out.** From a graph, right-click and select Zoom Out to zoom out for a less magnified view. Each time you select zoom out, it uses a multiplier of 2 to give you a less magnified view of the data. Note that the resolution of the data is not changed. The same menu options are available as for Zoom In (described above).
- **Auto Range.** From a graph, right-click and select Auto Range to go back to the original display after zooming in or out. You can select to restore the Domain Axis, Range Axis or Both Axes to the original magnification.

Refresh Metrics

When you are ready to view more current performance metrics, click **Refresh** on the top right corner of the screen to refresh the information displayed. The Profiler is refreshed with the latest metrics and call stacks. The system does not refresh itself automatically.

Reset Metrics

You can force the Java Diagnostics Profiler to use new baselines for the calculation of instance counts, average latency, and slowest latency, and to force-drop all captured call stacks, by clicking **Reset the Count and Time Information**.

> **Note:** You may want to reset metrics after your system has warmed up so that the metrics represent processing that takes place when your application is running in a more steady state.

Garbage Collection

When you want to deallocate used memory, you can forcibly perform garbage collection inside the JVM of the probed application by clicking **Force Garbage Collection** on the top right corner of the screen.

Export to PDF

When you want to export the page displayed, you can click the **Export this view to PDF (Acrobat)** icon

on the top right corner of the screen. See "Exporting Data" in the Diagnostics Server Installation and Administration Guide for details.

Diagnostics Home Page

The Diagnostics Home Page link displays the Diagnostics web site with information on products, solutions, demos, webinars and contact information for Micro Focus.

Accessing Help

When you click **Help**, on the top right hand corner of the screen, you access the Diagnostics Java Agent Guide.

Analyzing Performance Using the Call Profile Window

The Call Profile window (accessed from the Server Requests tab) displays a graphical representation of the method call stack for a selected server request. The depicted server request can be an aggregation of all of the calls made to the selected server request or a single instance of the server request depending on the server request on which you drilled down to open the call profile window. The metrics depicted in the graphical representation of the call stack are also depicted in the Call Tree Table on the same tab.

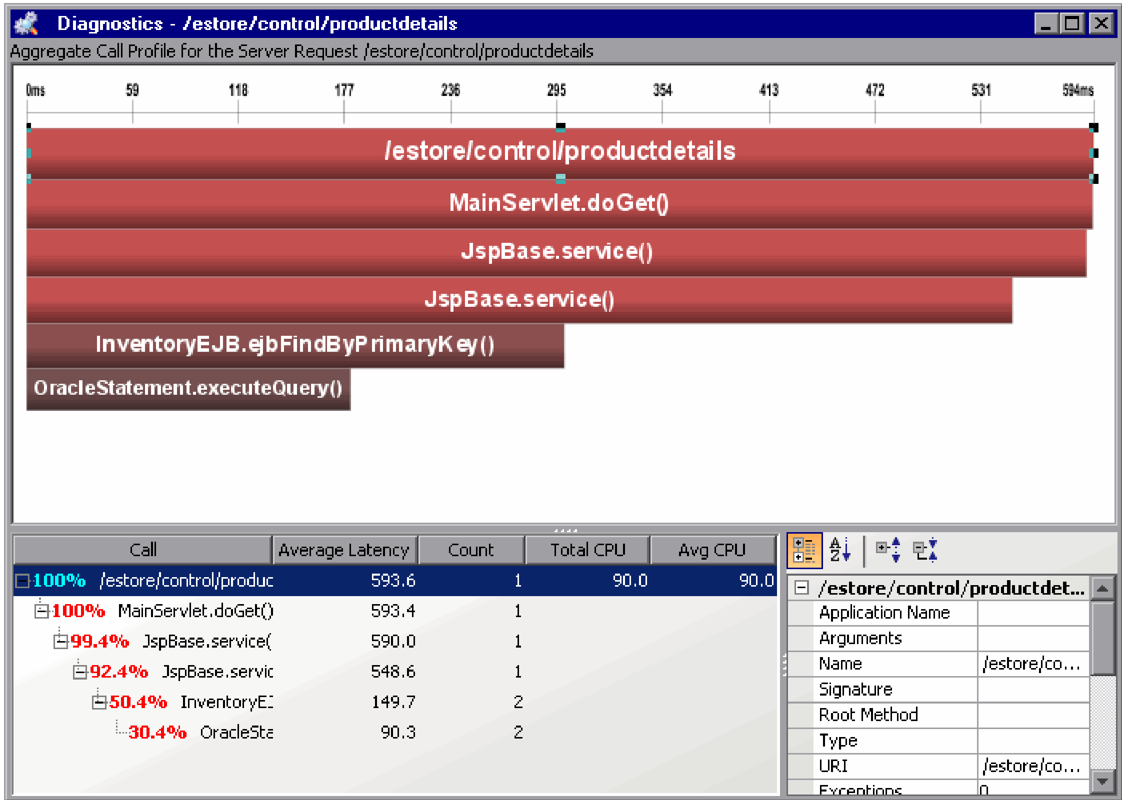
There are two types of call profile windows that are displayed depending on the how you navigated to the tab:

- **The Instance Call Profile window** displays the method calls that were made during the processing of the server request on which you drilled down.
- The **Aggregate Profile window** displays an aggregation of all of the method calls that were made during the processing of all of the server requests that were the same as the one on which you drilled down.

The Call Profile Window is made up of three areas:

- Call Profile Graph
- Call Tree Table
- Details Pane

An example of the Call Profile view showing all three of these areas:



When you click a call box in the Call Profile graph, the corresponding row is selected in the Call Tree table and the metrics for the selected call are displayed in the Details pane. When you click a row in the Call Tree table the corresponding call box in the Call Profile graph is selected and the metrics for the selected call are displayed in the Details pane.

> **Note:** There are differences in the layout and the metrics that are displayed in the Call Profile Window depending on the type of call profile that Diagnostics is displaying. These differences will be noted as each of the areas of the window are described.

Call Profile Graph

The horizontal axis of the Call Profile represents elapsed time, where time progresses from left to right.

For aggregated call profiles, the scale across the top of the profile denotes the total time.

For instance call profiles, the calls are distributed across the horizontal axis based upon the actual time when they occurred and so their positions help to show the sequence of each call relative to each other. The scale across the top of the instance call profile denotes the elapsed time since the server request was started.

The vertical axis of the call profile depicts the call stack depth or nesting level. Calls that are made at the higher levels of the call stack are shown at the top of the call profile and those made at deeper levels of the call stack are shown at the lower levels of the profile.

Each call box or node in the instance call profile represents a method call. The left edge of the box is the start time of the method call and the right edge is the return time from the call. The duration of the call is therefore represented by the length of the box. The position of the call box along the horizontal axis indicates the actual time when the call started and ended. The call boxes that appear directly beneath a call box are the child calls that are invoked by the parent call above them.

The gaps between the call boxes on a layer of the instance profile indicate one of the following processing conditions:

- The processing that took place during the gap occurred in code that is local to the parent at the previous higher level in the call profile and not in child calls in a lower layer.
- The call was waiting to acquire a lock or mutex.
- The processing that took place during the gap occurred in a child call that was not instrumented or included in a capture plan for the run.

The call boxes are colored to emphasize the different path calls.

- The calls that are part of a path through the profile that has the highest latency are colored red.
- Call path components that are not part of a critical high-latency path are colored grey.
- For a call profile showing a cross-VM call tree, each "hop" will be colored differently to help visually distinguish the calls that occurred on each tier.
- When asynchronous thread sampling is enabled you can see additional nodes added into the call profile view by sampling. These nodes are distinguished by their different (fuzzy) shading to emphasize lack of data about the represented method start and end times. The sampling nodes are transparent so you can see the instrumented methods, if any, behind the sampling nodes.
- Yellow dotted lines around a box indicates an exception was thrown.

If the duration of a call is very short or if the call appears further down in the call stack, the size of the call box can cause the name of the method that the call box represents to become too small to read. You can view the name of the method along with other details for a selected method by holding your pointer over the call box to cause the tooltip to be displayed. You can also see the details for a method selected from the call profile in the Details pane.

The call profile graph may have tabs across the top if data for exception instances and SOAP faults or payload was captured.

The tooltip contains the following details for the selected call box:

Method Detail	**Description**	**Window Type**
Method Name	Name of the method represented by the call box.	Aggregate Instance
Layer Name	The name of the Diagnostics layer where the call occurred.	Aggregate Instance
Total Contribution	The percentage contribution to the total latency of the server request that the methods processing contributed.	Aggregate Instance
Call Count	The total number of times that the method was called during the execution of the aggregated server requests instances.	Aggregate
Total Latency	The cumulative latency attributed to the processing of the method.	Aggregate Instance
Average Latency	The average latency that can be attributed to each of the method executions for the aggregated server request instances.	Aggregate

Call Tree Table

The **Call Tree** table appears directly below the **Call Profile**. This table shows the same information that is represented in the **Call Profile.**

The first row in the table contains the root of the call stack, which is the server request you selected when you requested that the Call Profile view be displayed. The rest of the rows in the tree are the method calls that were made at successive levels of depth in the call tree. You can use the expand/collapse controls in front of method calls so that you can display depth levels in the call tree as required.

In the call tree table the **X icon** indicates the cross VM outbound call. The number inside the X icon specifies the depth in the call tree. The **diamond icon** indicates the next depth level (for example 2 for second level).

Selecting an outbound call row in the table brings to the front, in the call profile graph, all boxes at the next VM depth level. Selecting any row in the table brings to the front, in the call profile graph, all boxes up to root.

When you select a row call in the table, the corresponding box is selected in the Call Profile graph, and the metrics for the selected call are displayed in the Details pane.

The Call Tree Table contains the following columns:

Column Label	Description	Window Type
Call	The name of the Server Request or Method Name. The percentage contribution of the method call to the total latency of the service request precedes the name. The percentage is colored red for those calls which are on the call tree's critical path.	Aggregate Instance
Average Latency	The average latency that can be attributed to each of the method executions for the aggregated server request instances.	Aggregate
Count	The total number of times that the method was called during the execution of the aggregated server requests instances.	Aggregate
Total Latency	The cumulative latency attributed to the processing of the method.	Instance
Total CPU	The total amount of CPU time used by the processing for the selected method or server request.	Aggregate Instance
Average CPU	The average amount of CPU time used by each of the aggregated method calls included in the selected method or server request.	Aggregate

The Total Latency for a parent call includes not only the sum of the latency of each of its children but also the latency for the processing that the method did on its own.

Call Profile Details Pane

The **Details pane** lists the metrics related to the server request or method selected in the Call Profile Graph or in the Call Tree Table.

To view the details of a particular call in the Details pane, select the call from the Call Tree Table or in the Call Profile Graph.

The metrics that are included in a metric category can be hidden or displayed by expanding or collapsing the list of metrics using the plus sign (+) and minus sign (-) next to the category name. Alternatively, you can double-click the category name to expand or collapse the list of metrics.

Thread Call Stack Trace Sampling

When asynchronous thread sampling is enabled you can see additional boxes added into the call profile graph by sampling. These boxes are distinguished by their different (fuzzy) shading to emphasize lack of data about the represented method start and end times. See "Configuration Tab Description" on page 260 for how to configure this sampling using the Java Diagnostics Profiler Configuration Tab.

See "Configuring Thread Stack Trace Sampling" on page 186 for configuration and troubleshooting information if you don't see any sampling nodes after enabling stack trace sampling.

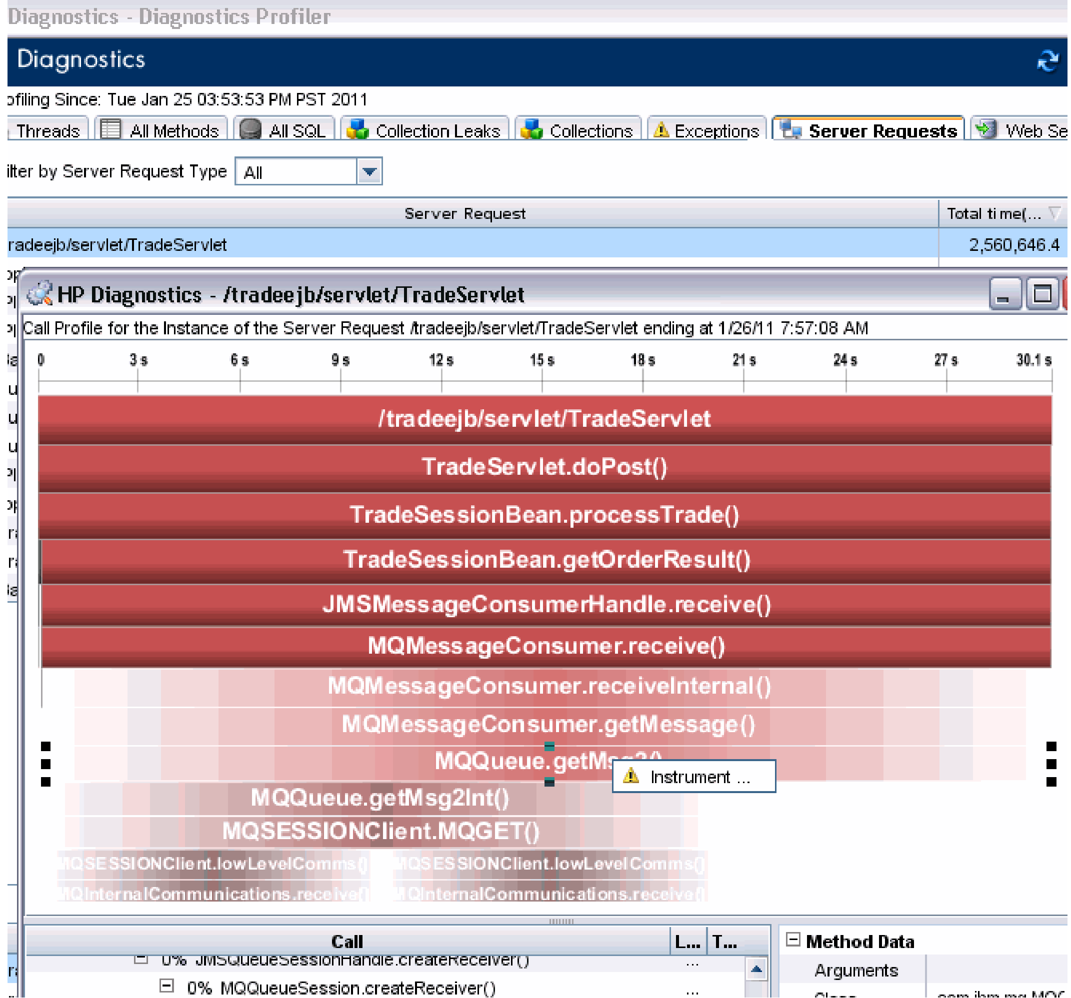
Instrumenting a Sampled Method Dynamically

Sampling methods displayed in the Call Profile when Thread Stack Trace Sampling is enabled give you an insight into the call hierarchy and latencies of these methods. But you may want to identify one of these sampling methods to actually instrument in order to get additional detail information.

