CA Wily Introscope®

Java Agent Guide

Version 8.2

Date: 10-2009

For help with Introscope or any other product from CA Wily Technology, contact Wily Technical Support at 1-888-GET-WILY ext. 1 or support@wilytech.com.

If you are the registered support contact for your company, you can access the support Web site directly at www.ca.com/wily/support.

We value your feedback

Please take this short online survey to help us improve the information we provide you. Link to the survey at: http://tinyurl.com/6j6ugb

If you have other comments or suggestions about Wily documentation, please send us an e-mail at wily-techpubs@ca.com.

6000 Shoreline Court, Suite 300
South San Francisco, CA 94080

US Toll Free 888 GET WILY ext. 1
US +1 630 505 6966
Fax +1 650 534 9340
Europe +44 (0)870 351 6752
Asia-Pacific +81 3 6868 2300
Japan Toll Free 0120 974 580
Latin America +55 11 5503 6167
www.ca.com/apm

Table of Contents

SECTION I

Java Agent Installation and ProbeBuilding

The chapters in this section describe the Java Agent implementation process.

CHAPTER 1

The Java Agent Overview

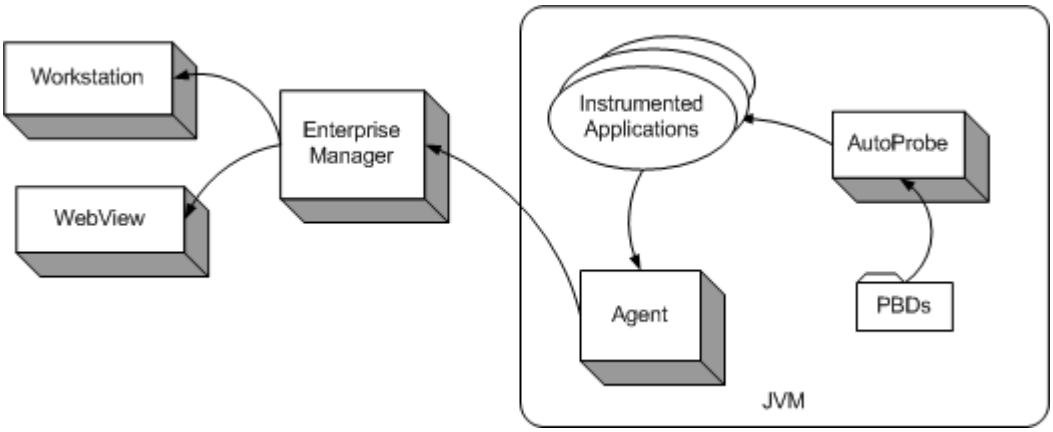
This chapter explains the Java Agent deployment process.

The Introscope environment

CA Wily Introscope is an application management tool that provides end-to-end performance management of your applications. The Java Agent is the component of Introscope that collects performance data from your applications running on Java Virtual Machines (JVMs), and sends it to the Introscope Enterprise Manager. The Enterprise Manager processes the data received from the Java Agent and sends it to the Introscope Workstation where you can review the information and set up actions and alerts based on the data received.

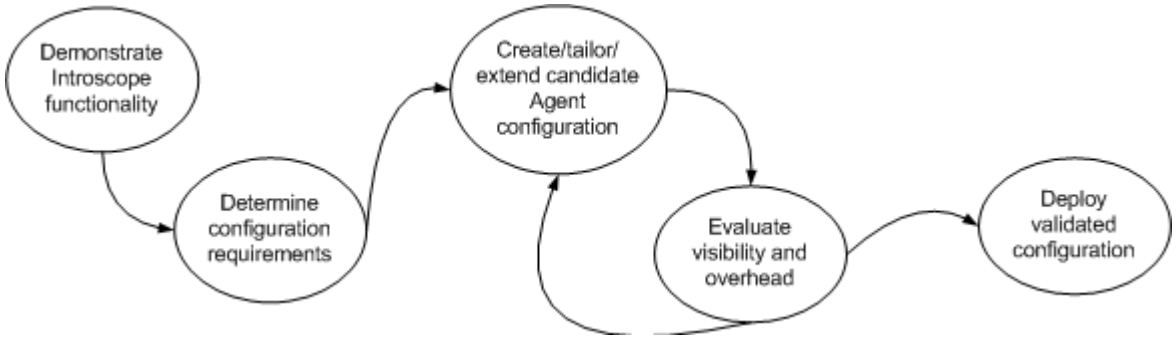
The Java Agent allows Introscope to collect minute details about how your applications are performing. Data is collected from your applications by the Java Agent. What types of data collected depends on which ProbeBuilder Directives (PBDs) files you choose to implement. Several standard PBDs are included when you install the Java Agent, as well as specific PBDs for your application server. Fine tuning the tracers and directives in the PBD files will deliver the metric information you want to monitor for your environment.

The figure below illustrates the key components of an Introscope environment.



Planning a Java Agent implementation

It is important to develop the right Java Agent configuration for your applications and the environments in which it runs. The figure below illustrates the key processes in a Java Agent implementation process.



Discover Introscope functionality

The first step in developing an Introscope implementation involves "test driving" the default Introscope Java Agent configuration. A default Java Agent configuration demonstrates data collection functionality and is key to understanding and evaluating the out-of-the box features of the Java Agent and Introscope as a whole. When you install Introscope, a default Java Agent configuration is included.

The Java Agent provides a variety of data collection options out-of-the box and can be customized to collect more environment-specific data. However, the more metrics a Java Agent collects, the more system resources it consumes.

When evaluating the environment, the primary goal is to understand the depth and breadth of Introscope's data collection and application management features. As you refine your Java Agent configuration, you will streamline data collection to balance the depth of data collection against overhead constraints and configure Java Agent features that help manage and limit resource consumption.

Determine configuration requirements

Before introducing Introscope into your environment, whether pre-production or live, you should determine your data collection requirements. This information will help you tailor the data collection behaviors of the Java Agent, and evaluate the impact on overhead through alternative configurations of the Java Agent.

Since Introscope is employed across an application lifecycle—in development, test, and production—your monitoring goals, environmental constraints, and service level requirements will change over time. You will need to configure Java Agents differently in each phase or environment.

Java Agent configuration is a trade-off between visibility vs. overhead. The goal is to obtain optimal visibility at a reasonable cost.

In pre-production environments, such as development and QA, you typically configure a higher level of data collection to provide deeper visibility into the performance characteristics of the application.

In production or production-like environments, you reduce the level of metric reporting to control Java Agent overhead, and when appropriate, implement optional configurations, such as Virtual Agents or agent failover.

If you intend to collect data from multiple environments, you will need to develop an appropriate Java Agent configurations for each.

Define Java Agent configuration

After defining your configuration requirements based on your application and its operating environment, you should create a "candidate" agent configuration. Most Java Agent behaviors are configured in the agent profile. Some features may also require some configuration in your application server, or required other configuration steps.

Depending on the complexity of your configuration and the target environment, you may choose to build up the agent configuration in stages, so that you can evaluate the impact of each add-on component—such as LeakHunter, ErrorDetector, or Introscope PowerPacks and ensure it is working before adding more.

Evaluate Java Agent performance overhead

When evaluating a Java Agent configuration, verify that the metrics collected provide sufficient visibility into application performance and availability, and that the volume of metrics do not impose an unacceptable load on the operating environment. The Java Agent should not report more metrics than are necessary to identify and localize performance and availability problems.

To effectively understand and evaluate Java Agent overhead, you must understand the performance characteristics of the application prior to Introscope-enabling it.

For example, you can load test your application before and after implementing out-of-the-box monitoring to verify impact. Similarly, a conservative approach is to extend data collection in a controlled fashion—for instance, one PowerPack at a time—and evaluate the impact of each add-on individually.

Occasionally, too many Java classes are selected for monitoring in a ProdeBuilding Directive (PBD) file, causing the Java Agent to start incorrectly, or to experience a "hang". If this happens, use the `AutoProbe.log` file to identify the classes that caused the Java Agent to hang and add a skip directive to the PBD file, skipping the classes that may have caused the problem. For more information about adding skip directives to your custom PBD files, see *Skip directives* on page 92.

Validate and deploy Java Agent configuration

After you have verified that a candidate agent configuration provides the visibility required for the target environment without imposing unacceptable overhead, you should deploy the validated configuration to that environment.

In practice, the process of deploying a validated configuration includes installing the validated configuration artifacts—specifically `IntroscopeAgent.profile` and modified or custom .pbd files—to the target environment.

Implementing the Java Agent

The sections that follow describe the steps in implementing data collection and the Java Agent directory structure and configuration artifacts.

Basic implementation

The Java Agent implementation process is as follows:

Step 1 Install the Java Agent on the target JVM. For more information, see *Installing and Configuring the Java Agent* on page 19.

Step 2 Configure the properties in `IntroscopeAgent.profile` that govern the operating and data collection behaviors of the Java Agent, including which PBDs to use during the ProbeBuilding process and optional ProbeBuilding behaviors. For more information, see *AutoProbe and ProbeBuilding Options* on page 45 and *ProbeBuilder Directives* on page 73.

Step 3 Use a supported method of ProbeBuilding and desired PBDs to instrument your applications, and configure ProbeBuilding options.

Step 4 Restart your application and start data collection.

Java Agent Configuration Options

This section is an overview of configurable agent behaviors.

Communications with Enterprise Manager

You must configure the location of the Enterprise Manager to which the Java Agent reports. If you do not, the Java Agent will try to connect with the Enterprise Manager on localhost port 5001 by default. If the agent will connect to an Enterprise Manager cluster, you must configure it to connect to a Collector Enterprise Manager. To enable an agent to failover to a secondary Enterprise Manager, you must define connection properties and connection order as well. For more information, see *Configuring connection to the Enterprise Manager* on page 36.

Java Agent naming

You can define a Java Agent's name explicitly or configure an automated mechanism for agent naming. By default, the Java Agent profile explicitly assigns an agent name, for instance "WebLogic Agent". For more information, see *Java Agent Naming* on page 99.

Virtual Agents

If you want multiple agents to monitor separate instances of a clustered application, you must configure those agents as a Virtual Agents which allows you to aggregate metrics at the application level. For more information, see *Using Virtual Agents to Aggregate Metrics* on page 119.

Logging options

By default, the Java Agent writes information log messages to the console window and log files. You can configure the Java Agent for more detailed logging. For more information, see *Configuring logging options* on page 113.

Domains

Unless you assign a Java Agent to a custom Introscope Domain, it is part of the SuperDomain by default. For information about Domains and their use in configuring user permission, see the *Introscope Installation Guide*.

ProbeBuilding alternatives and options

Introscope provides multiple methods of Introscope-enabling your applications: JVM AutoProbe, Application Server AutoProbe, and Manual ProbeBuilder. JVM AutoProbe is typically used in environments that support it, and is enabled by default in the agent profiles for those environments. There are optional features, such as dynamic instrumentation, available if your agent runs on JVM 1.5. For more information, see *AutoProbe and ProbeBuilding Options* on page 45.

» **Important** Application Server Autoprobe is not supported on any JVM 1.5 and above platforms.

ProbeBuilder Directive (PBDs)

The ProbeBuilder Directive (PBDs) used during the ProbeBuilding process determine the metrics that the agent reports. You configure a list of the desired PBDs or PBLs (lists of PBDs) in the agent profile. The default list specified in the profile results in the typical level of probe-building, appropriate for production environments where overhead is at a premium. You configure more detailed probe building to obtain more metrics in development or QA environments in the agent profile. You can further control the ProbeBuilding process by customizing PBDs to skip classes or packages, or to instrument custom classes and custom or private methods that the default PBDs do not specify. For more information, see *ProbeBuilder Directives* on page 73.

Data collection and reporting option

Most data collection behaviors are controlled by Java Agent properties.

- Socket Metrics—By default, the Java Agent does not report input and output bandwidth rate metrics for individual sockets. For more information, see *Turning off socket metrics* on page 113.
- URL Groups for Blame Reporting—To control the way that metrics for front-ends are aggregated and presented in the Investigator in WebView and the Workstation, you must configure URL groups. The Java Agent profile contains properties for specifying URL groups. For more information, see *Using URL groups* on page 134.
- Stall Event Reporting—By default, the Java Agent reports stalls as events, and stores them in the Transaction Event Database. You can disable this behavior, or tailor the stall reporting behavior. For more information, see *Disabling the capture of stalls as Events* on page 146.