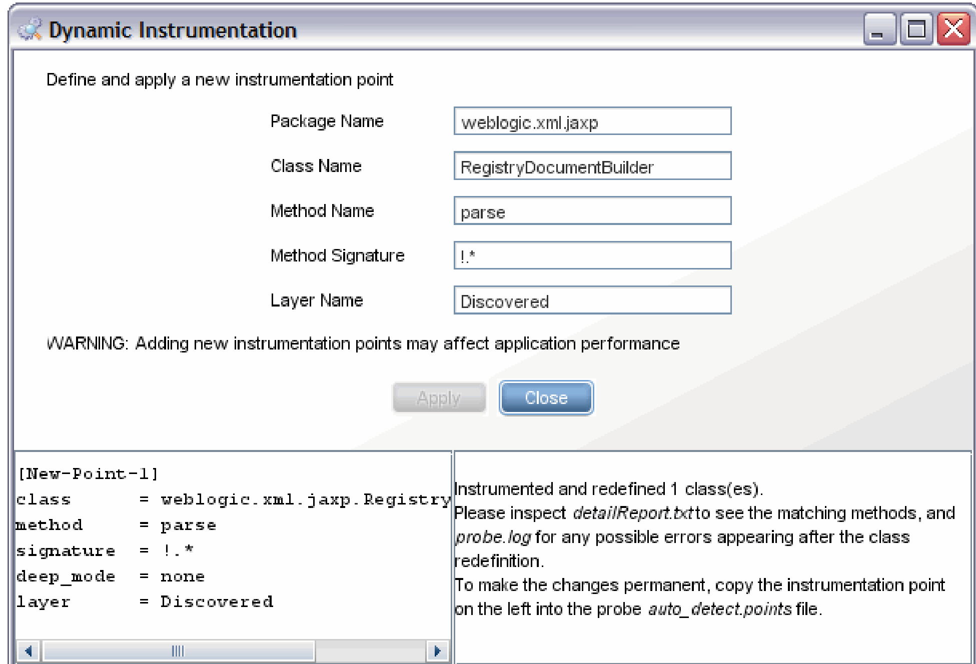
Dynamic instrumentation is Java bytecode instrumentation performed during the application execution *after* the respective class has been first loaded by the Java Virtual Machine. Instrumentation is temporary, for the current Java process. If you want to permanently instrument this method you must add the point you created to the instrumentation points file.

Note: Dynamic instrumentation (the **Instrument** menu item) is ONLY available when you access the Call Profile from the Diagnostics Profiler for Java. It is NOT available when accessing the Call Profile from an instance tree icon in the main Diagnostics UI.

From the Diagnostics Java Profiler UI, select the **Server Requests tab** and open the Call Profile window. Select the sampling (fuzzy) node in the Call Profile window and right-click to select **Instrument**.


Diagnostics - Diagnostics Profiler
Diagnostics
ofiling Since: Tue Jan 25 03:53:53 PM PST 2011
Threads
All Methods
All SQL
Collection Leaks
Collections
Exceptions
Server Requests
Web Se
ilter by Server Request Type
All
Server Request
Total time(...
radeejb/servlet/TradeServlet
2,560,646.4
HP Diagnostics - /tradeejb/servlet/TradeServlet
Call Profile for the Instance of the Server Request /tradeejb/servlet/TradeServlet ending at 1/26/11 7:57:08 AM
/tradeejb/servlet/TradeServlet
TradeServlet.doPost()
TradeSessionBean.processTrade()
TradeSessionBean.getOrderResult()
JMSMessageConsumerHandle.receive()
MQMessageConsumer.receive()
MQMessageConsumer.receiveInternal()
MQMessageConsumer.getMessage()
Instrument ...
MQQueue.getMsg2Int()
MQSESSIONClient.MQGET()
Call
Method Data
Arguments
0% MQQueueSession.createReceiver()

The Dynamic Instrumentation dialog box is displayed with values corresponding to the selected method.



You can change the package, class, and method name, and provide a method signature, if known, to narrow down the scope of the instrumentation. Since sampling does not reveal method signatures, by default all methods with matching names will be instrumented.

Note: The classes belonging to the Diagnostics Java probe or the Java runtime cannot be instrumented.

Click **Apply** after making the changes you want and the Java probe automatically creates a new point definition and tries to apply the instrumentation dynamically. The bottom part of the dialog window contains the result of this operation: the new instrumentation point definition is placed on the left side, while the result of instrumentation is located on the right.

Once the instrumentation is successful, you should copy and paste the instrumentation point to save it because when you refresh the Call Profile, the Dynamic Instrumentation window with the details on the instrumentation point you created is no longer available.

When you refresh the Call Profile view, the dynamically instrumented method will be displayed as a solid node because it is now instrumented. Instrumentation is temporary for the current Java process.

If you want to permanently instrument this method you must add the point you created to the instrumentation points file.

Comparison of Collection Leak Pinpointing and LWMD

Collection Leak Pinpointing (CLP) allows you to pinpoint Java collection related memory leak locations in Java applications. Enabling the feature in the probe is optional. Once enabled, the probe will automatically detect and report the leaking Java collection objects and their leak locations (stack traces), without any user interaction. CLP captures the stack trace when a collection is marked as a leak for the first time. The data collection for this feature has very low overhead and so it can be used in a production environment. See

"Custom Instrumentation for Java Applications" on page 96 for more information on configuring collection leak pinpointing.

Lightweight Memory Diagnostics (LWMD) can also be used to help you locate memory leaks. Enabling LWMD in the probe is optional. User interaction is required to enable LWMD. The data collection overhead for this feature is relatively high and it is not recommended for use in a production environment.

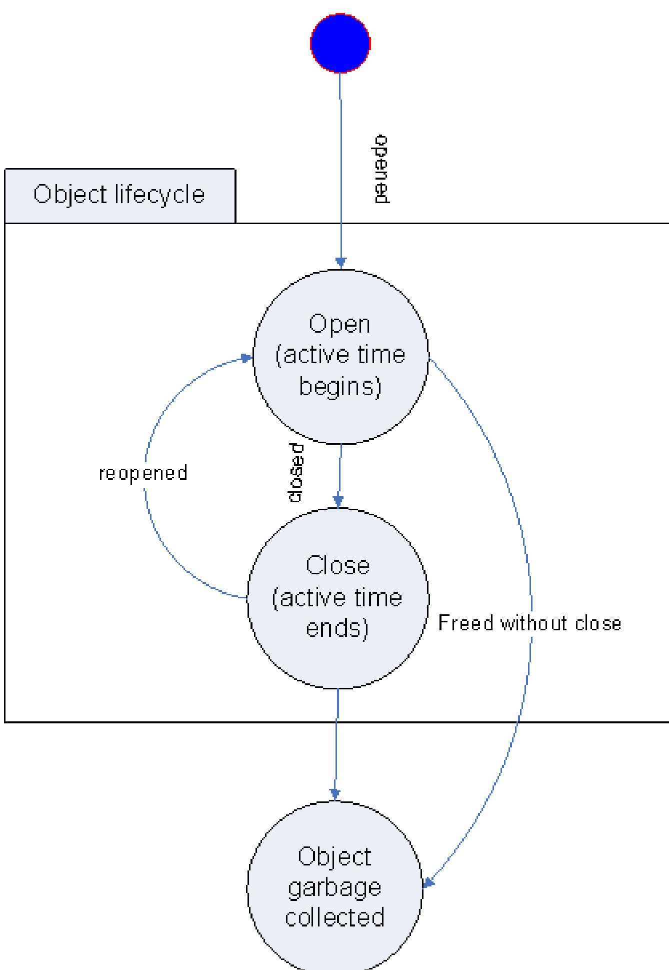
A comparison of CLP and LWMD is shown in the table below. Both are optional features and are used to help detect and locate the Java collection related memory leaks.

	CLP	LWMD
User interaction	Does not need user interaction at all. The probe will automatically detect and report the leaking Java collection objects and their leak locations.	Needs user interaction and manual steps.
Data collection overhead	Very low overhead, can be used in production environment.	Relatively high overhead, depends on the user specified scope. Not recommended to use in production environment.
Out-of-the-box status	Enabled by default	Disabled by default.
Instrumentation approach	To use this feature, you need to run the JRE instrumenter to pre-instrument the Java collection classes in the JRE jar file, and add the instrumented JRE classpath to the -Xbootclasspath/p java option to run the probe.	Once the feature is enabled, the application classes within the specified scope will be instrumented at runtime.
Instrumented classes	The Java collection classes (in java.util package and subpackages) in the JRE jar file.	The application classes within the specified scope that have Java collection object allocation.
Common data collected	Both collect full classname of the collection and size of the collection.	
Differences in data collected	Leak location: stack trace when called to add new elements to the leaking collection	The collection object allocation site: class, method, line number.

Object Lifecycle Monitoring

Every object has a lifespan. The lifespan begins with object construction and ends with its garbage collection. You can use the Allocation/LifeCycle Analysis tab in the Java Diagnostics Profiler to monitor and analyze object lifespan (see "How to Analyze Object Allocation" on page 230).

However, some objects follow a **lifecycle** during their lifespan. For example, the objects representing database resources (like database connection or cursors) go through such a lifecycle during their lifespan. See the diagram below:



These objects are brought into an *open* state by some resource acquisition operation and then *closed* after their usage. They usually acquire their resources before entering an open state and relinquish their resources after reaching a close state. Some of these objects are designed for re-use (for example, objects based on connection pool). So these objects might be re-opened and closed multiple times during their lifespan.

You can enable object lifecycle monitoring in Diagnostics and view lifecycle information for these objects in the Allocation/Lifecycle Analysis tab.

Two Examples of These Types of Objects

- **Database connection:** An object of type java.sql.Connection represents a database connection. The connection is opened by invoking javax.sql.DataSource.getConnection() method and it is closed by invoking java.sql.Connection.close() method.
- **Database cursors:** An object of type java.sql.ResultSet represents a database cursor. The cursor is opened by invoking java.sql.Statement.executeQuery() method and is closed by invoking java.sql.ResultSet.close() method.

Types of Performance Problems with These Objects

Diagnostics allows you to monitor the object's lifecycle between its open and close states to identify the following types of performance problems.

- **Resources are not released:** This problem arises when the object is not brought into a close state. This causes the resources attached to the object to be wasted for the lifetime of the object.
- **Resources are not released in a timely manner:** This problem arises when the resources are released

after unnecessarily keeping them around for quite long period of time. This can also happen if the object is not closed but the garbage collector automatically closes it during object finalization.

Viewing Object Lifecycle Information

Object lifecycle information is available in the details pane in the Profiler Allocation/Lifecycle Analysis tab for objects enabled for monitoring.

Tips for performance analysis:

The metric **Objects 'Opened' (Total)** shows the number of objects opened during the application's lifetime.

Please note that, if an object is re-opened at multiple location (a common case for pooling), the 'opened' metrics shows the number of times the object was opened. However, the location information refers to the location of the first 'opening' of the object.

Also an object is re-opened without being 'closed' then it is assumed that the object kept itself in the 'open' state. The 'opened' counter is not incremented.

The metric **Objects 'Closed' (Total)** shows the number of objects closed during the application's lifetime. If an object is re-closed without being 'opened' again, then it is assumed that the object kept itself in the 'close' state. The 'closed' counter is not incremented.

The metric **Objects Deallocated without Close** will have a value greater than zero if the resources are not properly released.

The metric **Object Active Lifespan** will have a higher average latency if the resources are not released in timely manner. Note that this metric shows the active lifespan for only those objects that have been closed.

Differences Between Object Lifecycle and Allocation Analysis

Unlike allocation analysis, the object lifecycle feature is not managed. This means that while allocation analysis can be performed by specifically selecting the **Start tracking allocations** and **Stop tracking allocations** links in the Allocation/Lifecycle Analysis tab. Object lifecycle monitoring, if enabled, will show data since the application start-up and the data will not be cleared by the **Clear allocation information** link.

Also, unlike allocation analysis, the object lifecycle feature does not support sampling. This means that all the method calls are captured for the object's lifecycle monitoring.

Heap Walker Memory Analysis Execution Steps

Heap Walker is a memory analysis process accessible from the Memory Analysis tab. You can use it to troubleshoot Java lingering object problems that are difficult to debug or reproduce. Using object tagging and heap snapshots, Heap Walker enables you to inspect individual objects suspected of having "leaked," and to determine why they are kept alive in the Java heap. This feature targets testing (pre-deployment) environments. You can also use it in production environments.

The steps for using the Heap Walker are described below. The Heap Walker also contains a wizard that guides you through the process of diagnosing a memory leak.

Step 1 - Establishing a Baseline

A typical large Java application allocates many objects during its initialization and warm-up. Classes are loaded, thread and database connection pools are populated, and numerous caches in all components are filled. These objects typically stay alive throughout the application execution. To avoid identifying these objects as potential leaks (that is, to avoid false positives), you should let the application run under load for some time to arrive at a stable state.

The application can be placed under memory leak test after initialization has completed, and object allocation has stabilized. Clicking **Start Tracking New Objects** initiates the test operation. After that, any objects allocated by the operation will be tracked as potential leaks.

The assumption here is that the deployed Java application, if allowed to fully initialize, allocates only temporary objects for all of its operations. All temporary objects should eventually be garbage collected. While most server applications comply with this design principle, there are known exceptions to this rule. Database connections, or threads in dynamically sized thread pools, can be created at any time during the application execution without time constraints on when they should be terminated.

The Heap Walker operations may also leave a footprint on the heap. (For example, some probe classes are loaded and initialized only when you start using Heap Walker.) Footprints should not be a problem if you are aware of them. However, if you need a clear picture, it is recommended that you perform the execution of all Heap Walker steps twice, treating the first pass as a warm-up only. You can ignore results from the first pass.

Step 2 - Exercising the Operation

The details of this step may differ, depending on whether the application is running in a production environment or in a test environment. In a test environment, the application owner can carefully stage the test load to contain only the desired operations. For example, testing can focus on newly developed code, or objects that are suspected of leaking memory based on the analysis of the logs or feedback from the IT center where the application is deployed.

It is often useful to use such an operation under test in some kind of a context. For example, if the application requires a user logon, it might be practical to wrap the tested operation by a logon and logout. It is typical for the application to hold the active session information in the heap. In this case, you can dismiss the session information only after a logout. Alternatively, you can perform the logon before new object tracking is started. In any case, you should arrange the tested operation in such a way that it leaves no permanent footprint in Java memory (adding records to a database is fine). The tested operation can be repeated several times. In the case of simple leaks, a single execution is usually enough. In a production environment, it is impossible to control the load, or to time the new object tracking by starting and stopping to catch only the desired portion of the load. You need to take this into account when analyzing the results.

Heap Walker can display the number and size of the currently tracked objects. These numbers are updated by taking a heap snapshot. You observe the numbers as they change over time. Measurements increase as the application allocates new objects. They decrease as the objects are garbage collected. After the tested operation is complete (or, in the case of a production environment, sufficient time has elapsed), you can click **Stop Tracking New Objects**. At this point, the set of tracked objects is closed. It can no longer grow.

It is normal, however, for several tracked objects to still be alive at this point. They can be present in numerous caches in the application, including the components that you do not own. It is also possible that some tracked objects require finalization. The finalizers are run periodically by the JVM, typically asynchronously to the activities controlled by the application. Objects pending finalization are considered alive, even though the application may hold no references to them.

Step 3 - Flushing Application Caches

Under normal circumstances, if the application remains under load, the caches clear of all the tracked objects eventually, and the pending finalizers run eventually. The JVM also runs garbage collection periodically. This garbage collection removes the tracked object from the heap, provided they are not leaks.

You can sometimes speed up this cleaning process by forcing garbage collection. Clicking **Run Garbage Collection** makes the JVM not only run the full GC cycle, but also run the pending finalizations.

Taking heap snapshots is especially useful at this point. The observed number and the total size of tracked objects should go down over time, as the cache flushing process progresses. Ideally, these numbers should eventually reach zero, meaning that all tracked objects have been garbage collected.

However, if there is a Java memory leak in the tested operation, the numbers stabilize at some non-zero values, and no longer decrease, despite repeated garbage collections and continuous load on the application. When you decide that the tracked objects remaining on the heap should be considered a leak, it is time to capture the **object reference graph**. This action dumps all references present in the heap to a file, and starts an additional (external) process, which sorts the file. The file is used in the next steps.

Step 4 - Analyzing Potential Leaks

After you capture the object reference graph, you can retrieve the list of tracked objects. In most cases, you select just one class of objects to retrieve. This can be accomplished by double-clicking the row with the selected class. It is also possible to retrieve objects for multiple classes. Simply select multiple rows (by holding down the Ctrl key), and then right-click to select **Inspect Selected Tracked Objects**.

For efficiency of operation, there is a limit on the total number of objects that can be retrieved. You can change the limit, using the selector located on the left side of the window. Retrieving a large number of objects rarely makes sense, as it is costly, and it does not necessarily increase your capability to solve the leak problem.

Step 5 - Walking the Heap

You can determine why any of the retrieved objects is alive by clicking the table row describing the object. This action displays an **Object Reference Diagram**. This diagram shows the selected object with a chain of references that are keeping the object alive, and indicates which object is a heap root.

For any object already displayed, it is possible to show all objects directly referencing it by double-clicking the object.

As above, to keep a limit on the overhead, there is a limit selector on the left side of the screen controlling the maximum number of objects to be retrieved and displayed.

An example of the Object Reference Diagram:

All displayed references (links between objects) are based on the captured object reference graph. Additional information, such as object type, size, or reference names are retrieved directly from the heap. Under some circumstances, the additional information cannot be retrieved because some of the objects keeping the specific object alive can cycle over time and be garbage collected. A continuously growing java.util.Vector object is a good illustration of this point, as the underlying array is replaced over time.

The objects are color-coded according to their age. There are three distinct object ages:

- **Baseline.** Objects allocated before new object tracking was started.
- **Tracked.** Objects allocated between new object tracking start and new object tracking stop, ostensibly by the tested operation.
- **Fresh.** Objects allocated after new object tracking was stopped.

The toolbar selections in the Object Reference Diagram are similar to the toolbar in topology views, so for toolbar details see "Working with Topologies" in the Diagnostics User Guide.

You may elect to capture a new Object Reference Diagram to obtain a fresh view of the object, and repeat "Step 4 - Analyzing Potential Leaks" above.

Heap Walker Performance Characteristics

Technically, starting or stopping new object tracking, and capturing the object reference graph, uses the JVM heap tagging operations. It may require substantial execution time, which can be up to several minutes for very large heaps. The application is practically paused during this time. Do not use Heap Walker if the nature of the deployed application cannot tolerate such long pauses. If in doubt, always test Heap Walker first in a test environment.

The above steps, and in particular starting new object tracking, also make the JVM allocate extra memory generally proportional to the current heap size. This memory is allocated outside of the Java heap, but within the JVM process. You need to take special care to ensure that such memory can be allocated. Keep in mind that the JVM itself, the application code, the JIT-compiled code, and any native libraries used by the application must fit into this space as well. For 32-bit processes, there is an operating system-dependent limit on the size of the process address space (for example, 2GB for Windows on Intel x86). If almost half (or more) of the available address space is already reserved by the Java heap, the tagging operation can crash the JVM.

Heap Walker gives you the total memory usage estimate when the **Start New Object Tracking** operation is activated for the first time. The estimate is for total system memory. It is based on additional memory needed by the JVM and on memory for the object references sorting program. At this point, you have a chance to quit Heap Walker without affecting the deployed application negatively (no additional memory is allocated). Obviously, in a production environment (deployed application), it is recommended that you use Heap Walker only if the system capacity is large enough to handle the additional memory pressure. The decision whether to continue with tagging depends not only on the total amount of memory available on the system running the application, but on the impact of a possible JVM crash on the business process as well.

When using Heap Walker in a test environment, it is usually possible to scale down the load and the maximum heap size to match the system capacity. There is no direct CPU overhead on the Java application, other than actually running a Heap Walker command (indicated by the progress bar). This also includes the tracking period. That is, even though tracking start and tracking stop consume large amounts of CPU time, there is no overhead while actually tracking new objects.

However, the increased memory footprint of the JVM may cause serious sluggishness if the JVM no longer fits into main memory, and makes excessive use of the swap area. If, after having tagged the heap, you notice severe application performance degradation while none of the Heap Walker operations are running, you most likely have a swap file thrashing problem.

How to Access the Java Diagnostics Profiler

Once you have installed the Java Agent, configured a probe to collect performance data and started the application that is being monitored, you can access the Java Diagnostics Profiler from your browser and view Diagnostics data. You can also access the Java Diagnostics Profiler by drilling down from the views of the Diagnostics Enterprise user interface.

To open the Java Diagnostics Profiler directly (standalone):

1. In your browser, go to the Java Diagnostics Profiler URL: http://<probe_host>:<probeport>/profiler.

   The probes are assigned to the first available port beginning at **35000**.

   > **Note:** You can find the port that a particular probe is using in the probe's **probe.log** file located in **<agent_install_directory>/log/<probe_id>** directory. In the **probe.log** file, find the line that begins with the words **webserver listening on**, for example: `webserver listening on 0.0.0.0:35003` The port is the number after the colon, in this example 35003.

2. Type your username and password.

   You are prompted to enter a username and password. The default username is **admin**. The default password is **admin**. You may be prompted again to enter a username and password. Re-enter the same details.

   For more information about authentication and usernames and passwords when you have the full Diagnostics product, refer to the Diagnostics Server Installation and Administration Guide section on Authentication and Authorization.

To drill down to the Diagnostics Java Profiler from the main Diagnostics UI:

1. From any view in Diagnostics Enterprise UI that shows probe entities, right-click the probe in the table and select **View Profiler for <probe name>** from the menu.
2. If the Profiler fails to open when performing the drill down from the Diagnostics UI, ensure that you have set a default browser within your operating system.

How to Enable LWMD for Collections Displays

This task describes how to enable Lightweight Memory Diagnostics (LWMD) for use in analyzing memory leaks.

By default, LWMD is disabled, so the Java Agent does not impose the additional overhead on its host when you are not going to use memory diagnostics metrics. When you detect a memory leak using the Memory Analysis tab, you can enable LWMD. When you have completed your investigation, you can disable LWMD once more.

Note: LWMD must be enabled in order for you to see any data in the Collections tab of the Java Diagnostics Profiler.

To enable LWMD:

1. Turn on the LWMD capture in the **dynamic.properties** file by setting the **lwm.diagnostics.capture** property equal to **true**.

   ```
   lwm.diagnostics.capture=true
   ```

2. Activate the LWMD point in the **auto_detect.points** file by setting **active** equal to **true** and indicate the scope of the LWMD instrumentation:

   ```
   [Light-Weight Memory Diagnostics]
   keyword = lwmd
   scope = !only\.in\.this\.Class\..*,!or\.in\.this\.Class\..*
   active=true
   ```

It is very important to limit the scope of the LWMD instrumentation to a particular package to reduce overhead. The syntax for the scope starts with an exclamation point (!) to indicate that a regular expression follows.

How to Enable Allocation Capture

This task describes how to enable allocation capture for the probe.

The Allocation/Lifecycle Analysis tab cannot display allocation objects or their metrics until allocation capture has been enabled for the probe. By default, allocation capture is disabled, so the Java Agent does not impose the additional overhead on its host when you are not going to use memory diagnostics metrics. If you suspect that you may have a memory issue with the way your application manages its object allocations, you can enable allocation capture. When you have completed your investigation, you can disable the allocation capture again.

To enable allocation capture to view data in the Allocation/LifeCycle Analysis Tab:

1. In the **auto_detect.points** file located in **<agent_install_directory>\etc**, modify the default settings to match the following:

   ```
   [Allocation]
   keyword = allocation
   detail = leak
   scope = !com\.mycompany\.mycomponent\..*
   active = true
   ```

   If you want to have reflective allocation tracked, you can add the reflection attribute to the detail argument in the Allocation point.

   ```
   [Allocation]
   keyword = allocation
   detail = leak,reflection
   scope = !com\.mycompany\.mycomponent\..*
   active = true
   ```

   This instruments the **Class.newInstance**, **Constructor.newInstance**, and **Object.clone** methods. The reflection instrumentation tracks all classes that are created.

2. Restart the monitored application, so the probe restarts and can apply the updated instrumentation.

How to Enable Object Lifecycle Monitoring

This task describes how to enable the monitoring of certain types of objects.

To enable object lifecycle monitoring to monitor object lifecycle data in the Allocation/LifeCycle Analysis tab:

1. Object lifecycle monitoring in Diagnostics is not enabled by default. The resource monitoring of certain types of objects can be individually enabled in the **etc/inst.properties** file.

   For example, to enable the database cursor monitoring set mercury.enable.resourcemonitor.jdbcResultSet=true for details.conditional.properties property in the **inst.properties** file. This enables object lifecycle monitoring for all resources of this type for a single probe.

2. You will need to restart the probe after making changes to the etc/inst.properties file.

3. Due to higher overhead of caller side instrumentation and possibly large number of objects (resources) to be tracked, it is recommended that this feature is only enabled during development stage. It should be enabled in production environment with great caution and with a very limited 'scope'.

You specify the scope in the object lifecycle monitoring section in the **auto_detect.points** file.

How to Analyze Object Allocation

This task describes how to analyze the object allocations your application is performing.

After you have identified a memory problem using the Heap Breakdown tab, you can analyze the object allocations that your application is performing by examining the allocations while the suspected application functionality is being executed. The following procedure describes how to run an experiment and study the resulting application performance.

To analyze object allocations:

1. If you have not already enabled allocation capture for the probe, do so as instructed in "How to Enable Allocation Capture" on page 228.
2. Begin tracking allocations by selecting **Start Tracking Allocations** from the Common Tasks menu.