- JMX and JSR-77—You can optionally configure the Java Agent to report JMX metrics, JSR-77 metrics, or WebLogic Diagnostic Framework metrics. For more information, see *Enabling JMX Reporting* on page 163 and *Enabling WebLogic Diagnostic Framework* on page 181
- Transaction Tracing Behavior—You can tailor the behavior of the automatic transaction tracing the agent performs, and configure the collection of User IDs for Servlet and JSP invocations. For more information, see *Configuring Transaction Trace Options* on page 141.
- PMI—In WebSphere environments you can configure the reporting of WebSphere Performance Monitoring Infrastructure (PMI) metrics. For more information, see *Configuring WebSphere PMI* on page 177.
- Platform Monitoring—Platform monitors enable the agent to report system metrics, including CPU statistics, to the Enterprise Manager. Platform monitors on all operating systems except Windows Server 2003 and AIX are automatically enabled upon agent installation. Windows Server 2003 and AIX platform monitors require a minimal configuration to work. For more information, see *Configuring Platform Monitoring* on page 171.
- SQL Agent—Introscope SQL Agent is installed automatically with the agent installation. This agent extension provides visibility into the performance of individual SQL statements. For more information, see *Configuring the Introscope SQL Agent* on page 147.

Wily CEM integration

There are several steps you are required to perform to ensure that the Java Agent integrates with a Wily CEM installation. See the *CA Wily Customer Experience Manager Integration Guide* for more information about integrating your Java Agent with Wily CEM.

CHAPTER 2

Installing and Configuring the Java Agent

This chapter has instructions for installing the Java Agent.

Before you start

This section lists some of the key information, decisions, and resources that should be identified or obtained before you install and configure your Java Agent.

Application server support

The following table lists the application servers supported by the Java Agent, and any requirements that application server has for functionality with the Java Agent:

Application Server	Requirements
Apache Tomcat	Apache Tomcat 4.1, 5.0, 5.5, or 6.0
JBoss	Version 4.0.3 SP1, 4.0.2, or 4.2x
Fujitsu Interstage	Japanese Version 6.0 (both Standard and Enterprise Edition)
Oracle 10g	Oracle 10g version 10.0.3 Application Server and any required updates.
SAP NetWeaver	SAP NetWeaver 6.4 (NWO4)
Sun ONE	Sun ONE application server and any required updates. For Sun ONE version 7.0, the minimum Sun ONE versions required for Introscope integration are: ■ Sun ONE AS Platform Edition 7.0.0_01 (update 1) ■ Sun ONE AS Standard Edition 7.0.0_01 (update 1) To download the appropriate application server versions, see http://wwws.sun.com/software/download/app_servers.html.
WebLogic	Version 6.1, or higher, and any required patches.
WebSphere	WebSphere application server 5.1, or higher, and any required patches.
WebSphere on z/OS	WebSphere application server 5.0, or higher, and any required patches.

» **Important** The 8.0 Introscope Agent does not support web applications running on Java 1.3.x. If you have a Java 1.3.x-based application, use the Introscope 7.2 agent to manage that application. The Introscope 8.0 Enterprise Manager supports agents back to version 6.0.

When the Java Agent is installed on an application server, after the server and Java Agent start, a Wily log directory is created here: `<Agent_Home>/wily/logs`. The application server process must have full read/write/execute permissions on the Wily Java Agent directory. To accomplish this, install the Java Agent on the same operating system as the user who runs the application server process. Or, install the Java Agent as a different user, then use the `chmod` command to bestow the necessary permissions.

Enterprise Manager connection information

The Java Agent runs on the JVM that runs the applications you wish to monitor, and connects to the Introscope Enterprise Manager. If your agent reports to a clustered Enterprise Manager you must configure it to connect to a *Collector* Enterprise Manager.

If you have multiple Enterprise Managers, clustered or not, you can configure your Java Agent to failover to an alternate Enterprise Manager if it disconnects from its primary Enterprise Manager. For more information on connecting to an Enterprise Manager, see *Configuring connection to the Enterprise Manager* on page 36. For more information on agent failover to alternate Enterprise Managers, see *Configuring Java Agent Failover* on page 123. For more information on clustered Enterprise Managers, see the *Introscope Configuration and Administration Guide*.

ProbeBuilding method and options

The Java Agent supports several methods of instrumenting your applications. CA Wily recommends using JVM AutoProbe to dynamically instrument all classes loaded by the JVM, adding probes that generate metrics from the Java bytecode. JVM AutoProbe is supported if you use:

- Java 1.5 JVM or higher
- a Sun or IBM 1.4 JVM
- JRockit 1.3 or higher
- HP Hotspot 1.3 or higher

The majority of CA Wily Introscope users instrument their applications using JVM AutoProbe.

» Note On OS/400, Application Server AutoProbe is not supported.

If your JVM does not support JVM AutoProbe, or you prefer to use another method, you can either configure Application Server AutoProbe, as described in *AutoProbe for Application Servers* on page 63, or perform the ProbeBuilding process manually as described in *Manual ProbeBuilding* on page 225.

» **Important** CA Wily highly recommends using JVM AutoProbe to instrument your applications. Other methods of instrumentation should only be used if JVM AutoProbe fails.

Planning the installation, configuration, and evaluation process

Most Introscope deployments include one or more extension products such as LeakHunter, Error Detector, or PowerPacks that extend data collection.

Before starting to implement the agent, map out the sequence of installations, review and decide what configuration options are appropriate for the agent and extension products, and identify at which points you wish to load test or otherwise evaluate the configuration.

» **Note** The Java Agent will require at least 30MB of free disk space on the system on which it will be installed.

Installing the Java Agent

There are two methods of installing a Java Agent:

- Use the Java Agent installer, which performs several tasks for you. For more information, see *The Java Agent installer*, below.

OR

- Manually install the Java Agent, where you perform all installation tasks. For more information, see *Manual installation* on page 29.

The Java Agent installer

The Java Agent installer has automated several of the installations tasks, making it easier to deploy agents across large environments.

The table below lists the available Java Agent installers. Use the installer appropriate for your environment:

Environment	Installers
Windows	■ `IntroscopeAgentInstaller8.0windows.zip` Contains `IntroscopeAgent8.0windows.exe` and a responsefile.
UNIX	■ `IntroscopeAgentInstaller8.0unix.tar` Contains `IntroscopeAgent8.0unix.bin` and a responsefile.

Environment	Installers
z/OS	■ `IntroscopeAgentInstaller8.0zos.tar` Contains `IntroscopeAgent8.0zOS.jar`, a `runinstaller.sh` script, a `tmppath` file, and a responsefile.
OS/400	■ `IntroscopeAgentInstaller8.0os400.zip` Contains the `IntroscopeAgent8.0os400.jar`, a `runinstaller.sh` script, and a responsefile.

» **Note** The Java Agent installer must be launched with JVM 1.4 or later. If you specify an application server JVM, that also must be version 1.4 or later.

» **Important** Users with applications running on Java 1.3 must use the 7.2 agent; the 8.0 agent does not support Java 1.3.

When you use the Java Agent installer, it performs the following tasks for you:

- Installs the Java Agent, including platform monitors and PBDs for the target application.
- Edits certain settings in the `IntroscopeAgent.profile`, *or* installs an agent profile provided by you.
- Generates the connector `.jar`, if appropriate.
- Installs custom PBDs, add-ons, or PowerPacks supplied by you in `.zip` or `.tar` formats.
- Before exiting, the installer prints the location of a text file that contains application server-specific "Next Steps".

Installing the Java Agent in GUI mode

The GUI mode of the Java Agent installer allows you to make selections from drop-down menus.

To install the Java Agent using the installer in GUI mode:

1 Choose an installer that matches your target environment and open it.

» **Important** If you are installing the Java Agent on a UNIX system, you must use the `tar -xvf` command to extract the `.tar` file. Do not use `unzip`.

2 Follow the onscreen prompts to install the Java Agent. You will need to know the following:

- The location of the application server root directory.

  If you do not want to specify an application server root directory, you should accept the default response (C:\ on Windows, / on UNIX).
- The application server type.

- Where you want the install directory to be located.

  If you supplied a root directory on the first screen of the install process, the installer suggests the application server root directory; if no root directory was supplied, the default Introscope install directory is suggested.
- Whether you want to use a custom agent profile, or create a new agent profile when the agent is installed.
  - If you choose to use a custom agent profile, the installer will ask you for the location of the this file. Supply the fully qualified path to the file location.
  - If you choose to create a new agent profile, you must decide to use either typical or full instrumentation, the agent and process name, and the Enterprise Manager host and port.

» **Note** Empty values are allowed for both the agent and process name, and the Enterprise Manager host and port. These values can be configured after installation.

- The application server JVM. The installer will suggest a type based on previous information you supplied.
- The instrumentation type.
- The location of a "pickup folder". Any `.zip` or `.tar` files in this folder are extracted into the agent directory, by default `<Agent_Home>/wily`.

If unusable input is provided for any of the above options, the installer will provide reasonable error messages explaining the problem and how to correct it.

3 When all information has been configured, click **Install**. The installer will install the Java Agent and print the location of a text file containing:

- the specific steps the installer took to install the Java Agent.
- application server-specific "Next Steps".

Installing the Java Agent in console mode

The Java Agent installer in console mode is supported on most non-Windows platforms. On these platforms, such as UNIX, z/OS, or OS/400, the console installer launches automatically.

When using the Java Agent installer in console mode, the console will prompt you to enter information about your installation. For example, you will need to know:

- The location of the application server root directory.

  If you do not want to specify an application server root directory, you should accept the default response (C:\ on Windows, / on UNIX).

- The application server type. The Java Agent can monitor the following application servers:
  - Default
  - JBoss
  - Tomcat
  - WebLogic
  - WebSphere
  - Sun One
  - Oracle
  - Interstage
- Where you want the install directory to be located.
- Whether you want to use a custom agent profile, or create a new agent profile when the agent is installed.
  - If you choose to use a custom agent profile, the installer will ask you for the location of the this file. Supply the fully qualified path to the file location.
  - If you choose to create a new agent profile, you must decide to use either typical or full instrumentation, the agent and process name, and the Enterprise Manager host and port.

» **Note** Empty values are allowed for both the agent and process name, and the Enterprise Manager host and port. These values can be configured after installation.

- The agent name and process name.
- Connection settings for the Enterprise Manager.
- The instrumentation type and level.
- The application server JVM.
- The location of a "pickup folder". Any `.zip` or `.tar` files in this folder are extracted into the agent directory, by default `<Agent_Home>/wily`.

The console displays the selections you made during installation and asks for confirmation of your selections. Once you have confirmed your settings, the Java Agent is installed.

Installing the Java Agent in silent mode

The Java Agent can be installed in silent mode, which requires no interaction with a GUI or console. Silent installations use the settings specified in a response file.

Installing the Java Agent using silent mode takes several steps:

Step 1 Open the `SampleResponseFile.Agent.txt` file, located in the same directory as the executable agent installer.

Step 2 Edit the `SampleResponseFile.Agent.txt` file to reflect your preferred settings.

Step 3 Place the `SampleResponseFile.Agent.txt` file in any directory.

The silent installer performs these tasks:

- places Java Agent files on the filesystem
- edits the `IntroscopeAgent.profile`
- generates a connector .jar, if necessary
- copies certain `.jar` files into your application server directories, where applicable
- generates a "Next Steps" readme file

The silent installer does not stop or start the application server, or modify the application server classpath. These tasks need to be performed manually.

» **Important** The silent installer does not perform upgrades. By default, if the silent installer detects a preexisting Java Agent in the specified install directory, no installation will be performed. CA Wily recommends you do not use this installer to overwrite an existing Java Agent installation.

To install the Java Agent using the installer in silent mode:

1. Open the `SampleResponseFile.Agent.txt`.
2. Set the following properties:
   - `USER_INSTALL_DIR=`
     Specify the directory where the Java Agent files are to be installed. CA Wily recommends selecting the application server's root directory.