   The probe starts collecting the metrics for the objects that are being allocated and de-allocated. No collection metrics are displayed in the tab until you select the **Refresh Allocation Information** or **Stop Tracking Allocations** menu options.
3. Execute the application functions that you suspect may be causing a leak, so any objects that are allocated while performing the function can be tracked.
4. Select the **Stop Tracking Allocations** menu option to limit the tracked objects to those that were captured while the suspect application functions were being performed.

   No additional instances are tracked after you stop tracking. The instances of the objects that were already allocated continue to be tracked as they are de-allocated, so the metrics on the tab can be refreshed with accurate counts of the objects that are alive or de-allocated, as well as with accurate object lifespans.
5. Select the **Refresh Allocation Information** menu option to update the tab with the current metrics for the allocated objects.

   Each time you select this menu option, the Profiler updates the metrics for the tracked objects in the allocations analysis table with the current counts and lifespans. The trend lines for the metrics in the graph are updated to chart the data points for the metrics at the refresh time.

   You should repeat this step as your application continues to run, so you can see what happens to the allocated objects over time.
6. If you want to run your experiment again, select the **Clear Allocation Information** menu option to clear the table and graph of all of the objects and metrics currently displayed, and begin this process again from the second step.

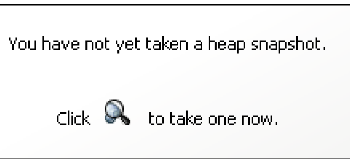
How to Enable Memory Analysis

This task describes how to enable memory analysis. By default, the Memory Analysis tab is disabled.

To enable advance memory analysis and display the Heap Walker views:

1. Use the **-agentpath:<agent_install_directory>/lib/<platform_dir>/jvmti.dll** parameter in the application startup script. Replace **jvmti.dll** with the appropriate library name if you run the probe on a non-Windows system.
2. Open the Java Diagnostics Profiler for the application, and click the **Memory Analysis** tab.

3. Click the icon to take a heap snapshot and open the first Heap Walker view.

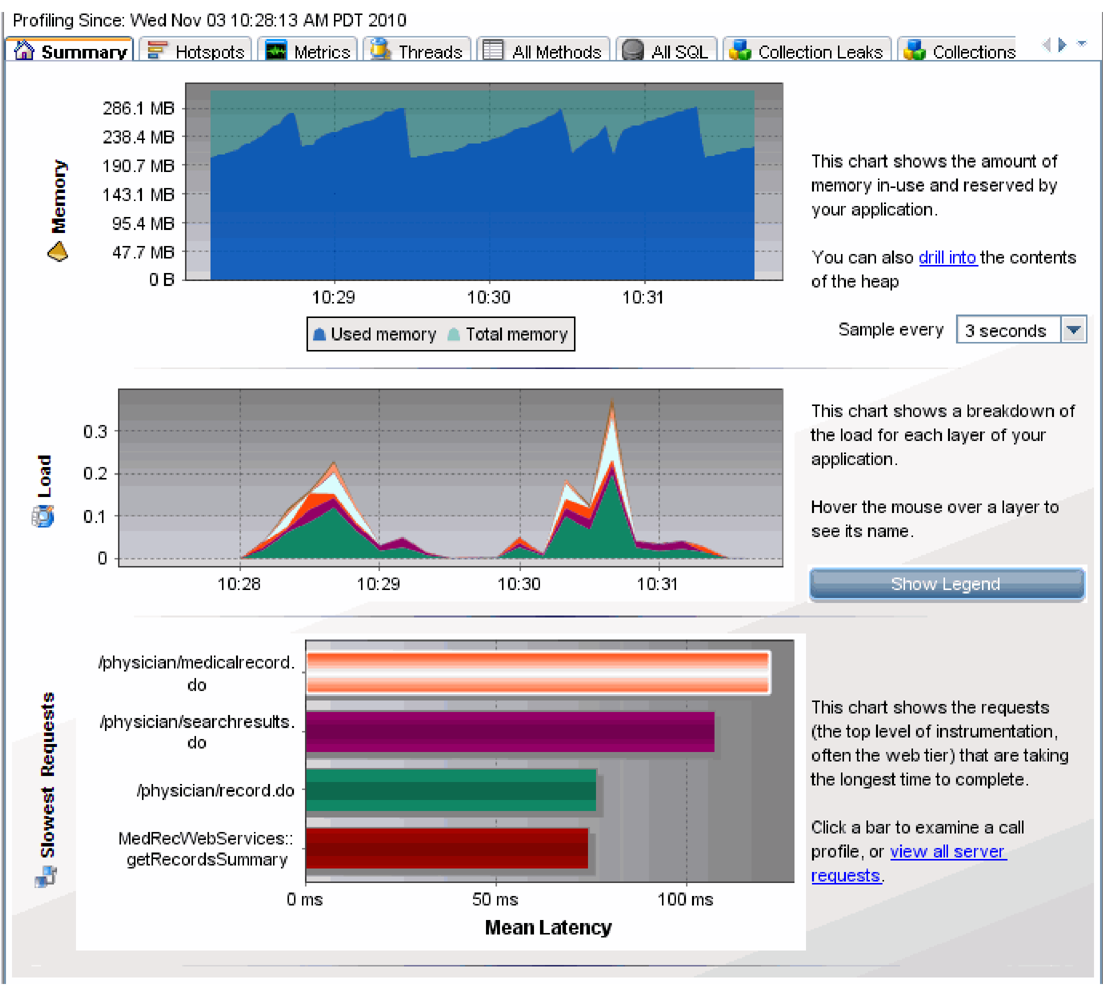
   Note: You cannot use Heap Walker when running your application with HotSpot 5.0 JVM with CMS enabled (the -XX:+UseConcMarkSweepGC option). Remove this option from the Java command if you plan to use Heap Walker.

When both the **-server** and the **-Xgc:parallel** options are selected, some versions of JRockit 5.0 JVM demonstrate instability. In some configurations, both options are selected by default. In such cases, specify the **-client** or the -**Xgc:gencon** option to override the default. This is a known BEA issue (CR334327) and should be resolved in future of releases of JRockit.

Summary Tab Description

The Summary tab consists of graphs that display information about the memory in use and reserved by your application, the load for each layer of your application and the slowest requests made to your application server.

The following is an example of the Java Profiler Summary Tab display.



To access	In the Java Diagnostics Profiler, select the Summary tab.
Important information	If the message 'Profiling not in progress' is displayed, select the Begin Profiling link in the upper left corner.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

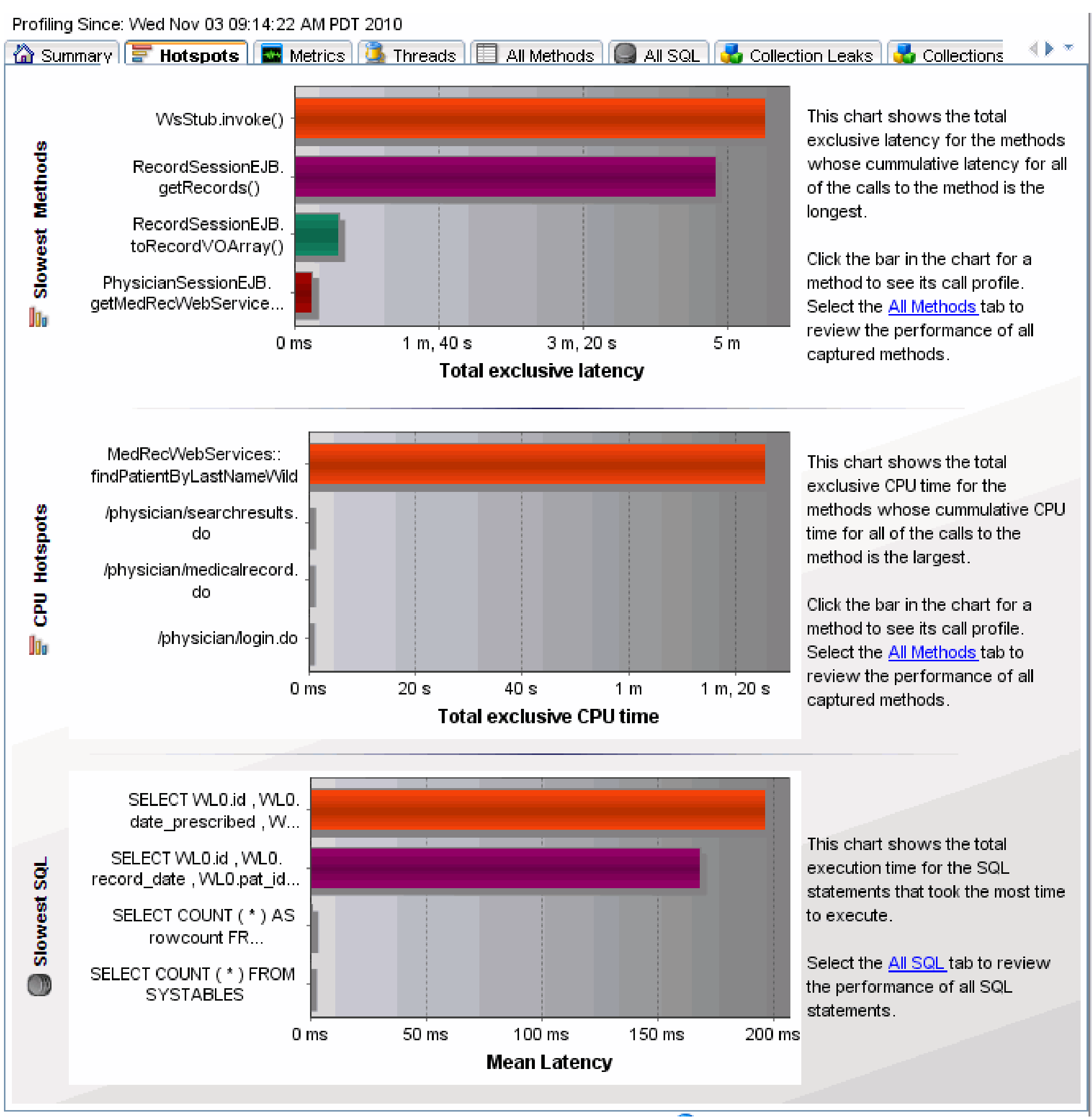
The following user interface elements are included:

UI Element	Description
Memory Graph	The Memory graph shows the amount of memory allocated in your application and the amount of memory (JVM heap size) reserved by your application. You can see more details about the exact amount of allocated memory or reserved memory in your application, by holding your pointer over various points on the graph to view the tooltip.
Load Graph	The Load graph shows the breakdown of the load for each layer of your application. The performance metrics for classes and methods are grouped into layers based upon the resources that the application invokes to perform the processing. The Java Diagnostics Profiler displays the layers on one level and does not split them into sublayers. You can see the name of each layer by holding your pointer over various points on the graph to view the tooltip. To view a legend of the graph that displays the names of all the layers, click **Show Legend**.
Slowest Requests Graph	The Slowest Request graph shows the server requests that are taking the longest time to complete. To view the aggregated call profile for a server request in the Slowest Request graph, click the bar for the server request. For more information about the call profile window, see "Analyzing Performance Using the Call Profile Window" on page 215.
Information Pane	The information pane at the bottom of the window displays the following information: • The date and time of the last time you refreshed the Profiler data. • The probe ID.

Hotspots Tab Description

The Hotspots tab displays bar charts of the significant metrics that have been captured during the monitoring of your application.

The following is an example of the Java Profiler Hotspots Tab display.



To access	In the Java Diagnostics Profiler, select the Hotspots tab.
Important information	You can view the details for a graphed metric by holding your pointer over the bar for the metric and viewing the tooltip.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

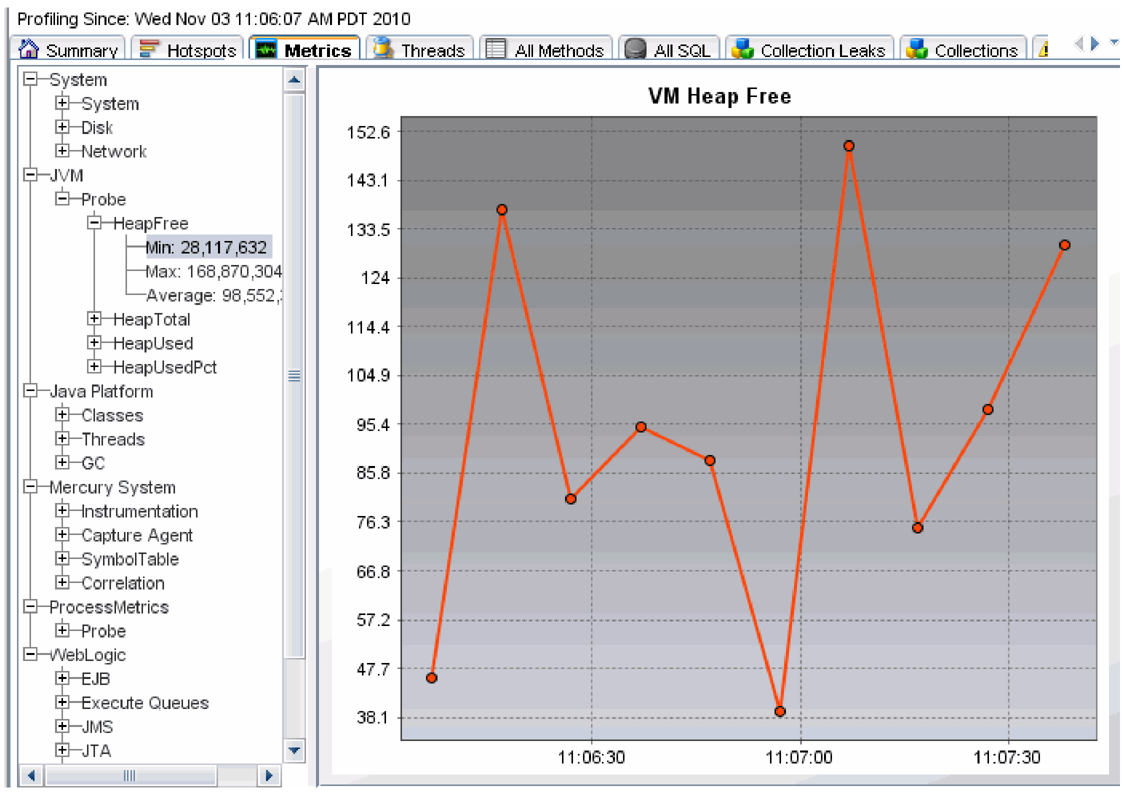
The following user interface elements are included:

UI Element	Description
Slowest Methods Graph	This chart shows the method calls that are taking the most time exclusively in that method. To view the call profile for a selected method call in the Slowest Methods graph, click the bar for the method. For more information about the call profile window, see "Analyzing Performance Using the Call Profile Window" on page 215. If the method is part of more than one server request, when you double-click the method, the following dialog box opens and asks you to select the particular server request for which you want to see the call profile. Double-click the appropriate server request row to view the call profile.
CPU Hotspots Graph	This chart shows the methods that are using the most CPU. To view the call profile for a particular method, click the bar for the method. For more information about the call profile window, see "Analyzing Performance Using the Call Profile Window" on page 215
Slowest SQL Graph	This chart shows the SQL statements that are taking the most time. To view the SQL statement details for a particular statement in the Slowest SQL graph, click the bar for the SQL statement to select it. For more information about SQL statement details, see "Analyzing Performance Using the Call Profile Window" on page 215.

Metrics Tab Description

The Metrics tab displays information about the Operating System, the JVM and the application server.

The following is an example of the Java Profiler Metrics Tab display.



To access	In the Java Diagnostics Profiler, select the Metrics tab.
Important information	When more than one probe is running on the same host, the System metrics only appear for the probe for which you opened the profiler. If you are using the Profiler without the Diagnostics product, then to preserve memory in the application server, metrics are only measured from the time you access the graph. However, if the probe is connected to a Diagnostics Server, the metrics are measured continuously, regardless of whether you have accessed the graph.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

The following user interface elements are included:

UI Element	Description
Tree Pane	Displays the metrics in an expandable tree. The top three levels displayed in the tree are: **System.** Metrics about the Operating System **JVM.** Metrics about the JVM **<application server>.** Metrics about the application server. Depending on the environment, the application servers that will be displayed are WebLogic, WebSphere, or SAP. When you expand each of the top levels, the tree displays the associated metrics for each top level. As you further expand each metric, you arrive at a minimum, a maximum and an average numerical value for each metric.
Graph Pane	Displays a graph of the metrics selected from the tree pane. When you click a specific metric in the tree, the graph pane displays a graph representing the selected metric. You can select more than one metric to display in the graph pane using the **Control** or **Shift** keys. The x-axis in the graph represents time. The y-axis in the graph represents the numerical value of the metric. Metrics are displayed for the last five minutes unless the probe is working with another Software product, in which case they are displayed for three hours.

Threads Tab Description

The Threads tab displays thread performance metrics for the Java threads that are captured by the probe and provides a way for you to capture stack traces for the captured threads. There is also a thread state analyzer that displays approximate thread state distribution percentage for each thread.

This page can be useful for helping to diagnose the following situations:

- Incorrect thread pooling or attempting to do too much in a single thread.
- Performance problems caused by deadlocks or concurrency-related issues.
- Problems that go deep into the interactions with the OS kernel where you need to see the CPU time broken into user and kernel times.

The following is an example of the Java Profiler Threads Tab display.

To access	In the Java Diagnostics Profiler, select the Threads tab.
Important information	The Threads tab is automatically disabled by Diagnostics when it detects that the JRE used to run the application has stability issues.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

The following user interface elements are included:

UI Element	Description
Controls	Used to control how often the thread metrics are updated, maximum stack trace depth for each thread, and what kind of data is displayed for the thread processing in your application. When the Threads tab is updated, the information displayed on the tab is refreshed with the latest thread metrics. You control how often the Profiler updates the thread metrics on the Threads tab. **Update button.** Select the Update button and the Profiler refreshes the information in the graph and the thread table and captures stack traces. **Automatically, Every (Thread Metric Update Frequency).** Check this box to turn automatic updates on. Select the update interval from the spinner. The Profiler immediately begins refreshing the thread metrics displayed in this tab based on the update interval specified. Whenever the Profiler updates the Threads tab display, stack traces are captured for each of the threads listed in the thread table. You can control how many stack traces for each thread are displayed in the stack trace history. **History Length.** Select the number of samples to keep and display. **Stack Trace Depth.** Select the maximum stack trace depth collected for each sample for each thread. **Export to PDF.** You can export data in the Threads tab using the PDF icon on the Profiler toolbar in the right corner near the Help link.

UI Element	Description
Chart Tab	Charts the metric for the selected threads. You may chart the metrics for one or more of the threads listed in the threads table and you can select the metric that is to be charted for each thread. Select a thread in the thread table to have it's metric graphed in the chart. Diagnostics removes the metrics for any previously charted threads from the graph and charts the metric for the selected thread. The graph legend is updated to indicate the color with which the selected thread's metrics were charted. To chart additional threads in the graph along with any that you have already charted, select additional threads in the thread table. To select each additional thread one at a time, select each row in the thread table using **Ctrl-Click.** To select a range of threads, select the row in the thread table using **Shift-Click.** Diagnostics charts the metrics for the selected thread along with the metrics for all of the threads in the thread table that are between the selected threads and the newly selected thread. The graph legend is updated to indicate the colors with which the selected threads metrics were charted. To remove the metrics from the chart for selected threads, use **Ctrl-Click** to select the row in the thread table that contains the thread whose metrics you'd like to remove from the chart. **Chart difference in.** To select a metric to be charted for each thread, select the metric from the drop down menu. Diagnostics updates the graph to chart the indicated metric for each of the threads selected in the thread table.
Thread Table	The table shown below the chart lists the metrics for each thread. The following columns are displayed: **Thread Name.** The name of the captured thread. **Thread State.** The state of the thread at the last thread metric update interval. **Kernel Time (ms).** The portion of the CPU time during which the thread was executing in kernel mode. **User Time (ms).** The portion of the CPU time during which the thread was executing in user mode. The following data comes from the JVM: Lock Name, Lock Owner Name, Lock Owner Id. The table can also include columns for **Waited Time** and **Blocked Time** metrics if you enable them. To enable these metrics, set the **threads.contention.monitoring.enabled** property to `true` in the **<agent_install_directory>/etc/probe.properties** file. This setting may cause instability for some older JVMs.

<table>
<tr><th>UI Element</th><th>Description</th></tr>
<tr><td>Stack Traces Tab</td><td>Stack traces for the threads selected in the threads table are displayed when you have indicated that you want thread stack traces captured.

The Stack Traces tab display is divided into two areas:

Captured Stack Traces. List contains a list of the times when stack trace captures occurred.

Stack Trace Details. Displays the stack traces that you indicated based on your selections from the stack trace capture list, the scope selection drop down, and the thread table.

The Stack Trace Details for drop down allows you to control which thread's stack traces the Profiler displays in the Stack Trace details area.

When you select All Threads, the stack traces for all threads are displayed in the stack trace details area. The selections made in the threads table do not impact the stack traces that are displayed in the stack trace details area when All Threads is selected.

When you select Selected Threads, the stack traces displayed in the stack trace details area are limited to those for the threads that you select in the threads table in the Chart tab.</td></tr>
</table>

UI Element	Description
State Analyzer	The State Analyzer displays approximate thread state distribution percentage for each thread, over the specified time period. Each thread is represented by a single row. The left panel provides the thread name. The center panel provides the thread state data. The total height of the colored bar represents 100%. If a thread has been in more than one state during the observation period, multiple colors are used to display the corresponding states, proportional to the time spent in those states. For automatic updates, the observation period is the same as the configured refresh period. The right panel displays the current method name, with line number, if available. If the stack traces collected for the thread over the observation period are all the same, the method name is displayed using a bold font. If different stack traces were observed, the displayed method is the topmost common method for the collected stack traces, and its display uses a regular font. If no such common method could be found, nothing is displayed. The following thread states are presented by the Thread State Analyzer: **Deadlocked.** The thread participates in a deadlock cycle. **Blocked.** The thread is delayed (suspended) when trying to enter a Java monitor. This can happen when the thread tries to invoke a synchronized method, enter a synchronized block, or re-enter the Java monitor after being awaken from the waiting state, while another thread has not left the Java monitor yet. **Running.** The thread is actively consuming CPU time. **I/O.** The thread is performing an I/O operation. It does not use any CPU time. The notion of I/O covers not only the traditional operations on files or sockets, but also covers any multimedia or graphics operations. In general, the thread is waiting for an external (out-of-process) event. **Sleeping.** The thread is delayed after invoking the Thread.sleep() method. **Waiting.** The thread is delayed, usually having executed Object.wait(). However, threads can get into this state by other means. In general, the thread is waiting for an internal (in-process) event. **Starving.** The thread is runnable, it is not suspended by any I/O, wait(), sleep() or Java monitor operation, but is not running. This can be caused by insufficient number of CPUs available, Garbage Collection pauses, excessive paging, or by a virtual machine guest OS experiencing a shortage of resources. **Unknown.** The Diagnostics Agent was unable to determine the state of the thread. The threads that do not run Java code at all (GC, JIT) will always be in this state. If your application uses native (JNI) methods for some of the I/O operations, you should add them to the **known.native.methods.io** property in **probe.properties** so the Thread State Analyzer can correctly assign the I/O state to them. Otherwise the time spent in such methods will be identified as starvation.

All Methods Tab Description

The All Methods tab lists the method calls that your application makes according to the instrumentation in the **auto_detect points** file.

The following is an example of the Java Profiler All Methods Tab display.

To access	In the Java Diagnostics Profiler, select the All Methods tab.
Important information	All CPU times shown are exclusive (not including time spent in profiled children). All of the metrics in the All Methods tab are counted from the time you enter the system or click the **Reset** button in the toolbar of the profiler.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

The following user interface elements are included:

UI Element	Description
Grouping	The Group Method calls by drop down menu allows you to view methods in the table grouped by their package (as in the example), layer or outbound call type. Or no grouping at all.
Filtering	The quick filter box has many options for filtering the table contents, for example on Method Name.

UI Element	Description
Table	The table displays information about the methods. The table is highly customizable. Right-click any column to show or hide columns and auto-resize the columns. You can also drag and drop columns to display them in a different order. The All Methods tab displays a table that contains the following columns, displayed by default: **Method name.** The names of the methods that were called. The Method name has the following syntax: **<package name>.<class name>.<method name>**. **Total Time.** The aggregate latency for all of the calls to the method. The total latency is shown in milliseconds. **Avg Time.** The average latency for all of the calls to the method. The average latency is shown in milliseconds. **Count.** The number of times that the method was invoked. **Exceptions.** The number of times that the method generated an exception. **Total CPU.** The total amount of CPU time that all invocations of the listed method used. **Avg CPU.** The CPU time that the method used during an average invocation. If CPU time metrics are not being displayed, CPU Timestamp collection for methods can be configured. See "Configuring Collection of CPU Time Metrics" on page 174. **Layer.** The Layer associated with this method according to the instrumentation in the **auto_detect points** file. The layers are displayed on one level and there is no distinction made between layers and sub-layers. To view the call profile for a method call, double-click the appropriate row. For more information about the call profile see "Analyzing Performance Using the Call Profile Window" on page 215. If the method is part of more than one server request, when you double-click the method, a dialog box opens for you to select the relevant server request. To create call profiles from more than one server requests select the first server request with a single click and select subsequent server request using control click. When you have finished making your selections, click **OK** to instruct the Profiler to create the call profiles. The call profile for each selected server request is displayed in a separate window.

All SQL Tab Description

The All SQL tab displays the SQL statements in a table.

The following is an example of the Java Profiler All SQL Tab display.