     On all platforms, the file path must end with a file separator. On Windows, backslashes must be escaped. For example:
     - On Windows: `C:\\myAppServerHome\\`
     - On UNIX: `/myAppServerHome/`

- `silentInstallChosenFeatures=Agent,Documentation`
  Specify the agent features to install. This must be a comma-delimited list. Valid values are `Agent` and `Documentation`, which are case-sensitive. If the `Documentation` feature is included, basic documentation is installed into the `<Agent_Home>\wily` directory.
- `appServer=Default`
  Specify the application server type. Valid values are `Default`, `JBoss`, `Tomcat`, `WebLogic`, `WebSphere`, `Sun ONE`, `Oracle`, or `Interstage`. Application server-specific ProbeBuilder Directives (PBDs) are installed with the Java Agent.
- `appServerHome=`
  You can also specify the home directory of the application server to monitor. This is an optional configuration. To use it, uncomment and set the property.

  The installer uses this information to copy certain `.jar` files into the application server directories if necessary.

  On all platforms, the path must end with a file separator. On Windows, backslashes must be escaped. For example, on Windows:

  ```
  C:\\apache\\tomcat\\5.0.30\\
  D:\\bea\\weblogic700\\
  ```
- `appServerJavaExecutable=`
  You can also specify the Java executable used to launch the monitored application server. The silent installer uses this information to check the JVM vendor and version, and generates a connector jar if necessary.

  On Windows, backslashes in the path must be escaped. Some sample values for this property are:

  - On Windows: `C:\\Program Files\\Java\\jre1.5.0_06\\bin\\java.exe`
  - On UNIX: `/usr/bin/java`
- `instrumentationLevel=Typical`
  Specify the level of instrumentation to use. Valid values are `Full` or `Typical`, which are case-sensitive.

  The `Full` setting achieves more detailed reporting and is recommended for test environments.

  The `Typical` setting provides less detail with reduced overhead, and is recommended for production environments.
- `instrumentationType=JVM AutoProbe`
  Specify the ProbeBuilding method that is used to instrument the application. Case-sensitive valid values are:

  - JVM AutoProbe
  - Application Server AutoProbe
  - Manual ProbeBuilding

CA Wily recommends using JVM AutoProbe. See the *AutoProbe and ProbeBuilding Options* on page 45 for more information on ProbeBuilding methods.

- `agentName=Default Agent`
- `processName=Default Process`
  Specify the agent and process names. Spaces and empty values are allowed.
- `emHost=localhost`
- `emPort=5001`
  Specify the hostname and port of the Introscope Enterprise Manager to which this agent will connect.
- `#alternateAgentProfile=`
  You can choose to use a custom agent profile. Uncomment and specify the absolute path to a custom agent profile. If you specify an alternate file, it will be used instead of the default agent profile, and all of the properties in the "Agent Settings" section of the `SampleResponseFile.Agent.txt` file will be ignored, except for the `instrumentationType` property.

  On Windows, backslashes must be escaped. Some sample values for this property are:

  - On Windows: `C:\\customAgentProfiles\\CustomIntroscopeAgent.profile`
  - On UNIX: `/home/iscadmin/customAgentProfiles/CustomIntroscopeAgent.profile`
- `#pickupFolder=`
  You can also specify the absolute path to a "pickup folder". The pickup folder provides a convenient way of installing extensions or add-ons alongside the base agent. Any `.zip` or `.tar` files present in the pickup folder will be extracted into the agent directory at install time. On all platforms, the path must end with a file separator. On Windows, backslashes must be escaped. For example:

  - On Windows: `C:\\pickupFolderContainingAddOns\\`
  - On UNIX: `/home/iscadmin/customAgentProfiles/pickupFolderContainingAddOns/`

3. Save the `SampleResponseFile.Agent.txt`.
4. Select the appropriate command format from the list below, and enter it at the command line to invoke the installer:

```
installer.exe -f <absolute path to response file>
installer.bin -f <absolute path to response file>
java -classpath installer.jar install -f <absolute path to response file>
```

Manual installation

Extract the Java Agent installer archive into a Java working directory—often the application server's home directory. For information about the directory structure of the agent installation, see *Java Agent installation directories and files* on page 31.

The Java Agent requires 30 MB of free disk space to be installed.

To manually install the Java Agent:

1. Select an installer archive for your specific JVM. For a list of installer archives, see *Java Agent installer archives*, below.
2. Extract the files of the installer archive into a location your JVM can access.

   » **Important** If you are installing the Java Agent on a UNIX system, you must use the `tar -xvf` command to extract the `.tar` file. Do not use `unzip`.

3. Configure the properties in `IntroscopeAgent.profile` to your specific environmental needs. This file governs the Java Agent's operating and data collection behaviors, including which PBDs to use during the ProbeBuilding process, optional ProbeBuilding behaviors, and the connection to the Enterprise Manager. You will need to know the location of the files extracted in *step 2* above.

   Configuration options for the `IntroscopeAgent.profile` are covered in the next sections of this chapter and the rest of this Guide. See *Configuring connection to the Enterprise Manager* on page 36 to get started.
4. Use a supported method of ProbeBuilding and desired PBDs to instrument your applications, and configure ProbeBuilding options. For more information about ProbeBuilding, see *Configuring ProbeBuilder options* on page 41.
5. Restart your application.

Java Agent installer archives

The table below lists the Java Agent installer archives for different application servers and operating systems.

Application Server	Installer Archive Packages
WebLogic	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0weblogic.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0weblogic.windows.zip ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0weblogic.zOS.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0weblogic.os400.zip
WebSphere	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0websphere.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0websphere.windows.zip ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0websphere.zOS.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0websphere.os400.zip
Sun ONE	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0sunoneas.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0sunoneas.windows.zip
Oracle 10g	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0oracleas.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0oracleas.windows.zip
SAP NetWeaver	***For SAP support only:*** **Windows:** IntroscopeForSAPNetWeaverConversionKit8.0.0.0.windows.zip **Solaris, HP-UX, AIX, and Linux:** IntroscopeForSAPNetWeaverConversionKit8.0.0.0.unix.tar
Apache Tomcat	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0tomcat.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0tomcat.windows.zip
Fujitsu Interstage	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0interstage.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0interstage.windows.zip

Application Server	Installer Archive Packages
JBoss	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0jboss.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0jboss.windows.zip
Other	■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0default.unix.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0default.windows.zip ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0default.zOS.tar ■ IntroscopeAgentFilesOnly-NoInstaller8.0.0.0default.os400.zip

Java Agent installation directories and files

Installing a Java Agent creates the following directory structure:

```
wily\
   connectors
   docs
   ext
   hotdeploy
   install
   tools
   UninstallerData
```

Java Agent operating and data collection behaviors are controlled by configuration properties stored in the agent profile and the PBDs it references.

Contents of the wily directory

The `wily` directory contains:

- `Agent.jar`—The Java Agent executable .jar file
- `WebAppSupport.jar`—Contains startup classes that you can configure to allow the Java Agent to obtain management information from an application server.
- `IntroscopeAgent.profile`—The agent profile contains properties that control the behavior of the agent and AutoProbe. Defaults are supplied for many properties; the default values for certain properties vary, depending on the application server your agent installation supports. You can also change the location of the agent profile. For more information, see *Configuring IntroscopeAgent.profile location* on page 186.
- ProbeBuilder Directives (PBDs)—Contains the standard ProbeBuilder Directives (PBDs) and ProbeBuilder Lists (PBLs) provided with the Java

Agent, as well as application server-specific PBDs, which vary depending on the application server your agent installation supports. For more information on the standard PBDs and PBLs, see *Default ProbeBuilder Directive (PBD) files* on page 75 and *Default ProbeBuilder List (PBL) files* on page 76.

Contents of the wily\connectors directory

The `wily\connectors` directory contains:

- `CreateAutoProbeConnector.jar`—Used in configuring JVM AutoProbe in supported environments.

Contents of the wily\docs directory

The `wily\docs` directory contains:

- `IntroscopeInstallUpgradeGuide.pdf`—how to install and upgrade other components of Introscope.
- `IntroscopeSizingGuide.pdf`—contains background, instructions, best practices, and tips for optimizing the sizing and performance of your Introscope deployment and environment.
- `JavaAgent.pdf` (this guide)—details how to install and configure the Introscope Java Agent.

Contents of the wily\hotdeploy directory

Directives placed in this directory will be automatically deployed to the Java Agent. When you create custom PBDs, save them to this directory. When PBDs are placed in this directory, you do not have to edit the `IntroscopeAgent.profile` to pick up new or changed PBDs. Also, if you have enabled dynamic ProbeBuilding, you do not have to restart your applications to apply new PBDs.

For more information about creating custom PBDs, see *ProbeBuilder Directives* on page 73 and *Creating custom tracers* on page 84. For more information about dynamic ProbeBuilding, see *Dynamic ProbeBuilding* on page 56.

» **Note** Any ProbeBuilder Lists (PBLs) placed in this directory will be ignored by the Java Agent.

The hotdeploy directory allows Introscope administrators to deploy new directives more quickly and easily, without editing the `IntroscopeAgent.profile`, and potentially without restarting applications. This ability heightens the need for caution. If your custom PBDs contain invalid syntax, or are configured to collect too many metrics, the impact will be felt more quickly. Invalid PBDs will cause AutoProbe to shut off and PBDs that collect too many metrics can affect application performance.

To address this, CA Wily recommends:

- testing and validating all directives in QA and performance environments before pushing them out to production environments.
- ensuring that your server environment's change control process is updated to reflect the new option for deploying PBDs.

Additionally, you can decide not to use the `hotdeploy` directory.

To unconfigure the hotdeploy directory:

1. Move any of the custom PBDs stored in the hotdeploy directory to the main `<Agent_Home>/wily` directory.
2. Open the `IntroscopeAgent.profile`.
3. Remove hotdeploy from the `introscope.autoprobe.directivesFile` property.
4. Add the PBDs you want to use to the `introscope.autoprobe.directivesFile`, for example:

```
introscope.autoprobe.directivesFile=default-
  typical.pbl,custom1.pbd,custom2.pbd,custom3.pbd
```

5. Save the `IntroscopeAgent.profile` and restart the agent.

Contents of the wily\ext directory

The `wily\ext` directory contains:

- `CEMTracer.jar`—Used in environments that integrate Wily Customer Experience Manager (CEM) with the Java Agent.
- `ServletHeaderDecorator.jar`—Used in environments that integrate Wily CEM with the Introscope agent.
- `Inheritance.jar`—Used in Java 1.5 environments to enable AutoProbe to instrument multiple levels of subclasses of a probed class.
- `JavaI5DynamicInstrumentation.jar`—Used in Java 1.5 environments to enable dynamic instrumentation, which allows you to implement new and changed PBDs without restarting the managed application or the agent.
- `ProbeBuilder.jar`—Contains ProbeBuilder executable.
- One of the following extensions to ProbeBuilder.jar
  - `BasicDirectiveLoader.jar`
  - `SignedJARDirectiveLoader.jar`
  - `UnifiedDirectiveLoader.jar`
- `SQLAgent.jar`—Contains SQL Agent executable.
- `Supportability-Agent.jar`—Supportability extensions for use by Wily Technical Support in agent support.

Contents of the wily\install directory

The `wily\install` directory contains:

- `autogenerated.responsefile.<date_and_time_stamp>`—an automatically generated response file that reflects the settings you chose during installation. This file can be used to replicate the installation on other systems.
- `Introscope_Agent_8.0_InstallLog.log`—if you installed your Java Agent using the agent installer, this log details the actions taken by the automatic installer.
- `IntroscopeAgentInstallation_README.txt`—details the next steps you should take after installing the Java Agent, based on the decisions you made during the automatic installation.
- `SampleResponseFile.Agent.txt`—Sample silent responsefile

Contents of the wily\tools directory

The `wily\tools` directory contains:

- `URLGrouper.jar`—Contains the URLGrouper, a command-line utility that analyzes web server log files and produces BRTA property settings for a set of URL Groups. For more information about the URLGrouper, see *Running the URLGrouper* on page 139.

Contents of the wily\UninstallerData directory

The `wily\UninstallerData` directory contains:

- `Uninstall Introscope Agent .exe`—Use this file to uninstall the Java Agent.