To access	In the Java Diagnostics Profiler, select the All SQL tab.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

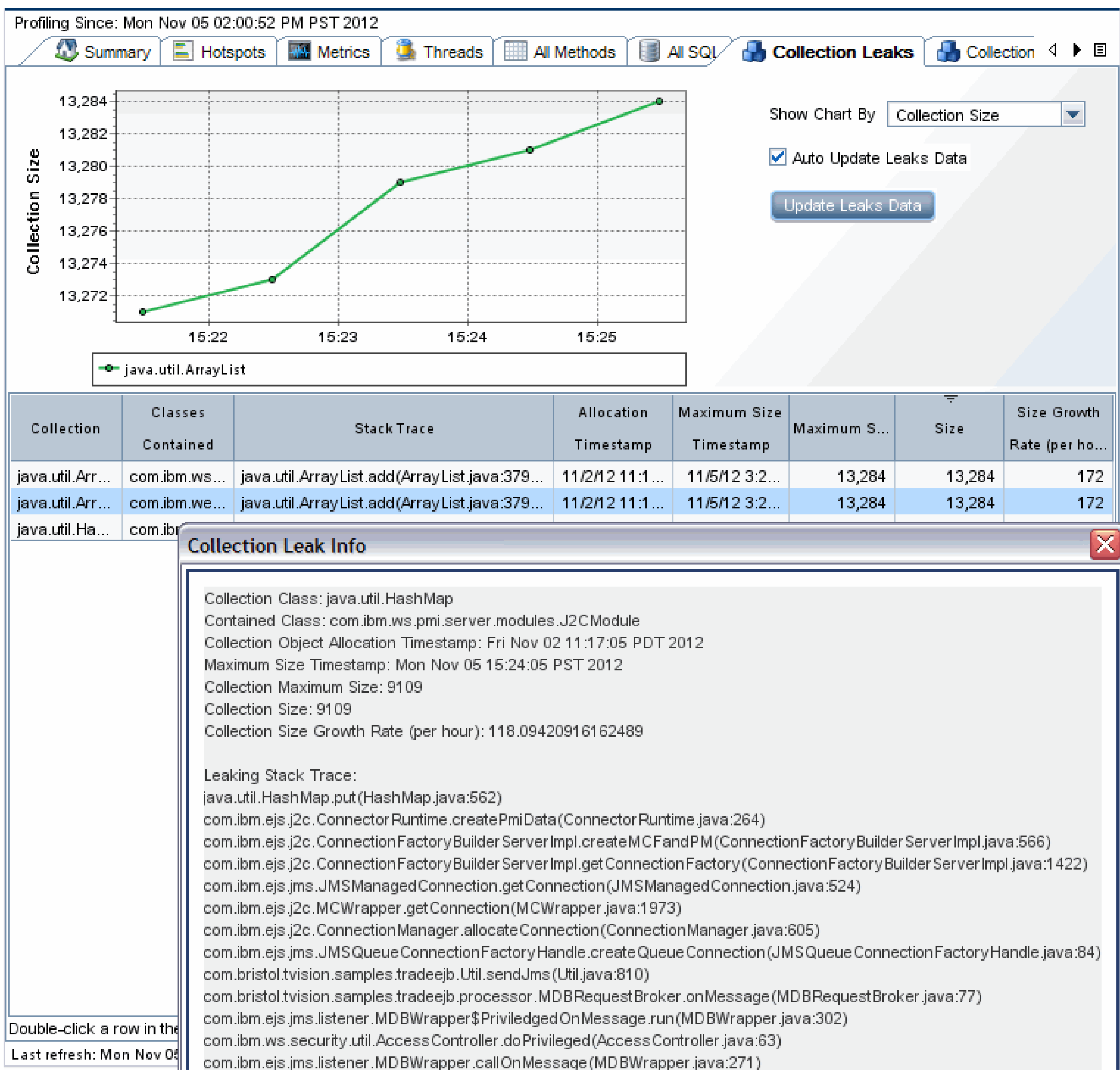
The following user interface elements are included:

UI Element	Description
Table	The All SQL tab displays the SQL Statement table, which contains the following columns. **SQL.** The name of the SQL statement that was invoked by the application server. **Total Time.** The total latency of all invocations of the SQL statement. **Avg Time.** The average latency of all invocations of the SQL statement. **Count.** The number of times the SQL statement was invoked by the application server. **Exceptions.** The number of times that the statement generated an exception. To view the SQL statement details, double-click the relevant statement. The SQL statement details dialog box opens, displaying all the information shown in the SQL table for each statement. The SQL statement details dialog box enables you to view the full string of the SQL statement and to copy the text.

Collection Leaks Tab Description

The Collection Leaks tab displays information on the probe's currently leaking collection objects in a table and a chart of collection size or collection size growth.

The following is an example of the Java Profiler Collection Leaks Tab display.



To access	In the Java Diagnostics Profiler, select the Collection Leaks tab.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227
Important Information	For this feature you need to enable Collection Leak Pinpointing (CLP) instrumentation by running the JRE instrumenter to pre-instrument the Java Collection classes in the JRE your application/application server will run with; and copy the java parameter to include them in your java options.

See also	See "Custom Instrumentation for Java Applications" on page 96 and "Advanced Java Agent and Application Server Configuration" on page 158 for more information on configuring collection leak pinpointing and for how to enable/disable and configure CLP reporting.

The following user interface elements are included:

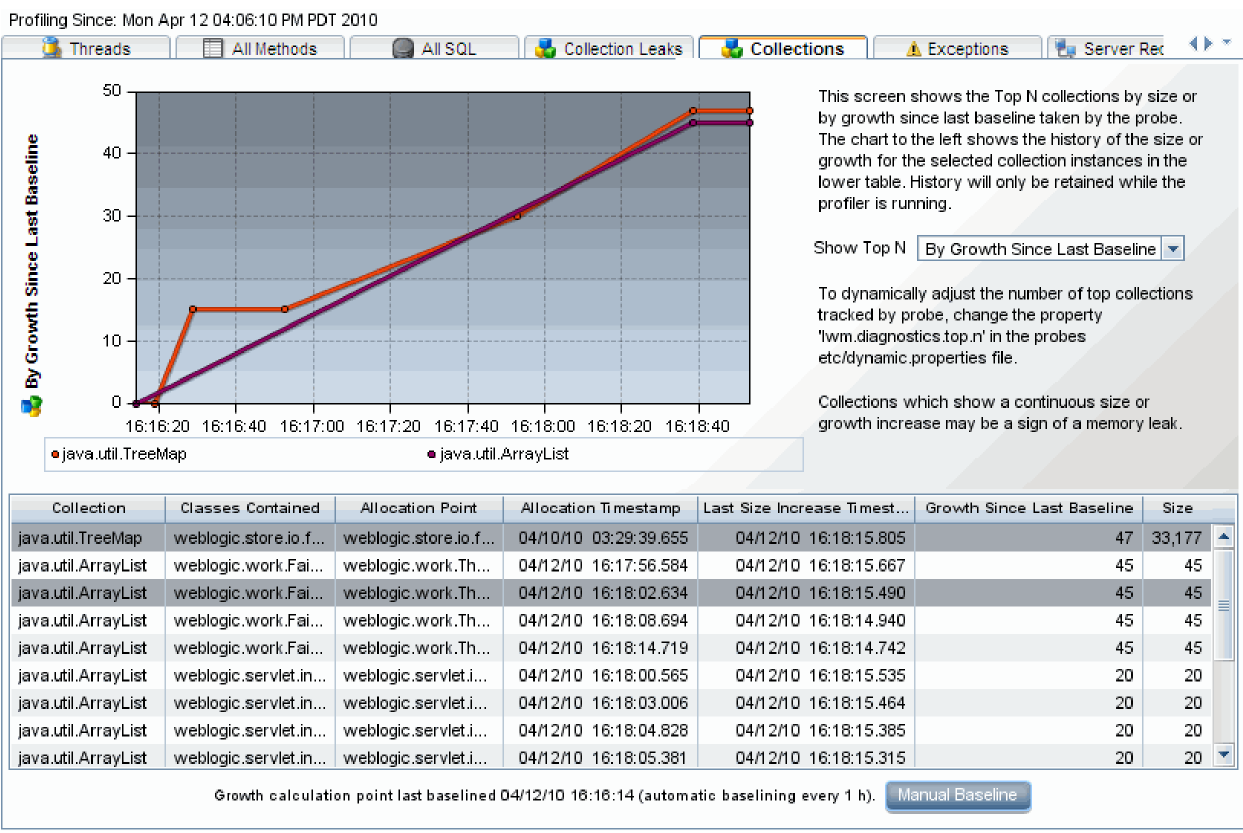
UI Element	Description
Collection Leak graph	When you click the row in the collections table, the graph is updated to show a trend line for the collection leak. The trend line shows either the Collection Size Growth, or the Collection Size, depending on the selection you make from the **Show Chart By** drop down list.
Collection Leaks Table	The collection table lists the probe's currently flagged leak collection objects. The collections can be sorted by various columns in the table. Check the Auto Update Leaks Data checkbox to automatically update the data display. Click the Update Leaks Data button to update the data. To view the collection leak details, double-click the relevant collection and a dialog box opens with the collection leak details including stack trace information. The Collections Table contains the following columns: **Collection.** The collection type. **Classes Contained.** The type of the objects contained within the collection. If there are multiple types of objects found within the collections, the value in the table appears as **Unknown**. **Stack Trace.** Leak location stack trace. **Allocation Timestamp.** The time at which the collection was allocated. **Maximum Size Timestamp.** The time when the maximum size was captured. **Maximum Size.** The maximum size of the collection ever observed by the Java agent (in number of elements). **Size.** The average size of the collection (in number of elements). **Size Growth Rate (per hour).** The average growth rate for the collection, measured over the period of time since the collection creation until now (in number of elements per hour).

Collections Tab Description

The Java Diagnostics Profiler monitors your applications' memory usage with Lightweight Memory Diagnostics (LWMD). LWMD monitors the memory used by your applications by tracking collection objects.

The Collections tab shows the metrics for the collections in your application in a graph and corresponding table. The table lists the collections, information about the allocation of the collections, and the metrics for their growth rate and size. The graph contains the metrics charted for the collections that you selected. The growth rate of the collections are calculated from a baseline. The Profiler updates the baseline periodically. If you want, you can update it manually.

The following is an example of the Java Profiler Collections Tab display.



To access	In the Java Diagnostics Profiler, select the Collections tab.
Important information	LWMD must be enabled to view data in the Collections tab and do Memory Analysis using the Heap Breakdown.
Relevant tasks	"How to Enable LWMD for Collections Displays" on page 228

The following user interface elements are included:

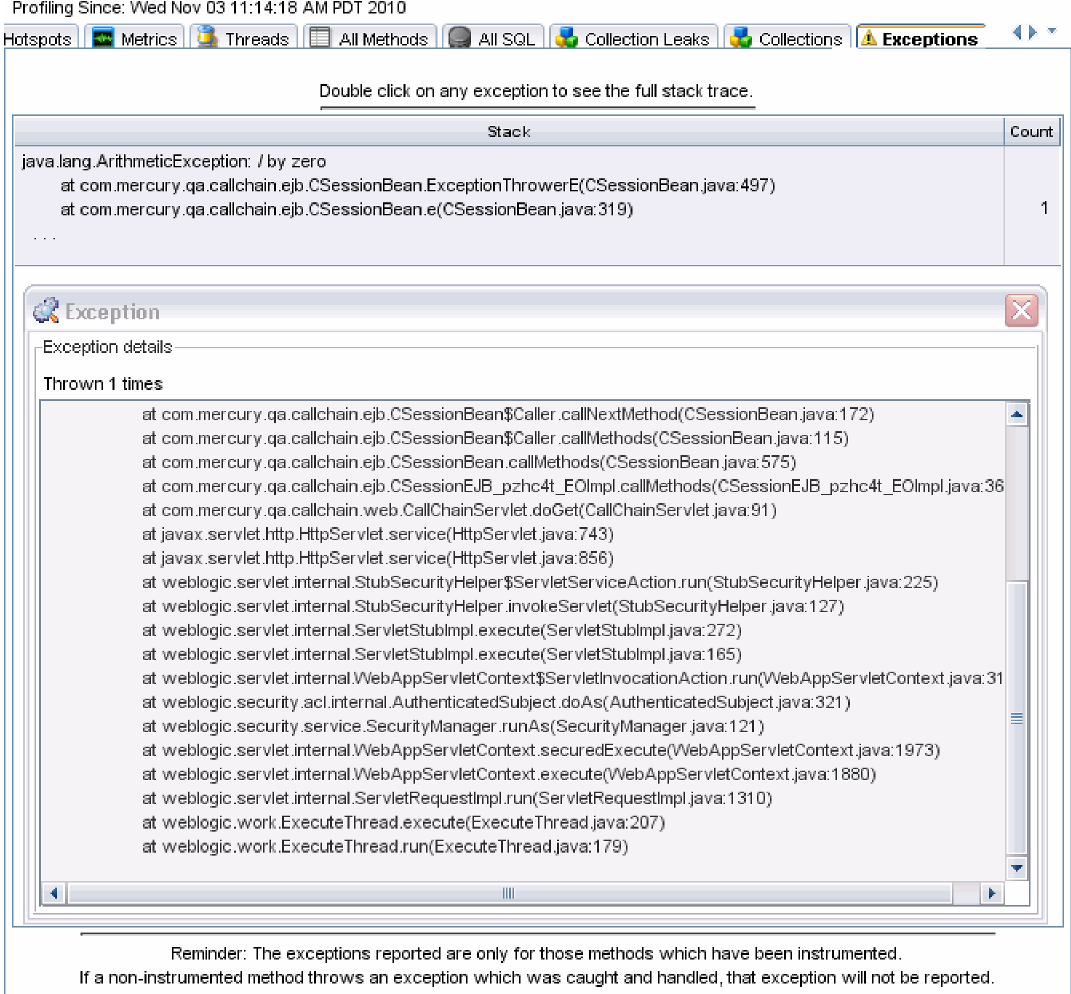
UI Element	Description
Collections Table	The collection table lists the collections. The collections are sorted by either the amount of growth since the last baseline, or by the size of the collection, depending on the selection you make from the **Show Top N** box to the right of the graph. Your selection from the **Show Top N** box controls the metrics that are charted in the collections graph as well as the sort order of the rows in the collections tables. When you choose **By Size,** the collection table is sorted in descending order by collection size. The size metrics for the selected collections are charted in the collections graph. When you chose **By Growth in Last Baseline,** the collection table is sorted in descending order by the amount of growth in the collection since the last baseline. The growth metrics for the selected collections are charted in the collections graph. The Collections Table contains the following columns: **Collection.** The collection type. **Classes Contained.** Thetype of the objects contained within the collection. If there are multiple types of objects found within the collections, the value in the table appears as **Unknown**. **Allocation Point.** The location where the collection is allocated in the code. **Allocation Timestamp.** The time at which the collection was allocated. **Last Size Increase Timestamp.** The last time that a size increase was captured. **Growth Since Last Baseline.** The increase or decrease in the number of objects within the collection since the last baseline. **Size.** The number of objects in the collection.
Collections Graph	When you click the row for a collection in the collections table, the collections graph is updated to chart either the size or the growth of the collection since the last baseline, depending on the selection you make from the **Show Top N** box to the right of the graph. You may chart the metrics for more than one of the collections by selecting subsequent rows with a **CTRL-click**.

UI Element	Description
Baseline Information	The baseline determines the time from which the growth in the size of the collections is measured. You can view the time that the last baseline was set at the bottom of the Collections display. The Profiler automatically sets a new baseline at preset periodic intervals. You can also set a new baseline manually. To set a new baseline manually, click **Manual Baseline**. The Profiler resets the Growth Since Last Baseline metric for each collection, and refreshes the charted metrics in the graph. By default, a new baseline is set automatically every hour. You can change the automatic baselining interval in the **dynamic.properties** file. You do not need to stop the application server when you change the automatic baselining interval. You can change the automatic baselining interval in the <agent_install_ directory>\etc\dynamic.properties file. Locate the line: lwm.diagnostics.auto.baseline.interval=60m. Change the time interval according to your needs as explained in the comments of the file. If you want to stop automatic baselining, enter **0** for the time interval.

Exceptions Tab Description

The Exceptions tab displays all the exceptions that were generated in the application server for methods that have been instrumented.

The following is an example of the Java Profiler Exceptions Tab display.



To access	In the Java Diagnostics Profiler, select the Exceptions tab.
Important information	If a non-instrumented method throws an exception which was caught and handled, that exception will not be reported.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

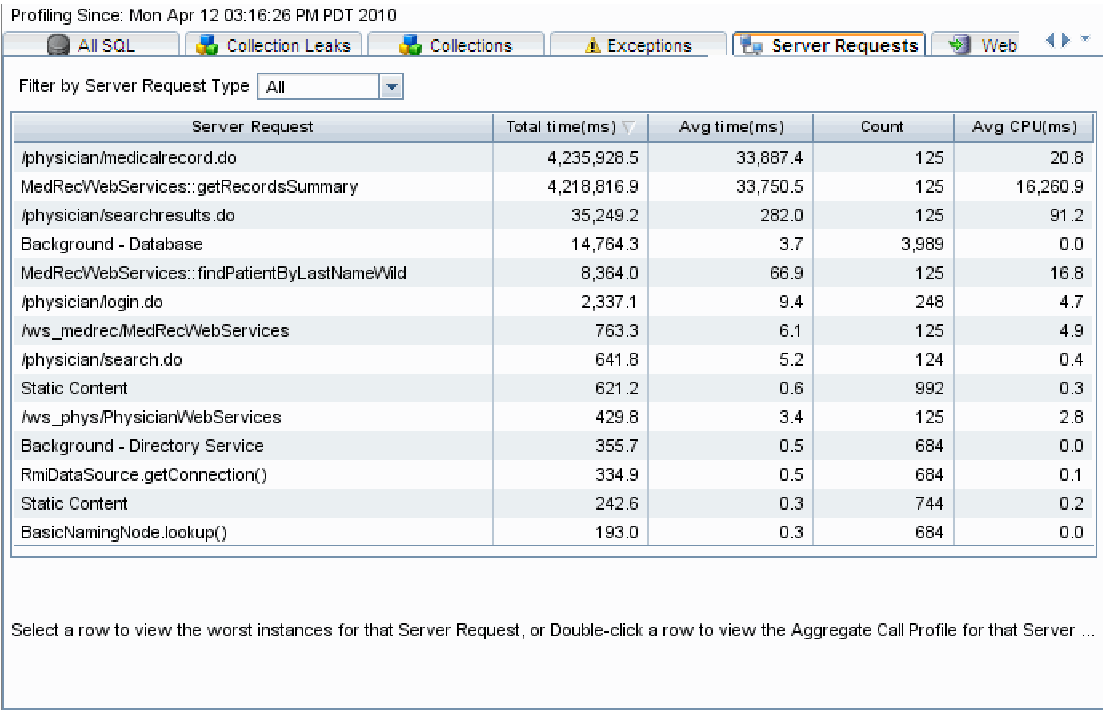
The following user interface elements are included:

UI Element	Description
Table	The Exceptions tab displays the Exceptions table which contains the following columns: **Stack.** Shows the first three lines of the exception stack trace. **Count.** The number of times the exception was generated. To see the full stack trace of the exception, double-click the row containing the exception to open the Exception dialog box.

Server Requests Tab Description

The Server Requests tab displays information about the server requests made to the application server.

The following is an example of the Java Profiler Server Requests Tab display.



Server Request	Total time(ms)	Avg time(ms)	Count	Avg CPU(ms)
/physician/medicalrecord.do	4,235,928.5	33,887.4	125	20.8
MedRecWebServices::getRecordsSummary	4,218,816.9	33,750.5	125	16,260.9
/physician/searchresults.do	35,249.2	282.0	125	91.2
Background - Database	14,764.3	3.7	3,989	0.0
MedRecWebServices::findPatientByLastNameWild	8,364.0	66.9	125	16.8
/physician/login.do	2,337.1	9.4	248	4.7
/ws_medrec/MedRecWebServices	763.3	6.1	125	4.9
/physician/search.do	641.8	5.2	124	0.4
Static Content	621.2	0.6	992	0.3
/ws_phys/PhysicianWebServices	429.8	3.4	125	2.8
Background - Directory Service	355.7	0.5	684	0.0
RmiDataSource.getConnection()	334.9	0.5	684	0.1
Static Content	242.6	0.3	744	0.2
BasicNamingNode.lookup()	193.0	0.3	684	0.0

To access	In the Java Diagnostics Profiler, select the Server Requests tab.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

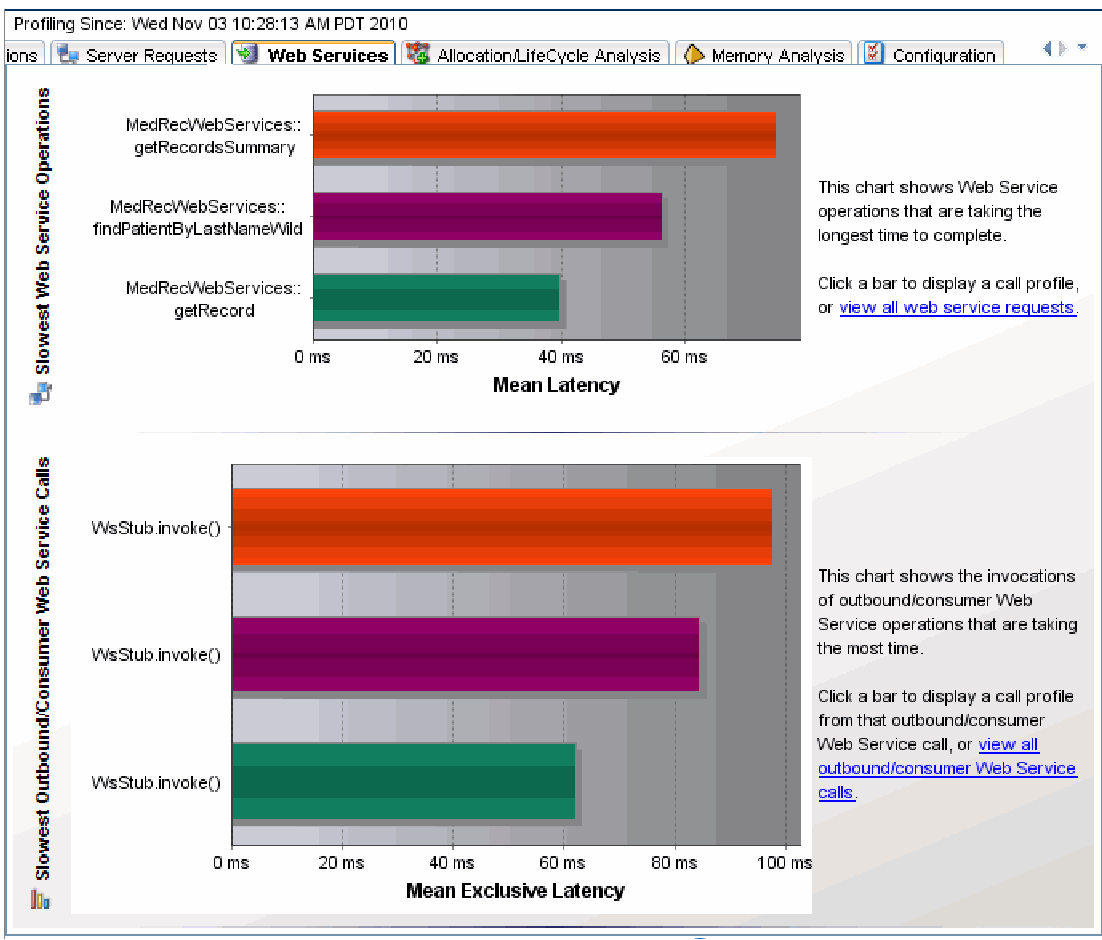
The following user interface elements are included:

UI Element	Description
Table (Server Requests)	The server request table at the top of the display lists the aggregated performance information for all instances of the server requests. When you select a server request in this table by clicking the row, a table at the bottom of the tab is populated with the three server request instances that have the worst total time. When you double-click a server request in this table, the Profiler displays the call profile for the selected aggregated server request in a new window. For more information about the call profile window, see "Analyzing Performance Using the Call Profile Window" on page 215. The aggregated Server Requests table contains the following columns: **Server Request.** The URI or the root method for the server request. The URI parameters are trimmed. To break down server requests according to URI parameters, contact support. **Total Time.** The total latency of all invocations of the server request. **Average Time.** The Average latency of all invocations of the server request. **Count.** The number of times this server request was invoked. **Avg CPU.** The CPU time that the method used during an average invocation. If CPU time metrics are not being displayed, CPU Timestamp collection for methods can be configured. See "Configuring Collection of CPU Time Metrics" on page 174 for details. **Layer.** Displays thelayer for server requests that were invoked by root methods that are not part of an HTTP request. HTTP server requests do not have a layer.
Table (Slowest Instances)	When you click a server request, the bottom section of the window displays a table containing the three slowest instances of the server request. The table contains the following columns: **Server Request.** The name of the server request. **Start Timestamp.** Point in time when the server request instance was invoked. **End Timestamp.** Point in time when the server request ended. **Total Time.** Total amount of time the server request took to execute. **Threw Exception.** Indicates whether or not an exception was thrown during the processing of this server request instance. To view the instance call profile for an instance of a server request, double-click a server request instance. For more information about the call profile see "Analyzing Performance Using the Call Profile Window" on page 215.

Web Services Tab Description

The Web Services tab contains graphs displaying the slowest Web service operations (inbound Web service calls) received and processed in your monitored environment and the slowest outbound Web service calls made from within your monitored environment.

The following is an example of the Java Profiler Web Services Tab display.



To access	In the Java Diagnostics Profiler, select the Web Services tab.
Important information	Web service operations and calls are displayed in the graphs, in the following format: **<Web-service-name>::<operation-name>.** For example, **MedRecWebServices::getRecordsSummary**.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

The following user interface elements are included:

UI Element	Description
Slowest Web Service Operations Graph	The Slowest Web Service Operations graph displays the slowest Web service operations (inbound Web service calls) received and processed in your monitored environment. The Java Diagnostics Profiler displays Web service operations as a type of server request. You can view the call profile for a Web service operation displayed in the graph, by clicking the bar representing the relevant Web service operation. For more information about the call profile window, see "Analyzing Performance Using the Call Profile Window" on page 215. You can view a list of all the Web service operations in the Server Requests tab, by clicking the **view all web service requests** link to the right of the graph. For more information about the Server Requests tab, see "Server Requests Tab Description" on page 252.
Slowest Outbound/Consumer Web Service Calls Graph	The Slowest Outbound/Consumer Web Service Calls graph displays the slowest outbound/consumer Web service calls made from within your monitored environment. The Java Diagnostics Profiler displays outbound Web service calls as remote calls within a server request. You can view the call profile for the server request containing a particular outbound Web service call displayed in the graph. To view the call profile, click the bar representing the relevant Web service call. For more information about the call profile window, see "Analyzing Performance Using the Call Profile Window" on page 215. If the remote call is part of more than one server request, when you double-click the method, a dialog box opens and asks you to select the relevant server request. Double-click the appropriate server request row to view the call profile. You can view all the outbound Web service calls in the All Methods tab, by clicking the **view all outbound Web service calls** link to the right of the graph. For more information about the All Methods tab, see "All Methods Tab Description" on page 242.

Allocation/LifeCycle Analysis Tab Description

The Allocation/Lifecycle Analysis tab shows the metrics for the objects that have been allocated by your application in a graph and a corresponding table. The table lists the allocated objects, along with the number of allocated instances and their lifespan. The graph contains the charted metrics for the selected allocated objects.

The Allocation/Lifecycle Analysis tab can be used for:

- **Allocation Analysis.** Use the information displayed to investigate a memory leak that you have observed in the Heap Breakdown tab by examining the allocation and de-allocation of objects while the leak is happening.
- **Lifecycle Analysis.** Use the information displayed to monitor object lifecycles. This feature can be used for resource monitoring of certain database resources.

To analyze allocations, you must use the controls in the Common Tasks menu to track allocations and refresh the displayed metrics as you exercise the application functionality that you believe may be experiencing leaks.

The following is an example of the Java Profiler Allocation/LifeCycle Analysis Tab display.

To access	In the Java Diagnostics Profiler, select the Allocation/LifeCycle Analysis tab.
Important information	The Allocation/Lifecycle Analysis tab is similar to the views with a detail layout in the Diagnostics views. Instead of appearing in the view title, the view filters appear in view filter menus, along the side of the graph. The Common Tasks menu controls the tracking of the allocations as well as the refreshing of the information that is displayed for the entire view.
Relevant tasks	Allocation capture must be enabled to view allocation data. See "How to Enable Allocation Capture" on page 228. Object lifecycle monitoring must be enabled to view object lifecycle data. See "How to Enable Object Lifecycle Monitoring" on page 229.

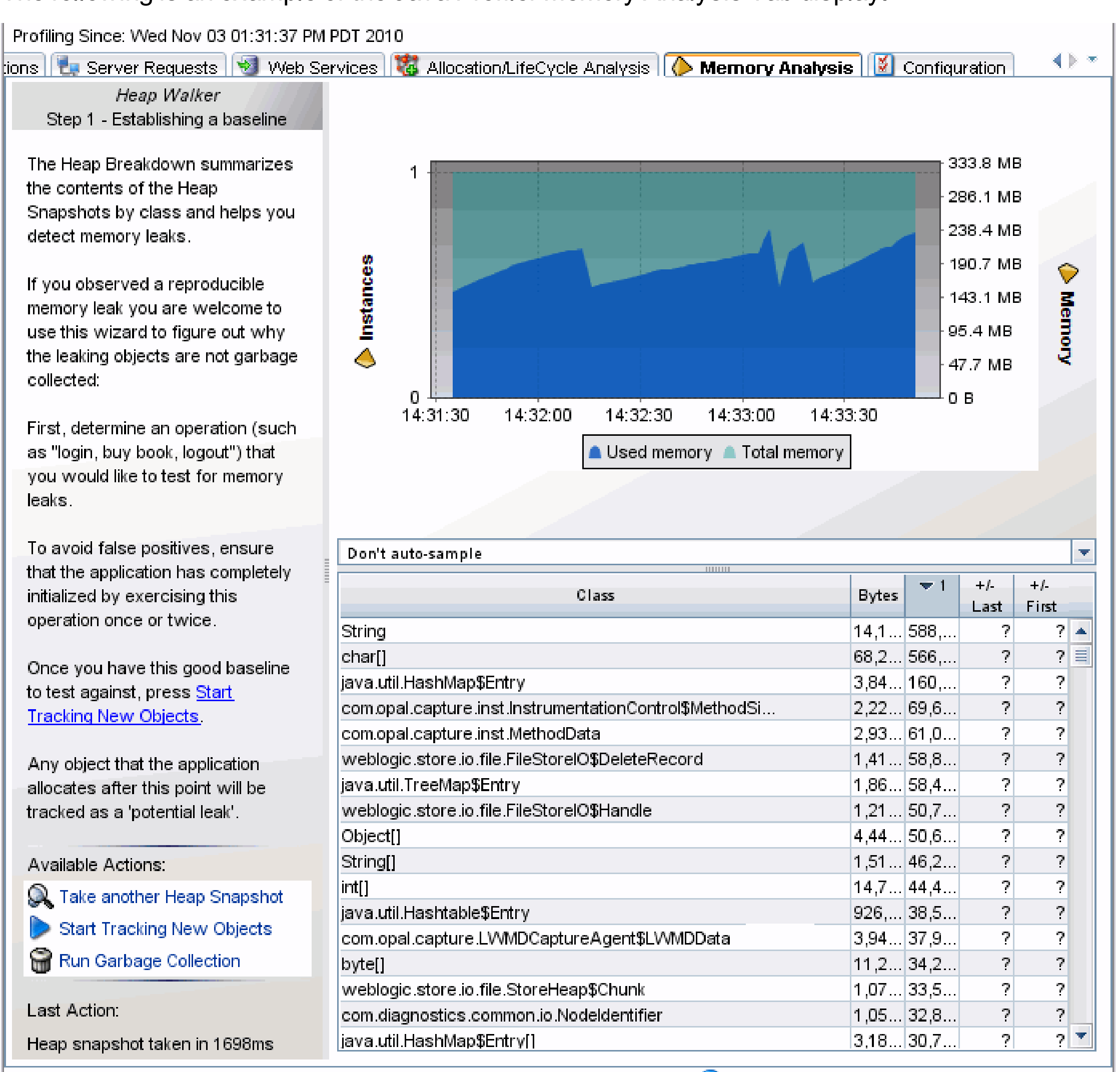
The following user interface elements are included:

UI Element	Description
Object Table	The object table lists the objects that have been allocated since you started tracking allocations. You can customize this table to adjust the sort order and the columns that appear in the table, just like the graph-entity tables in other views with the detail layout. By default, the table is sorted in order by Objects Currently Alive. It displays the following columns: **Chart.** Allows you to indicate if the metrics for the allocated object are to be charted in the graph. You can select objects to be charted by clicking on the box in this column manually. Or you can let the Profiler select the objects to chart dynamically, using the criteria that you specify in the Graph filter. **Color.** Indicates the color that the Profiler uses to chart the metrics for the allocated object. No color is shown for metrics that are not charted. **Objects Currently Alive.** A count of the total number of allocated objects that have not yet been garbage collected. **Objects Allocated.** A count of the total number of objects that have been allocated whether they have been garbage collected or not. **Objects Deallocated.** A count of the total number of objects that have been garbage collected. **Object Lifespan.** The average duration of the life of all de-allocated objects. If no objects have been de-allocated, this column is blank. In the Details pane, metrics for **Objects Lifecycle** and **Object Active Lifespan** are also available. See "Object Lifecycle Monitoring" on page 222 for more information.
Graph	The graph charts the metrics that you selected from the details table for each of the objects selected in the allocation/lifecycle analysis table. Using the controls in the views with a detail layout, you control which metrics are charted and which entities have their metrics charted.

Memory Analysis Tab Description

Data displayed in the Memory Analysis Tab helps you to find memory leaks. The Memory Analysis Tab includes a wizard that guides you through the process of diagnosing a memory leak. See "Heap Walker Memory Analysis Execution Steps" on page 224 for details about the heap walker process you can use to diagnose memory leaks.

The following is an example of the Java Profiler Memory Analysis Tab display.



To access	In the Java Diagnostics Profiler, select the Memory Analysis tab.
Important information	Heap Walker has JVM and memory requirements as described in this topic. Also See "Heap Walker Performance Characteristics" on page 227.
Relevant tasks	By default, the Memory Analysis tab is disabled. You must enable memory analysis (see "How to Enable Memory Analysis" on page 230). See "Heap Walker Memory Analysis Execution Steps" on page 224 for a description of how to use the heap walker wizard.

The following user interface elements are included:

UI Element	Description
Heap Metrics Table	The Heap Metrics table contains the following columns: **Class.** The name of the class. **Bytes.** Actual amount of memory, in bytes, that has been allocated by objects of this class. By design the heap dump does not report classes with less than 1000 bytes of total footprint but this is configurable in **dynamic.properties** using the **heapdump.class.bytes.min** property. **Count.** The number of object instances of this class that are allocated in the JVM. **+/-Last.** The count change since the most recent time a heap snapshot was taken. **+/-First.** The count change since the initial heap snapshot was taken
Heap Breakdown Graph	When you select a class name in the Heap Breakdown table, the Heap Breakdown graph shows the count over time of objects belonging to that class. You can select more than one class to display on the graph by selecting subsequent rows with a **CTRL-click**. The graph legend will display up to three rows and then a scroll bar will be added so you can scroll to see additional items.
Heap Walker	The Memory Analysis Tab includes a wizard that guides you through the process of diagnosing a memory leak. See also "Heap Walker Memory Analysis Execution Steps" on page 224 for how to use the heap walker wizard. Heap Walker requires the following: **JVM Requirements:** Heap Walker uses the JVM Tool Interface (JVM TI). As a result, the profiled application must run on a Java VM that implements JVM TI, including the optional JVM TI capability can_tag_objects. Sun HotSpot JVM, version 5.0, for Linux and Windows on Intel x86, are examples of compatible JVMs. **Memory Requirements.** Tagging the heap, and processing the object reference graph, requires large amounts of memory (total physical memory available for the JVM, not Java heap memory). The amount of memory required depends on the size of the heap used by the application. You will see an error message if there isn't enough memory on the system based on the heap size.

Configuration Tab Description

The Configuration tab in the Java Diagnostics Profiler provides a way for you to maintain the instrumentation points and some of the probe configuration without having to manually edit the capture points file or property files.

The following is an example of the Java Profiler Configuration Tab display.

To access	In the Java Diagnostics Profiler, select the Configuration tab. You can use this page whether profiling has been started for the probe or not.
Important information	See "Custom Instrumentation for Java Applications" on page 96 and "Advanced Java Agent and Application Server Configuration" on page 158 for more information on the properties configured in this page. In VMware, the CPU time metric is from the perspective of the guest operating system and is affected by the VMware virtual timer. See the VMware whitepaper on timekeeping at http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf. and see "Time Synchronization for Probes Running in VMware in the Diagnostics Server Installation and Administration Guide.
Relevant tasks	"How to Access the Java Diagnostics Profiler" on page 227

Probe Settings

The following user interface elements are included:

UI Element	Description
General	**Enable Monitoring Data Collection.** You can enable and disable monitoring data collection by checking or unchecking this box. By unchecking this box, you can disable monitoring data collection without stopping the Java Agent. **Monitoring Profile.** You can select the monitoring profile by choosing an option from the drop-down menu. For details on monitoring profiles, see "Monitoring Profiles" on page 86. **Collect CPU Timestamps.** You can enable and disable CPU Timestamp collection by choosing an option from the drop-down menu.

UI Element	Description
Trimming	Properties that reduce the amount of data pulled from the probe. **Server Request Minimum Latency**. Only server requests that take more than this amount of time will be captured, unless a threshold has been set on that server request. **Method Minimum Latency**. Only regular methods that execute slower than this number of milliseconds will be captured. **SQL Statement Minimum Latency**. If an SQL statement takes less than this amount of time, it will not be trended, until it does exceed this time. **URI Replacement Pattern**. Specifies the URIs substitutions which will be used by the agent when reporting the HTTP server requests. Pattern applies after all other URI adjustments.
Stack Tracing	When asynchronous thread sampling is enabled, you can see, in the Call Profile view, which methods were executed during long running fragments even if no instrumented methods were hit during this time. You can enable and configure the following properties. **Thread Stack Trace Sampling.** Enables or disables asynchronous thread stack trace sampling; possible values are false, auto (the default), and true. When set to auto, stack trace sampling is enabled IF the probe is running on selected (certified) platforms and JVMs. For other JVMs, the setting must be set to Enable explicitly. Use caution because the JVM could generate errors or abort. See Diagnostics Release Notes for limitations. **Sampling Interval**. The time that must elapse before the next consecutive sampling attempt is made. Small values cause frequent sampling and provide rich data but at the cost of increased overhead. The overhead caused by frequent sampling affects primarily the latency of server requests. The overall CPU usage by the probe can go up as well, but this effect is not as profound as the latency increase. For systems with many CPUs, the process CPU consumption can actually go down (not a good thing). **Tardy Method Latency Threshold.** The minimum time an instrumented method must run without hitting any instrumentation points before stack trace sampling is attempted for this method. The purpose of this property is to control the overhead of sampling by limiting the stack trace collection to only the most interesting cases. **Maximum Stack Trace Depth.** The limit for the depth of stack traces obtained from the JVM. You will most likely not need to adjust this value. These properties can also be set in the dynamic.properties file. And additional configuration can be done in dispatcher.properties for **enable.stack.trace.aggregation**, **aggregated.stack.trace.validity.threshold**.

UI Element	Description
Collection Leaks	You must run the JRE instrumenter if you want to use the collection leaks pinpointing (CLP) feature in the Java Agent. **Report Collection Leaks.** You can enable and disable reporting by checking or unchecking this box. **Collection Leaks Flag Threshold.** The threshold of time duration in which the collection has size growth. If a collection's size growth period exceeds this threshold, it will be flagged as a memory leak by the probe. **Collection Leaks Unflag Threshold.** For an already flagged leaking collection, if its size stops growing continually for this threshold time period, that probe will unflag it as a leak. These same values can also be set in the **dynamic.properties** file for the probe: **clp.diagnostics.reporting**, **clp.diagnostics.growth.time** and **clp.diagnostics.nongrowth.time**.
Client Monitoring	**Enable Client Monitoring Instrumentation.** You can enable and disable client monitoring by checking or unchecking this box. Client monitoring is set to false by default. **Client Monitoring Sampling Percentage.** The percentage of instances for which Client Monitoring instrumentation will be in effect, if it is enabled.

Instrumentation

The following user interface elements are included:

UI Element	Description
View Currently Used Instrumentation	Click the link to view the instrumentation for the application that the probe is monitoring. The instrumentation presented is from the capture points file that Diagnostics uses to instrument your applications. See "Maintaining Instrumentation from the Java Profiler UI" on page 147 for more information.
Shared Instrumentation	Click **Edit** to modify the currently-used shared instrumentation. The instrumentation presented is from the capture points file that Diagnostics uses to instrument your applications. See "Maintaining Instrumentation from the Java Profiler UI" on page 147 for more information.
Instance Instrumentation	Click **Edit** to modify the currently-used instance instrumentation. The instrumentation presented is from the capture points file that Diagnostics uses to instrument your applications. See "Maintaining Instrumentation from the Java Profiler UI" on page 147 for more information.

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Java Agent Guide (Diagnostics 9.51)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to docs.feedback@microfocus.com.

We appreciate your feedback!