JBoss configuration

If you installed the Java Agent on a JBoss system using either installation method, there are further configurations you must perform to enable the reporting of JBoss metrics to Introscope. You must configure JBoss for Introscope, and deploy web application support for JBoss. When deploying web application support for JBoss, the Introscope service is deployed as a stand alone XML service descriptor.

This functionality requires Introscope 8.0 or higher. It is compatible with JBoss 4.0.3 SP1, 4.0.2, and 4.2x. The JMX service can run with Introscope 6.0 but the following error is logged:

```
8/16/05 01:40:25 PM PDT [ERROR] [IntroscopeAgent] Failed to activate JMX
  data collection
```

```
java.lang.IllegalArgumentException: The MBean server builder implementation
  class org.jboss.mx.server.MBeanServerBuilderImpl was not found:
  java.lang.ClassNotFoundException:
  org.jboss.mx.server.MBeanServerBuilderImpl
```

This error can be ignored - JMX data collection has in fact been activated.

To configure JBoss for Introscope:

1. Modify the `run.bat` file in the `bin` directory of your JBoss installation by adding the following:

```
rem =====================================================================
rem Enable Introscope
rem =====================================================================
rem Use this for Java 1.5
set JAVA_OPTS= -javaagent:%JBOSS_HOME%\wily\Agent.jar -
  Dcom.wily.introscope.agentProfile=%JBOSS_HOME%\wily\IntroscopeAgent.pro
  file %JAVA_OPTS%
rem Otherwise, use this
rem set JAVA_OPTS= -Xbootclasspath/
  p:%JBOSS_HOME%\wily\connectors\AutoProbeConnector.jar;%JBOSS_HOME%\wily
  \Agent.jar -
  Dcom.wily.introscope.agentProfile=%JBOSS_HOME%\wily\IntroscopeAgent.pro
  file %JAVA_OPTS%
rem=====================================================================
```

   » **Note** The above assumes you installed the Java Agent in the root directory of your JBoss installation. If not, modify the file paths accordingly.

2. Save the `run.bat` file.
3. Open the `IntroscopeAgent.profile` located in the `<Agent_Home>\wily` directory and set the following property:

```
introscope.agent.jmx.enable=true
```

4. Save the `IntroscopeAgent.profile`.

To deploy web application support for JBoss:

1. Place the `WebAppSupport.jar` file, located in the `<Agent_Home>\wily` directory of your Java Agent installation, in to the `/server/default/lib` directory of your JBoss installation.

   » **Note** This assumes you are using the default configuration. If not, place the file in the appropriate directory.

2. Create an XML file called `introscope-jboss-service.xml` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Introscope Custom Service for JBoss
-->
```

```
<service>
  <mbean code="com.wily.introscope.api.jboss.IntroscopeCustomService"
    name="user:service=IntroscopeCustomService"/>
</service>
```

3 Place the `introscope-jboss-service.xml` file in to the `/server/default/deploy` directory of your JBoss installation

» **Note** This assumes you are using the default configuration. If not, place the file in the appropriate directory.

JBoss PBDs and PBLs

When you install the Java Agent on a JBoss application server, JBoss-specific PBDs and PBLs are installed in the <Agent_Home>\wily directory. Use these files to taylor your JBoss data collection:

- `jboss4x.pbd`
- `jsf.pbd`
- `jsf-toggles-full.pbd`
- `jsf-toggles-typical.pbd`
- `jboss-full.pbl`
- `jboss-typical.pbl`

Configuring connection to the Enterprise Manager

To report metrics, the Java Agent must connect to an Enterprise Manager. You configure how the agent connects to the Enterprise Manager by changing properties in the `IntroscopeAgent.profile`, located in the `<Agent_Home>/wily` directory. CA Wily recommends putting the Enterprise Manager on a system separate from the agent systems.

The default communications settings in the `IntroscopeAgent.profile` enable an agent to connect to an Enterprise Manager located on the same system as the agent. To comply with the recommended deployment (different systems), you must modify the `IntroscopeAgent.profile` to connect the agent to a remote Enterprise Manager.

To configure Java Agent connection to the Enterprise Manager:

1 Open the `IntroscopeAgent.profile`.

2 Modify the following properties to specify the location of your remote Enterprise Manager:

- `introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT`
  Defines the host name or IP address of the target Enterprise Manager.
- `introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT`

Defines the Enterprise Manager listening port, which should be the same port specified in the Enterprise Manager property: `introscope.enterprisemanager.port.DEFAULT` . By default, this port is 5001.

» **Note** To change Enterprise Manager properties, open the `IntroscopeEnterpriseManager.properties` file, located in the `<Introscope_Home>/config` directory and modify the desired properties. Save the file for your changes to be implemented.

- `introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT`
  Defines the socket factory used for connections to the Enterprise Manager.

» **Note** The default value of this property, `com.wily.isengard.postofficehub.link.net.DefaultSocketFactory`, is for plain socket communication.

3 Save the `IntroscopeAgent.profile` and start (or restart) the Java Agent.

For more information about connection properties, see *Agent to Enterprise Manager connection* on page 196.

You can also configure alternate Enterprise Managers for the Java Agent to connect to, should the connection to the primary Enterprise Manager be lost. For more information on how to configure these Enterprise Managers, see *Configuring Java Agent Failover* on page 123.

Connecting to the Enterprise Manager with HTTP tunneling

You can configure agents to communicate with an Enterprise Manager over HTTP. This allows communication to pass through firewalls permitting only HTTP traffic.

CA Wily Technology does not recommend configuring agents to tunnel over HTTP if a direct socket connection to the Enterprise Manager is feasible. Tunneling imposes additional CPU and memory overhead on the managed host and Enterprise Manager beyond that expected for a direct socket connection.

» **Important** HTTP/1.1 is required to enable agent HTTP tunneling.

To configure HTTP tunneling:

1 Open the `IntroscopeAgent.profile`.

2 Configure the agent to connect to the HTTP listening port of the Enterprise Manager's embedded web server, using an HTTP tunneling socket factory. Use these properties to specify the agent tunneling connection:

- `introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT`
  Defines the host name or IP address of the target Enterprise Manager.

- `introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT`
  Defines the HTTP listening port of the Enterprise Manager's embedded web server, which is usually specified in the Enterprise Manager property: `introscope.enterprisemanager.webserver.port`. By default, this port is 8081.

  » **Note** To change Enterprise Manager properties, open the `IntroscopeEnterpriseManager.properties` file, located in the `<Introscope_Home>/config` directory and modify the desired properties. Save the file for your changes to be implemented.

- `introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT`
  Set the value to `com.wily.isengard.postofficehub.link.net.HttpTunnelingSocketFactory`.

3. Save the `IntroscopeAgent.profile`.

Configuring a proxy server for HTTP tunneling

You can configure the HTTP tunneled agent to connect through a proxy server to the Enterprise Manager. This is necessary for a forward-proxy server configuration where the agent is running behind a firewall that only allows outbound HTTP traffic routed through the proxy server.

These proxy server configuration properties apply only if the agent is configured to tunnel over HTTP. The proxy server configuration applies to any configured HTTP tunneled connection on the agent, not to a single connection. This is especially important to consider when configuring failover between multiple Enterprise Managers, where the connection to each Enterprise Manager is over HTTP.

» **Important** HTTP/1.1 is required to enable agent HTTP tunneling. If you are using HTTP tunneling, your proxy server must support HTTP Post.

To configure a proxy server for HTTP tunneling:

1. Open the `IntroscopeAgent.profile`.
2. Specify the proxy server properties:
   - *introscope.agent.enterprisemanager.transport.http.proxy.host* on page 189
   - *introscope.agent.enterprisemanager.transport.http.proxy.port* on page 189
3. If the proxy server requires the agent to authenticate with it, set these properties:
   - *introscope.agent.enterprisemanager.transport.http.proxy.username* on page 189
   - *introscope.agent.enterprisemanager.transport.http.proxy.password* on page 189
4. Save the `IntroscopeAgent.profile`.

Connecting to the Enterprise Manager with HTTPS tunneling

The agent can connect to the Enterprise Manager using HTTP over Secure Sockets Layer (SSL) by configuring properties in the `IntroscopeAgent.profile`.

To configure connection through HTTPS:

1. Open the `IntroscopeAgent.profile`.
2. Configure the agent to connect to the HTTPS listening port of the Enterprise Manager's embedded web server, using an HTTP tunneling socket factory.
   - `introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT`
     Defines the host name or IP address of the target Enterprise Manager.
   - `introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT`
     Defines the HTTPS listening port of the Enterprise Manager's embedded web server.
   - `introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT=8444`
   - `introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT=com.wily.isengard.postofficehub.link.net.HttpsTunnelingSocketFactory`
     Set the value to:

     `com.wily.isengard.postofficehub.link.net.HttpsTunnelingSocketFactory`.
3. Save the `IntroscopeAgent.profile`.

Connecting to the Enterprise Manager over SSL

The agent can also connect to the Enterprise Manager using just SSL.

To configure connection through SSL:

1. Open the `IntroscopeAgent.profile`.
2. Configure the agent to connect to the Enterprise Manager's SSL listening port using an SSL socket factory.
   - `introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT`
     Defines the host name or IP address of the target Enterprise Manager.
   - `introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT`
     Defines the Enterprise Manager's SSL listening port.
   - `#introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT`
     Set the value to:

     `com.wily.isengard.postofficehub.link.net.SSLSocketFactory`

3 If the agent needs to authenticate the Enterprise Manager, uncomment and set the truststore property to the location of a truststore containing the Enterprise Manager's certificate.

- `introscope.agent.enterprisemanager.transport.tcp.truststore.DEFAULT`
  Set to either an absolute path or a path relative to the agent's working directory. On Windows, backslashes must be escaped. For example: `C:\\my_truststore`.
- `introscope.agent.enterprisemanager.transport.tcp.trustpassword.DEFAULT`
  The truststore password. Uncomment and set only if needed.

» **Note** If no truststore is specified, the agent by default trusts all certificates.

4 If the Enterprise Manager requires client authentication, the agent must be configured with a keystore. Set the keystore property to the location of a keystore containing the agent's certificate:

- `introscope.agent.enterprisemanager.transport.tcp.keystore.DEFAULT`
  Set to either an absolute path or a path relative to the agent's working directory. On Windows, backslashes must be escaped. For example: `C:\\keystore`.
- `introscope.agent.enterprisemanager.transport.tcp.keypassword.DEFAULT`
  The keystore password. Uncomment and set only if needed. By default no keystore is specified.
- `introscope.agent.enterprisemanager.transport.tcp.ciphersuites.DEFAULT`
  To restrict the enabled cipher suites, set this property to a comma-separated list of cipher suites.

» **Note** If not specified, the default enabled cipher suites are used.

5 Save the `IntroscopeAgent.profile`.

Configuring the Java Agent name

The Java Agent profile provides default values for the agent name and Custom Process name, which appear as nodes in the Investigator hierarchy of metrics reported by the Java Agent.

Depending on your environment, you may wish to configure the Java Agent to obtain a name automatically. For more information about agent naming, and autonaming capabilities, see *Java Agent Naming* on page 99.

If desired, you can define the agent name and process name using these properties in `IntroscopeAgent.profile`.

- `introscope.agent.agentName`—Name the application server that the Java Agent is monitoring.The `agentName` value must start with an alphabetical

character, and cannot contain a percent (%) character. See *introscope.agent.agentName* on page 195 for a description.

- `introscope.agent.customProcessName`—Name the process being monitored. See *introscope.agent.customProcessName* on page 195 for a description.

Configuring ProbeBuilder options

By default, AutoProbe will use the typical PBD set provided with the Java Agent, which results in the collection of a moderate number of metrics. For instructions on how to customize the metric collection level, or to configure optional ProbeBuilding behaviors, see *AutoProbe and ProbeBuilding Options* on page 45.

Upgrading multiple agent types

Some environments have thousands of agents distributed across many different application servers. For example, an environment might have 8,000 agents, with 3,000 agents on WebLogic, 2,000 on WebSphere, and 3,000 on JBoss.

It can become quite a burden to understand the environmental needs for upgrading agents (which agents where need to be upgraded?), and to actually perform the upgrade of all agents to a new version. To ease this burden, the Introscope Java Agent 8.0 release includes superset agent packages, one package for each of the following operating system platforms:

- `IntroscopeAgentFilesOnly-NoInstaller8.0.0.0allappserver.windows.zip`
- `IntroscopeAgentFilesOnly-NoInstaller8.0.0.0allappserver.unix.tar`
- `IntroscopeAgentFilesOnly-NoInstaller8.0.0.0allappserver.zOS.tar`
- `IntroscopeAgentFilesOnly-NoInstaller8.0.0.0allappserver.os400.zip`

» **Important** The superset packages do not include any files for SAP NetWeaver at this time.

Each package contains:

- All application server-specific PBDs and PBLs
- All application server agent profiles, with the application server name embedded in the file name. For example:

  ```
  IntroscopeAgent.weblogic.profile
  IntroscopeAgent.websphere.profile
  ```

» **Note** The default `IntroscopeAgent.profile` has not been included. See *step 3* on page 42 for more information.

- All agent `.jars` and platform monitors suitable for the operating system type

To upgrade multiple agent types using the superset agent packages:

» **Important** This upgrade method is not supported for SAP NetWeaver at this time.

1. Select a superset package appropriate for the target operating system.
2. Extract the selected agent package into the application server's home directory. Follow the manual installation instructions for Java Agent installation. For more information, see *Manual installation* on page 29.

   » **Note** The extra PBDs and PBLs in the `<Agent_Home>/wily` directory that refer to other application servers can be safely ignored.

3. If you have not already configured an `IntroscopeAgent.profile`, select the appropriate `IntroscopeAgent.<application_server_name>.profile`, rename it to `IntroscopeAgent.profile`, and configure the file for use with your environment.

   » **Note** If you have already configured an `IntroscopeAgent.profile`, open the corresponding `IntroscopeAgent.<application_server_name>.profile` file in an editor and look for new properties you may want to use. Transfer these properties to your existing `IntroscopeAgent.profile`.

Uninstalling the Java Agent

Uninstalling the Java Agent requires you to know where the Java Agent was installed for each application being monitored.

If you used the Java Agent installer to install your Java Agent, the uninstaller can be used to remove installed files. Launch the uninstaller and follow the on-screen directions.

To uninstall the Java Agent from JVM 1.5.x, JVM 1.6.x, HP, JRockit JVMs, or WLS 6.1+:

1. Remove the Java Agent switches from the JVM command line. These include:
   - `-Xbootclasspath`
   - `-javaagent`
   - any other Wily-specific arguments (for example: `Dcom.wily.introscope.agentProfile=xxxxx`)
2. Reboot your application.
3. Manually delete the `<Agent_Home>/wily` directory, or run the uninstaller.

To uninstall the Java Agent from WebSphere Application Server 5.0, 5.1, or 6.0:

1. Connect to the Administration Console for your WebSphere Application Server.
2. Remove the Java Agent switches from the "Generic JVM Arguments".
3. Reboot your application.

4 Manually delete the `wily` directory, or run the uninstaller.

Uninstalling the Java Agent from z/OS

The recommended way to uninstall the Java Agent from z/OS is to delete the `<Agent_Home>/wily` directory using an `rm -rf` command. This is necessary because the executable uninstaller does not run properly on z/OS due to a third party bug.

» **Note** For an active Introscope 8.0 installation on z/OS, it is important to keep the `UninstallerData` folder intact. If you delete the `UninstallerData` folder, you will not be able to upgrade to future versions of Introscope. Do not delete the `UninstallerData` folder unless you have decided to uninstall the entire instance.

CHAPTER 3

AutoProbe and ProbeBuilding Options

This chapter has information about ProbeBuilding options.

Configuring JVM AutoProbe

CA Wily recommends using JVM AutoProbe to instrument the applications you want to monitor. Configure JVM AutoProbe using the instructions that correspond to your JVM:

It is important to know which version of Java your JVM uses. Different versions of Java require a different configuration of AutoProbe:

Java version	Configuration used
Java 1.5 and later	`-javaagent` syntax. See *JVM 1.5 AutoProbe* on page 51 for more information.
Java 1.4 and earlier	`-Xbootclasspath` syntax. See your specific JVM for more information.

» **Note** For a complete list of products which are integrated with Introscope, and which JVMs are supported, see the *Introscope Compatibility Guide* on the CA Wily Community Site: *http://support.wilytech.com*

Sun, IBM, or HP JVM AutoProbe

This section has instructions for creating and running the AutoProbe Connector for a Sun, IBM, or HP JVM. If your JVM is v1.5, follow the instructions in *JVM 1.5 AutoProbe* on page 51.

Create AutoProbe Connector

1. Change the working directory to `wily/connectors` under the installation directory.
2. Run the Create AutoProbe Connector tool using one of these commands:
   - to specify the JVM using the JVM that is running the tool:
     ```
     java -jar CreateAutoProbeConnector.jar -current
     ```
   - to specify the JVM by passing the JVM directory on the command line:
     ```
     java -jar CreateAutoProbeConnector.jar -jvm <directory>
     ```

   The output is a file with the form: `wily/connectors/AutoProbeConnector.jar`

Run AutoProbe Connector

After you create the AutoProbe Connector for the Sun or IBM JVM, you run it. The way you run the Connector depends on the application server you use. For more information, see the following sections:

Run AutoProbe Connector for WebLogic

Different versions of WebLogic use different versions of Java to run. If you use Java 1.4 or earlier, you will use the following steps to run the AutoProbe connector. If you use Java 1.5 or later, see *JVM 1.5 AutoProbe* on page 51 for more information.

For a comprehensive list of WebLogic versions and their corresponding JDKs, see the *Introscope Compatibility Guide* on the Wily Community Site: *http://support.wilytech.com.*

To run the AutoProbe Connector for WebLogic:

1. Edit the bootstrap classpath in the application startup script to include the `AutoProbeConnector.jar` you created (such as `startMedRecServer.cmd`) using this command:

   `-Xbootclasspath/p:PathToAutoProbeConnectorJar:PathToAgentJar`

   add the -X switch to the final start command at the end of the script, after the `JAVA_VM` and `JAVA_OPTIONS`. The excerpt below shows the correct place to insert the switch:

```
"$JAVA_HOME/bin/java" ${JAVA_VM} ${MEM_ARGS} ${JAVA_OPTIONS}
-Xbootclasspath/p:${WL_HOME}/wily/connectors/
  AutoProbeConnector.jar:${WL_HOME}/wily/Agent.jar
-Dweblogic.Name=${SERVER_NAME}
  -Dweblogic.management.username=${WLS_USER}
  -Dweblogic.management.password=${WLS_PW}
  -Dweblogic.ProductionModeEnabled=${PRODUCTION_MODE}
  -Djava.security.policy="${WL_HOME}/server/lib/weblogic.policy"
  weblogic.Server
```

2 If you are using something other than the default bootstrap classpath, add the `Agent.jar` and `AutoProbeConnector.jar` files to the beginning of your customized bootstrap classpath.

Run AutoProbe Connector for WebSphere 5.0, 5.1, or 6.0

1 Start the WebSphere **Administrator's Console**, and navigate to the **JVM Settings** section for the application server you want to modify:

- For WebSphere 6.0: navigate to **Application Servers > *your_server* > Java and Process Management > Process Definition > Java Virtual Machine**.
- For WebSphere 5.1/5.0: navigate to **Application Servers > *your_server* > Process Definition > Java Virtual Machine**

2 Set the **Generic JVM Arguments** field in the following format:

```
-Xbootclasspath/p:<Path-To-
  AutoProbeConnector.jar>AutoProbeConnector.jar;<Path-To-
  Agent.jar>Agent.jar
-Dcom.wily.introscope.agentProfile=<path-to-
  IntroscopeAgent.profile>IntroscopeAgent.profile
-Dcom.wily.introscope.agent.agentName=<your-agent-Name>
```

For example:

```
-Xbootclasspath/p:<AppServerHome>/wily/connectors/
  AutoProbeConnector.jar:<AppServerHome>/wily/Agent.jar
-Dcom.wily.introscope.agentProfile=<path-to-
  IntroscopeAgent.profile>IntroscopeAgent.profile
-Dcom.wily.introscope.agent.agentName=<your-Agent-Name>
```

Use a semicolon (`;`) as a path separator on Windows systems.

» **Note** The `/p:` option in the command above prepends the supplied path to the default bootstrap classpath.

For example: If you installed the Java Agent in `c:\Program Files\IBM\WebSphere\AppServer\profiles\default`, the Generic JVM Arguments field would be set to:

```
-Xbootclasspath/p:C:/PROGRA~1/IBM/WebSphere/AppServer/profiles/
  default/wily/connectors/AutoProbeConnector.jar;C:/PROGRA~1/IBM/
  WebSphere/AppServer/profiles/default/wily/Agent.jar
-Dcom.wily.introscope.agentProfile=C:/PROGRA~1/IBM/WebSphere/
  AppServer/profiles/default/wily/IntroscopeAgent.profile
-Dcom.wily.introscope.agent.agentName=WebSphereAgent
```

For troubleshooting Java Agent start-up problems, review the basic WebSphere Application Server log (i.e `native_stderr.log`). If an incorrect configuration is placed in the Generic JVM Arguments section and you experience errors starting the server, please review and correct the generic JVM arguments in the server.xml file.

3 Click **OK**.

4 Apply the changes in the Administrator's Console.

5 The default for file encoding for WebSphere 5.x and 6.0 on z/OS is now ASCII, but Introscope expects an EBCDIC file format. When you configure JVM AutoProbe for z/OS, click on **Custom Properties** and add the name value pair:

```
name: com.wily.introscope.default.encoding
value: Cp1047
```

» **Note** This step is for z/OS operating systems only.

6 In WebSphere environments with Java2 Security enabled, for AutoProbe to run correctly, it may be necessary to add permissions to your Java2 Security Policy. If Java2 Security is enabled, follow the instructions in *Modifying Java2 Security Policy* on page 71.

7 Restart WebSphere.

In a moment you will see WebSphere metrics in your Introscope Workstation.

Run AutoProbe Connector for SAP J2EE 6.20

1 Open the file:

```
<drive>:\usr\sap\<J2EE_ENGINE_ID>\j2ee\j2ee_<INSTANCE>\cluster\
  server\cmdline.properties
```

2 Append these commands to **JavaParameters** section:

```
  -Xbootclasspath/p:PathToAutoProbeConnectorJar;PathToAgentJar
  -Dcom.wily.introscope.agentProfile=<path-to-IntroscopeAgent.profile>
  -Dcom.wily.introscope.agent.agentName=<yourAgentName>
```

For example:

```
-Xbootclasspath/
  p:C:\usr\sap\P602\j2ee\j2ee_00\ccms\wily\connectors\AutoProbeConnector.
  jar;C:\usr\sap\P602\j2ee\j2ee_00\ccms\wily\Agent.jar
-Dcom.wily.introscope.agentProfile=C:\usr\sap\P602\j2ee\
  j2ee_00\ccms\wily\IntroscopeAgent.profile
```

3 Restart the SAP server.

Run AutoProbe Connector for NetWeaver 04/SAP J2EE 6.40

1 Run the **SAP J2EE Configtool**.

2 Select the server to modify.

3 Add these new java parameters in the **Java Parameters** field:

- `-Xbootclasspath/p:PathToAutoProbeConnectorJar;PathToAgentJar`
- `-Dcom.wily.introscope.agentProfile=<path-to-IntroscopeAgent.profile>`

For example:

```
-Xbootclasspath/p:D:/usr/sap/ccms/wily/connectors/
  AutoProbeConnector.jar;D:/usr/sap/ccms/wily/Agent.jar
```

```
-Dcom.wily.introscope.agentProfile=D:/usr/sap/ccms/wily/
  IntroscopeAgent.profile
```

» **Note** For NetWeaver 6.40 on Windows, the slashes for these java parameters must be forward slashes.

4. Click **Disk** to save.
5. Repeat steps 2 - 4 for each server.
6. Restart the SAP server.
7. To verify that Configtool changes were made, open the file:
   `<drive>:\usr\sap\ccms\P66\JC00\j2ee\cluster\instance.properties`
8. Look for a line beginning with `ID<server_id>.JavaParameters`, and confirm that it contains the lines you entered.

Run AutoProbe Connector for Sun ONE

1. Log in as Administrator or Root.

   You must be logged in with Administrator or Root permissions to add Introscope information to startup scripts for Sun ONE 7.0.
2. Open the server.xml file, located at:
   `<SunONE install dir>/domains/domain1/server1/config/`

   » **Note** The item separator is a colon (:).
3. Add this line to the `server.xml` file:

   ```
   <jvm-options>
   -Xbootclasspath/p:PathToAutoProbeConnectorJar:PathToAgentJar
   </jvm-options>
   ```

   For example:

   ```
   <jvm-options>
   -Xbootclasspath/p:/sw/sun/sunone7/wily/connectors/
     AutoProbeConnector.jar:/sw/sun/sunone7/wily/Agent.jar
   </jvm-options>
   ```

Run AutoProbe Connector for Oracle 10g

To run the AutoProbe Connector, modify the bootstrap classpath:

```
-Xbootclasspath/p:wily/connectors/AutoProbeConnector.jar:PathToAgentJar
```

Run AutoProbe Connector for other application servers

To run the AutoProbe Connector, add the `Agent.jar` and the AutoProbe Connector to the Application Server bootstrap classpath using this command:

```
-Xbootclasspath/p:wily/connectors/AutoProbeConnector.jar:PathToAgentJar
```

JRockit JVM AutoProbe

To configure the JRockit JVM for AutoProbe, start the JRockit JVM using these command-line options:

- For JRockit 7.0, SP2 and previous:

```
-Djrockit.preprocessor.class=com.wily.introscope.api.weblogic.
  PreProcessor
-Xbootclasspath/a:PathToAgentJar
```

- For JRockit 7.0, SP3 and above, and for WebLogic JRockit 8.1:

```
-Xbootclasspath/a:PathToAgentJar
-Xmanagement:class=com.wily.introscope.api.jrockit.
  AutoProbeLoader
```

- For WebLogic 9.0 with JRockit 5.0:

```
JAVA_VENDOR=BeaJAVA_OPTIONS=%JAVA_OPTIONS% -javaagent:PathToAgentJar
```

JVM 1.5 AutoProbe

If you use JVM 1.5 from any vendor, configure AutoProbe to use Java 1.5 JVM by adding these options to the JVM command line:

```
-javaagent:PathToAgentJar
-Dcom.wily.introscope.agentProfile=PathToAgentProfile
```

JVM AutoProbe and OS/400

JVM AutoProbe only functions with OS/400 under certain conditions.

Java 1.4

Under Java 1.4, you can use JVM AutoProbe with OS/400 if you have the following in your environment:

- JVM versions 1.4.2 (64-bit only)
- WAS 5.1 or 6.0
- The following Program Temporary Fixes (PTFs) applied:
  - MF41376
  - MF41469
  - MF41505
  - SI27807
  - SI27807
  - SI27808
  - V5R3M0

- V5R3M5
- V5R4M0

- Specify the following JVM system properties:

```
-Xbootclasspath/p:/QIBM/ProdData/Java400/jdk14/lib/
  instrumentation.jar:Agent.jar
-agentlib:QJVAIAGENT=Agent.jar
-Dos400.jvmti.force.jitc
-Dcom.wily.introscope.agentProfile=PathToAgentProfile
```

» **Note** Substitute your own system's paths to the Java home, the agent jar and the agent profile.

It is not necessary to generate a connector jar when using JVM 1.4 AutoProbe and OS/400 in this configuration.

Java 1.5

Under Java 1.5, you can also use JVM AutoProbe with OS/400 if you have the following in your environment:

- Java 1.5 (32-bit mode)

  Configure AutoProbe to use Java 1.5 JVM by adding these options to the JVM command line:

  ```
  -javaagent:PathToAgentJar
  -Dcom.wily.introscope.agentProfile=PathToAgentProfile
  ```

- WAS 6.1

JVM AutoProbe and Apache Tomcat

You can install the Java Agent on an Apache Tomcat application server. Depending on which method you used to install the Java Agent on the Tomcat application server, there are further configurations you must make to ensure the Java Agent operates and reports metrics correctly.

If you used the Java Agent installer:

Step 1 Follow the installer instructions. See *The Java Agent installer* on page 22 for more information.

Step 2 Edit the `IntroscopeAgent.profile` as desired. There are multiple ways to configure the profile - see *Configuring connection to the Enterprise Manager* on page 36, *Configuring the Java Agent name* on page 40, *Configuring ProbeBuilder options* on page 41 for more information.

Step 3 Configure the Tomcat PBD with your tracing decisions. See *Tomcat PBD tracing options* on page 53 for more information.

Step 4 Edit the Tomcat startup script to add Wily specific code. See *Editing the startup script* on page 54 for more information.

If you manually installed the Java Agent:

Step 1 Follow the manual installation instructions. See *Manual installation* on page 29 for more information.

Step 2 Edit the `IntroscopeAgent.profile` as desired. There are multiple ways to configure the profile - see *Configuring connection to the Enterprise Manager* on page 36, *Configuring the Java Agent name* on page 40, and *Configuring ProbeBuilder options* on page 41 for more information.

Step 3 Configure the Tomcat PBD with your tracing decisions. See *Tomcat PBD tracing options* on page 53 for more information.

Step 4 If you are using application server AutoProbe, create the `AutoProbeConnector jar`. See *AutoProbe for Application Servers* on page 63 for more information.

Step 5 Copy the `WebAppSupport.jar` from the `<Agent_Home>/wily` root directory into `tomcat_root/common/endorsed` and `<Agent_Home>/wily/ext` directories.

Step 6 Edit the Tomcat startup script to add Wily specific code. See *Editing the startup script* on page 54 for more information.

Tomcat PBD tracing options

Once you have installed the Java Agent on an Apache Tomcat application server, there are some tracing options that must be configured. You must determine if:

- you want to use unformatted or formatted session tracing (see *step 2* below).
- you want to trace Apache sessions or HTTP sessions (see *step 3* below).
- you want to use unformatted or formatted DBCP tracing (see *step 4*, below).

Once you have determined the above, you configure the PBD for your APache Tomcat version.

To configure your Tomcat PBDs:

1 Open the PBD for the version of your Apache Tomcat application server from the `wily` directory. The following PBDs are available for configuration:

- `tomcat41x.pbd`
- `tomcat50x.pbd`
- `tomcat55x.pbd`

» **Note** Configure only one of the PBDs, and delete the ones you are not using.

2 In the `HTTP Session Configuration` section of the `toggles-full.pbd` or `toggles-typical.pbd`, select and uncomment the formatting option you want to use. Use one of the following:

- `FormattedSessionTracing` produces metrics that are easier to read, but may not work on all Tomcat installations.
- `UnformattedSessionTracing` produces metrics that may not be as easy to read, but this option functions in all Tomcat installations. This option is enabled by default.

» **Note** Use one of the formatting options, not both.

3 Also in the `HTTP Session Configuration` section, decide which type of session you are going to trace and do one of the following:

- If you are tracing Apache sessions, uncomment these two session options:

```
TurnOn: ApacheStandardSessionTracing
TurnOn: SuperpagesSessionTracing
```

» **Note** These session options are enabled by default.

- If you are tracing HTTP sessions, you must comment out the above session options and uncomment the following:

```
#TurnOn: HTTPSessionTracing
```

4 In the DBCP Configuration section of the PBD, determine which type of DBCP formatting you want to use and uncomment one of the following:

- `TurnOn: FormattedDBCPTracing`

  FormattedSessionTracing will produce metrics with better readability, but may not work on all Tomcat installs.

- `TurnOn: UnformattedDBCPTracing`

  This option is default setting.

5 Save the Tomcat PBD.

If the Tomcat PBD you modified is in the `hotdeploy` directory, you do not need to restart your Java Agent. If the PBD is in another directory, you must restart your Java Agent.

Editing the startup script

To ensure the Java Agent starts with the Apache Tomcat application server, you must edit the start up script to include some code from Wily. Here is an example from Tomcat 5.0:

To edit the start up script:

1 Open the `catalina.bat` file, usually located in the `<tomcat_root>/bin` directory.

2 Insert the following code into the startup script, customizing paths, etc., to suit your own location and configuration:

```
::  ----- Wily Introscope ---------------------------------------
:: Place this code right before the commented-out start command
:: Only put Wily on the classpath when starting Tomcat
if not "%ACTION%" == "start" goto skipWilyVars
set WILY_HOME=S:\sw\apache\tomcat\5.0.30\wily
```

» **Note** Comment in the section below for JDK version 1.4, or comment in the following section for JDK version 1.5, but not both versions.

```
::  NOTE: Configuration below for jdk versions <= 1.4
set WILY_ARGS=-Xbootclasspath/
  p:"%WILY_HOME%\connectors\AutoProbeConnector.jar;%WILY_HOME%\Agent.jar;
  %WILY_HOME%\WebAppSupport.jar"
::  NOTE: Configuration below for jdk versions >= 1.5
::set WILY_ARGS=-javaagent:"%WILY_HOME%\Agent.jar"
set WILY_NAME=-Dcom.wily.introscope.agent.agentName=NewTomcatAgent
set WILY_OPTS=-
  Dcom.wily.introscope.agentProfile="%WILY_HOME%\IntroscopeAgent.profile"
  %WILY_NAME%
echo Using WILY_HOME: %WILY_HOME%
echo Using WILY_ARGS: %WILY_ARGS%
echo Using WILY_NAME: %WILY_NAME%
echo Using WILY_OPTS: %WILY_OPTS%
:skipWilyVars

::Comment out the original start command
::%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
  Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
  Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
  Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS% %ACTION%

::Print the command line before executing it
echo About to execute command: %_EXECJAVA% %WILY_ARGS% %WILY_OPTS%
  %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -
  Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath "%CLASSPATH%" -
  Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
  Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS% %ACTION%

%_EXECJAVA% %WILY_ARGS% %WILY_OPTS% %JAVA_OPTS% %CATALINA_OPTS%
  %DEBUG_OPTS% -Djava.endorsed.dirs="%JAVA_ENDORSED_DIRS%" -classpath
  "%CLASSPATH%" -Dcatalina.base="%CATALINA_BASE%" -
  Dcatalina.home="%CATALINA_HOME%" -Djava.io.tmpdir="%CATALINA_TMPDIR%"
  %MAINCLASS% %CMD_LINE_ARGS% %ACTION%
```

3 Save the `catalina.bat` file.

Configuring ProbeBuilder options

By default, AutoProbe will use the typical PBD set provided with the Java Agent, which results in the collection of a moderate number of metrics. The following sections have instructions on how to customize the metric collection level, and how to configure optional ProbeBuilding behaviors.

Full or typical tracing options

In Introscope, ProbeBuilder List (PBL) files govern which tracer groups are used in the instrumentation process. The `introscope.autoprobe.directivesFile` property specifies one or more PBL files.

Introscope provides two versions of each default PBL—a *full* version which enables a larger set of Tracer Groups than the typical version which results in more detailed metric reporting, and a *typical* version that enables a smaller set of Tracer Groups, resulting in less detailed metric reporting, and as a result, reduced overhead. By default, `introscope.autoprobe.directivesFile` specifies the typical version of the default PBL file.

To change the tracing level between full and typical:

- Specify the name of the PBL file you wish to use in `introscope.autoprobe.directivesFile`.

  For example, to use the Full version of the standard PBL for WebLogic Server, set the property to:

  ```
  introscope.autoprobe.directivesFile=weblogic-full.pbl
  ```

  and restart the managed application.

  » **Note** For more information on full and typical system directives files and customizing the TYPICAL settings, see *ProbeBuilder Directives overview* on page 74.

Dynamic ProbeBuilding

Introscope uses dynamic ProbeBuilding (also called dynamic instrumentation) to implement new and changed PBDs without restarting managed applications or the agent itself. This is useful for making corrections to PBDs, or to temporarily change data collection levels during triage or diagnosis without interrupting application service.

» **Important** Dynamic instrumentation is only available for use with Java 1.5 or higher. Dynamic instrumentation is dependant on Java 1.5 capabilities, so previous versions of Java are not able to use this Introscope function.

When dynamic instrumentation is enabled, Introscope periodically checks for new and changed PBDs. To minimize overhead, Introscope selectively re-instruments classes affected by the modified PBDs. To improve performance, the scope of dynamic agent re-instrumentation is limited to reloading only those classes whose instrumentation has changed when PBDs were edited.

When a PBD edited or added to the hotdeploy directory, only user directives (such as adding or removing directives for a class, or toggling tracer groups) are re-instrumented. System directives (such as adding a tracer or changing a new tracer mapping) are not re-instrumented. Arrays, interfaces, and classes specified in Skip directives are not re-instrumented, as well as any transformations. In addition, you can exclude all classes loaded by particular classloaders from the re-instrumentation process and limit the scope of the re-instrumentation process to specific class packages.

For more information about the hotdeploy directory, see *Contents of the wily\hotdeploy directory* on page 32.

Dynamic instrumentation is not enabled by default.

If a class is re-instrumented so that it no longer reports data for a metric, the metric is still displayed in the Introscope Investigator. Existing metrics do not disappear from the Investigator window if their classes are re-instrumented.

» Important Due to a limitation in Java 1.5, access to some class bytes is not available, with the following effects:

- Modifications to the `j2ee.pbd` file may not be picked up, and metrics may continue to be published under old names.
- Some exceptions may appear in the agent log.

To avoid this issues, restart the application server after modifying the `j2ee.pbd` file.

When configuring dynamic instrumentation, CA Wily recommends that you base your changes on tracer groups. For example, if you want to control the level of instrumentation for the tracer group XYZ, you should create two tracer groups:

- XYZTracing - regular tracing options
- XYZTracingLite - fewer components are traced

Once these two tracer groups have been created, you can toggle between them, turning off XYZTracing and turning on XYZTracingLite. By toggling between the two tracer groups you can view the impact that dynamic instrumentation has on your environmental performance and adjust the tracing groups accordingly. This would affect all classes being traced as part of each tracer group.

» **Important** Changes to directives not using tracer groups are not supported. For example: changes in any directive like `TraceAllMethods` that does not have an `IfFlagged` switch are not be supported. However, Introscope does not ship any out of the box directives without tracer groups or flags.
Changes to skips or transformations are also not supported.

For more information on tracer groups, see *Default tracer groups and toggles files* on page 76, *Turning tracer groups on or off* on page 79, and *Adding classes to a tracer group* on page 79.

To configure dynamic ProbeBuilding:

1. Open the `IntroscopeAgent.profile` file, usually located in the `<Agent_Home>/wily` directory.
2. Verify that the property, `introscope.autoprobe.enable`, is set to `true`.
3. Uncomment the following properties and set values:
   - `introscope.autoprobe.dynamicinstrument.enabled=true`

     This property enables dynamic instrumentation.

     » **Note** You must restart the managed application before changes to this property take effect.

   - `introscope.autoprobe.dynamicinstrument.pollIntervalMinutes=1`

     The polling interval in minutes to check for PBD changes. The default is set to one minute intervals.

     » **Note** You must restart the managed application before changes to this property take effect.

   - `introscope.autoprobe.dynamicinstrument.classFileSizeLimitInMegs=1`

     Some classloader implementations have been observed to return huge class files.This is to prevent memory errors.

     » **Note** You must restart the managed application before changes to this property take effect.

   - `introscope.autoprobe.dynamic.limitRedefinedClassesPerBatchTo=10`

     Re-defining too many classes at a time might be very CPU intensive. In cases where the changes in PBDs trigger a re-definition of a large number of classes, this property batches the process at a comfortable rate.

     » **Important** The following properties are no longer available for use and have been removed from the `IntroscopeAgent.profile`:
     `introscope.autoprobe.dynamicinstrument.instrumentList=all, com.x.y,foo.a.b`
     `introscope.autoprobe.dynamicinstrument.avoidClassLoaders=System,com.myappserver.`

4 Save changes to the `IntroscopeAgent.profile`.

5 Restart the managed application (if appropriate).

ProbeBuilding class hierarchies (JVM 1.5)

In pre-1.5 JVMs, Introscope does not automatically instrument classes in the deeper levels of class hierarchy—only the classes that explicitly extend a probed class. For more information, see *Instrumenting and inheritance* on page 93.

On JVM 1.5, you can configure Introscope to instrument multiple levels of subclasses of a probed class—the Tracer Groups in the associated internal directive will be updated appropriately, and the classes will be dynamically instrumented. Directive changes will be written to a log file as well.

If you prefer to update your PBDs manually, you can disable directive updates and use the log file to determine appropriate updates.

Enable instrumentation of multiple levels of subclasses

Follow these steps to configure Introscope to dynamically update internal directives.

To enable instrumentation of multiple levels of subclasses:

1 Verify that dynamic instrumentation is enabled as described in *Dynamic ProbeBuilding* on page 56.

2 Open the `IntroscopeAgent.profile`.

3 To enable instrumentation of multiple levels of subclasses, uncomment this property setting:

```
introscope.autoprobe.hierarchysupport.enabled=true
```

4 Save the `IntroscopeAgent.profile`.

Configure periodic polling for uninstrumented subclasses

When multi-level subclass instrumentation is enabled, Introscope will check for uninstrumented subclasses at application startup.

To configure Introscope to poll for uninstrumented subclasses:

1 Open the `IntroscopeAgent.profile`.

2 Uncomment this property setting:

```
introscope.autoprobe.hierarchysupport.runOnceOnly=false
```

3 To change the frequency with which Introscope polls for uninstrumented subclasses from its default value of 5, uncomment this property and set it to the desired polling frequency:

```
introscope.autoprobe.hierarchysupport.pollIntervalMinutes
```

4. Optionally, you can limit the number of times Introscope polls uninstrumented subclasses by uncommenting this property and setting it to the desired limit:

```
introscope.autoprobe.hierarchysupport.executionCount
```

5. Save the `IntroscopeAgent.profile`.

Disable directive updates

If multi-level subclass instrumentation is enabled, when Introscope detects uninstrumented subclasses, by default, it updates internal directives appropriately to ensure the classes are instrumented. If you prefer to update PBDs manually, you can disable internal directive updates by uncommenting this property in the `IntroscopeAgent.profile`:

```
introscope.autoprobe.hierarchysupport.disableDirectivesChange=true
```

Controlling directive logging

When multi-level subclass instrumentation is enabled, you must uncomment the following properties in the `IntroscopeAgent.profile` to have multi-level subclass instrumentation logs created. When these properties are configured, a log file named pbdupdate.log is created in the `<Agent_Home>/wily` directory (by default), or in the custom location (if specified). The multi-level instrumentation details are written to the agent logs.

```
log4j.additivity.IntroscopeAgent.inheritance=false
log4j.logger.IntroscopeAgent.inheritance=INFO,pbdlog
log4j.appender.pbdlog.File=pbdupdate.log
log4j.appender.pbdlog=com.wily.introscope.agent.AutoNamingRollingFileAppen
  der
log4j.appender.pbdlog.layout=com.wily.org.apache.log4j.PatternLayout
log4j.appender.pbdlog.layout.ConversionPattern=%d{M/dd/yy hh:mm:ss a z} [%-
  3p] [%c] %m%n_
```

You must restart the managed application before changes to these properties take effect.

Removing line numbers in bytecode

When you instrument application bytecode, the AutoProbe or ProbeBuilder preserves the bytecode line numbers by default. Preserving bytecode line number information is helpful when using debuggers, or when obtaining stack trace information.

You can turn off this feature, (which will remove all line numbers when AutoProbe or ProbeBuilder instruments the application code), by adding a system property on the Java command line.

To remove line numbers in bytecode when using AutoProbe or ProbeBuilder:

- define the following system property on the Java command line with the `-D` option:

```
com.wily.probebuilder.removeLineNumbers=true
```

CHAPTER 4

AutoProbe for Application Servers

This chapter provides instructions for configuring AutoProbe for application servers. AutoProbe dynamically instruments all applications loaded by a specific application server.

Before you start

This chapter provides instructions for configuring AutoProbe for application servers. This is one of three ways to instrument your applications. The other two methods of instrument applications supported by CA Wily are:

- **AutoProbe for Java Virtual Machines**. For more information, see *Configuring JVM AutoProbe* on page 46. This is the recommended method of instrumenting your code.
- **Manual ProbeBuilding**. For more information, see *Manual ProbeBuilding* on page 225.

» **Important** Use only one method of instrumentation.

The instructions in this chapter assume that you have performed the following installation and configuration tasks:

- *Installing the Java Agent* on page 22
- *Configuring connection to the Enterprise Manager* on page 36
- *Configuring the Java Agent name* on page 40
- *Configuring ProbeBuilder options* on page 41

» **WARNING** Application Server AutoProbe is not supported on any JVM 1.5 and above platforms.

Configuring WebLogic Server

The following sections detail how to configure WebLogic installations to use AutoProbe to instrument applications. Use instructions for your specific version of WebLogic.

WebLogic Server 6.1

To configure WebLogic Server 6.1 to use AutoProbe:

1. Edit the Java `classpath` in the WebLogic Server startup script to include the `wily/Agent.jar` file.
2. Configure Tracer Groups to collect servlet data. For more information, see *Configuring HTTP servlet tracing* on page 70.

WebLogic Server 6.1 SP3

To configure WebLogic Server 6.1 SP3 to use AutoProbe:

1. Edit the Java `classpath` in the WebLogic Server startup script (for example, `StartPetstore.cmd`) to include the `wily/Agent.jar` file.
2. Set this property on the Java command line with the `-D` option, to activate Introscope AutoProbe:

   ```
   -Dweblogic.classloader.preprocessor=
   com.wily.introscope.api.weblogic.PreProcessor
   ```
3. Configure Tracer Groups to collect servlet data. For more information, see *Configuring HTTP servlet tracing* on page 70.

WebLogic Server 7.0

To configure WebLogic Server 7.0 to use AutoProbe:

1. Edit the Java `classpath` in the WebLogic Server startup script (`startWLS.cmd`) to include the `wily/Agent.jar` file.
2. Set this property on the Java command line (in the same startup script `startWLS.cmd)` with the `-D` option, to activate Introscope AutoProbe:

   ```
   -Dweblogic.classloader.preprocessor=
   com.wily.introscope.api.weblogic.PreProcessor
   ```
3. Configure Tracer Groups to collect servlet data. For more information, see *Configuring HTTP servlet tracing* on page 70.

WebLogic Server 8.1, 9.0, or 9.1

To configure WebLogic Server 8.1, 9.0, or 9.1 to use AutoProbe:

1. Edit the `classpath` in the application startup script (such as `startMedRecServer.cmd`) to include the `wily/Agent.jar` file.
2. Set the following property in the application startup script on the Java command line with the `-D` option to activate Introscope AutoProbe:

   ```
   -Dweblogic.classloader.preprocessor=
   com.wily.introscope.api.weblogic.PreProcessor
   ```
3. Configure Tracer Groups to collect servlet data. For more information, see *Configuring HTTP servlet tracing* on page 70.

Configuring WebSphere Application Server (WAS)

The following sections detail how to configure WebSphere Application Server (WAS) installations to use AutoProbe to instrument applications. Use the instructions for your specific version of WebSphere.

WebSphere 6.0/5.1/5.0

To configure WebSphere 6.0/5.1/5.0 to use AutoProbe:

1. Add the `Agent.jar` to the runtime extensions directory for WebSphere at `<product_installation_root>/lib/ext`.
2. In WebSphere, start the Administrator's Console and go to the **JVM Settings** section for the application server you want to modify:
   - For WebSphere 6.0, navigate to **Application Servers > *your_server* > Java and Process Management > Process Definition > Java Virtual Machine**.
   - For WebSphere 5.0, navigate to **Application Servers > your_server > Process Definition > Java Virtual Machine**.
3. Set the **Generic JVM Arguments** field to specify the classloader plug-in and the location of the `IntroscopeAgent.profile` file. You will set EITHER the `com.wily.introscope.agentProfile`,

   OR

   `com.wily.introscope.agentResource`. The argument will then have the following value (there are several properties set in one argument):

   ```
   -Dcom.ibm.websphere.classloader.plugin=com.wily.introscope.api
   .websphere.WASAutoProbe
   -Dcom.wily.introscope.agentProfile=<path to IntroscopeAgent.profile>
   ```

   OR

   ```
   -Dcom.ibm.websphere.classloader.plugin=com.wily.introscope.api
   .websphere.WASAutoProbe
   -Dcom.wily.introscope.agentResource=<path to Resource containing
     IntroscopeAgent.profile>
   ```

   » **Note** Although the examples shown break across lines, make sure that your argument does not have any breaks.
4. Apply the changes in the Administrator's Console.
5. Restart the Web Application Server.
6. Configure Tracer Groups to collect servlet data. For more informaiton, see *Configuring HTTP servlet tracing* on page 70.

Java2 Security Policy

If you have Java2 Security enabled, you may need to add permissions to your Java2 Security Policy. For more information, see *Modifying Java2 Security Policy* on page 71.

Configuring WebSphere z/OS

The following sections detail how to configure WebSphere on z/OS installations to use AutoProbe to instrument applications. Use instructions for your specific version of WebSphere on z/OS.

WebSphere 5.x and 6.0 for z/OS

To WebSphere 5.x and 6.0 for z/OS to use AutoProbe:

1. In WebSphere, start the Administrator's Console, and go to the **JVM Settings** section for the application server you want to modify.
2. Select **Application Servers >** <***your server***> **> Process Definition.**
3. You should see two items, **Control** and **Servant**. Click **Servant**, then **JavaVirtualMachine**.
4. Set the **Generic JVM Argument** field to specify the classloader plug-in, and the location of the `IntroscopeAgent.profile` file. You will set EITHER the `com.wily.introscope.agentProfile`,

   OR

   `com.wily.introscope.agentResource`. The argument will then have the following value (there are several properties set in one argument):

   ```
   -Dcom.ibm.websphere.classloader.plugin=com.wily.introscope.api
     .websphere.WASAutoProbe
   -Dcom.wily.introscope.agentProfile=<path to IntroscopeAgent.profile>
   OR
   -Dcom.ibm.websphere.classloader.plugin=com.wily.introscope.api
     .websphere.WASAutoProbe
   -Dcom.wily.introscope.agentResource=<path to Resource containing
     IntroscopeAgent.profile>
   ```
5. Place the `Agent.jar` file in the `<WebSphere Instance dir>/lib/ext` directory.

   » **Note** Do not place the `Agent.jar` file in the WebSphere installation directory.

   The following shows examples of the wrong and right directory:

   NO: `/usr/lpp/zWebSphere/V5R0M0/lib/ext`
   YES: `/WebSphere/V5R0M0/AppServer/lib/ext`

6 The default for file encoding for WebSphere 5.x and 6.0 on z/OS is now ASCII, but Introscope expects EBCDIC file format. When you configure JVM AutoProbe for z/OS, click on **Custom Properties** and add the name value pair:

```
name: com.wily.introscope.default.encoding
value: Cp1047
```

» **Note** This step is for z/OS operating systems only.

7 Confirm that all newly created Introscope files and directories within the `./wily` directory are read-accessible by the WebSphere process.

8 Confirm that all *.log files (written by the Java Agent and ProbeBuilder) in the `./wily` folder have write-access to the WebSphere process. These include:

- all the Introscope files and directories
- the Introscope files inside `<WAS instance dir>/lib/ext`

9 Restart WebSphere application server.

10 When WebSphere says "open for e-business," open the Administrator's Console. Metrics should start reporting.

11 Configure Tracer Groups to collect servlet data. For more informaiton, see *Configuring HTTP servlet tracing* on page 70.

Java2 Security Policy

If you have Java2 Security enabled, you may need to add permissions to your Java2 Security Policy. For more information, see *Modifying Java2 Security Policy* on page 71.

Configuring Sun ONE

The following sections detail how to configure Sun ONE installations to use AutoProbe to instrument applications. Use instructions for your specific version of Sun ONE.

Sun ONE 7.0

To configure Sun ONE 7.0 to use AutoProbe:

» **Note** The use of "..." in the .xml examples below indicates that there is additional information in the .xml code (not relevant to the example) that is not shown.

1 In order to add Introscope information to startup scripts for Sun ONE 7.0, you must be logged in as Administrator or Root.

2 Open the `server.xml` file, located at:

```
<Sun ONE install dir>/domains/domain1/server1/config/
```

» **Note** The item separator is a colon (:).

3. Add the full path of `wily/Agent.jar` to the "server-classpath" property of the *java-config* element in the `server.xml` file. For example:

```
<java-config ... server-classpath="/sw/sun/sunone7/wily/Agent.jar:..."
  ...>
```

4. Add the following to the *java-config* element:

   - Add the *bytecode-preprocessors* property and set it to the value `com.wily.introscope.api.sun.appserver.SunONEAutoProbe`.

     For example:

```
<java-config ... bytecode-
  preprocessors="com.wily.introscope.api.sun.appserver.SunONEAutoProbe"
  >
```

   - Add a *jvm-options* element to define the location of the agent profile. Define either `com.wily.introscope.agentProfile`, or `com.wily.introscope.agentResource`.

   The following is an example of `com.wily.introscope.agentProfile`:

```
<java-config ...>
...
<jvm-options>-Dcom.wily.introscope.agentProfile=/sw/sun/sunone7/wily/
  IntroscopeAgent.profile </jvm-options>
</java-config>
```

   The following is an example of com.wily.introscope.agentResource:

```
<java-config ...>
...
<jvm-options>-Dcom.wily.introscope.agentResource=<virtual path to>/
  IntroscopeAgent.profile</jvm-options>
</java-config>
```

◆ **OPTIONAL**: If you configured `com.wily.introscope.agentResource`, add the resource file to the server classpath.

5. Configure Tracer Groups to collect servlet data. For more informaiton, see *Configuring HTTP servlet tracing* on page 70.

Configuring Oracle 10g

The following sections detail how to configure Oracle 10g installations to use AutoProbe to instrument applications. Use instructions for your specific version of Oracle 10g.

Oracle 10g 10.0.3

To configure Oracle 10g 10.0.3 to use AutoProbe:

1. Add `Agent.jar` to the application server classpath.
2. Set the system property `oracle.classpreprocessor.classes` with the value of `com.wily.introscope.api.oracle.OracleAutoProbe`.
3. Set the system property `oracle.j2ee.class.preprocessing` with the value of `true`.
4. Run this command at the command line:

   `-Dcom.wily.introscope.probebuilder.oracle.enable=true`
5. Restart the Oracle Application Server 10g, using this command:

   `java -Doracle.classpreprocessor.classes=com.wily.introscope.api.oracle.OracleAutoProbe -Doracle.j2ee.class.preprocessing=true -Dcom.wily.introscope.probebuilder.oracle.enable=true -classpath oc4j.jar:`*`<path to wily install dir>`*`/wily/Agent.jar com.evermind.server.OC4JServer -config <path to oracle install dir>/config/server.xml`
6. Configure Tracer Groups to collect servlet data. For more informaiton, see *Configuring HTTP servlet tracing* on page 70.

Configuring HTTP servlet tracing

Before you use AutoProbe with your application servers to instrument your applications, you must configure Tracer Groups in the `toggles-full.pbd` and `toggles-typical.pbd` files. This will enable servlet data to be collected.

You will turn one Tracer Group off, and turn another Tracer Group on.

To configure HTTP servlet tracing:

1. Navigate to the `<your-application-server-home>/wily/toggles-full.pbd` file and open it.
2. Go to the `HTTP Servlets Configuration` section of the PBD.
3. Turn **off** the `HTTPServletTracing` Tracer Group by placing a pound sign at the beginning of the line. For example:

```
#TurnOn: HTTPServletTracing
```

4 Turn **on** the `HTTPAppServerAutoProbeServletTracing` Tracer Group by removing the pound sign from the beginning of the line. For example:

```
TurnOn: HTTPAppServerAutoProbeServletTracing
```

5 Repeat steps 2-4 for `<your-app-server-home>/wily/toggles-typical.pbd` file.

Modifying Java2 Security Policy

If you have Java2 Security enabled, you may need to add the following permissions to your Java2 Security Policy.

To add permissions to your Java2 Security Policy:

- Edit the file `<WebSphere home>/properties/server.policy` to include the following:

```
// permissions for Introscope AutoProbe
  grant codeBase "file:${was.install.root}/-" {
  permission java.io.FilePermission "${was.install.root}${/
  }wily${/}-", "read";
  permission java.net.SocketPermission "*", "connect,resolve";
  permission java.lang.RuntimePermission "setIO";
  permission java.lang.RuntimePermission "getClassLoader";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "accessClassInPackage.*";
  permission java.lang.RuntimePermission "accessDeclaredMembers";
  };
  grant {
  permission java.util.PropertyPermission "*", "read,write";
  };
```

» **Note** Line breaks are shown for user readability and are not needed when adding the permissions to the `server.policy` file.

CHAPTER 5

ProbeBuilder Directives

This chapter describes how to create and modify ProbeBuilder Directives.

ProbeBuilder Directives overview

ProbeBuilder Directive (PBD) files tell the Introscope ProbeBuilder how to add probes, such as timers and counters, in order to instrument an application. PBD files govern what metrics your agents report to the Introscope Enterprise Manager.

Introscope includes a set of default PBD files. You can also create custom Introscope PBD files to track any classes or methods to obtain specific information about your applications. See *Default ProbeBuilder Directive (PBD) files* on page 75, *Default ProbeBuilder List (PBL) files* on page 76, and *Creating advanced custom tracers* on page 89 for more information.

There are two kinds of files used to specify ProbeBuilder Directives:

- **ProbeBuilder Directive (PBD) files**

  A ProbeBuilder Directive (PBD) file contains directives used by ProbeBuilder to instrument your applications. This determines which metrics the agents report to the Enterprise Manager.

- **ProbeBuilder List (PBL) files**

  A ProbeBuilder List (PBL) file contains a list of multiple PBD filenames. Different PBL files can refer to the same PBD files.

If you are using Introscope AutoProbe, the relevant PBD and PBL files for your specific application server are placed in the `<ApplicationServer_Home>/wily` directory when you install the Java Agent.

Components traced by default PBDs

The default Introscope PBD files implement tracing of the following Java components:

- Oracle JDBC
- JSP Tag Libraries
- JSP IO Tag Libraries
- JSP DB Tag Libraries
- Struts
- Servlets
- Java Server Pages (JSPs)
- Enterprise JavaBeans (EJBs)
- Java Database Connectivity (JDBC)
- Network Sockets
- Remote Method Invocation (RMI)
- Extensible Markup Language (XML)
- Java Transaction API (JTA)
- Java Naming and Directory Interface (JNDI)
- Java Message Service (JMS)
- Common Object Request Broker Architecture (CORBA)
- User Datagram Protocol (UDP)
- File Systems
- Threads
- System Logs
- Thrown and Caught Exceptions (off by default)

Default ProbeBuilder Directive (PBD) files

The Java Agent has the following default PBD files:

PBD File Name	Description
errors.pbd	This file configures Error Detector by specifying what code-level events constitute serious errors. By default, only front- and back-end errors are considered serious. That is, only errors that will be manifest as a user-facing error page or that indicate a problem with a backend system (ADO.NET, Messaging, etc.).
j2ee.pbd	This file provides tracer groups for common Java 2 components. Please use either `toggles-full.pbd` or `toggles-typical.pbd` to `TurnOn` specific tracing.
java2.pbd	This file provides tracer groups for common Java 2 components. Please use either `toggles-full.pbd` or `toggles-typical.pbd` to `TurnOn` specific tracing.
jvm.pbd	This file provides directives which implement support for various Java Virtual Machines. It is intended to be used with the Introscope default files.
oraclejdbc.pbd	This file provides tracer groups for Oracle JDBC components. Comment or uncomment the `TurnOn` directives to alter the set of Oracle JDBC components that are traced.
ServletHeader Decorator.pbd	This file is used to enable the Servlet Header Decorator which is part of the integration solution with the CEM product.
sqlagent.pbd	This is the configuration file for SQL Agent instrumentation. Instrument your JDBC vendor's .zip/.jar with it, possibly updating the Connection, Statement, and ResultSet configuration to indicate your vendor-specific types. In most cases you will not need to edit this file.
sql-agent-summary-metrics-6.1.pbd	This is the configuration file for SQL Agent instrumentation summary metrics, which give high level metrics for JDBC. Always use this PBD file when using the `sqlagent-6.1.pbd` file.
struts.pbd	
summary-metrics-6.1.pbd	
taglibs.pbd	

PBD File Name	Description
toggles-full.pbd	This file provides on/off switches in the form of `TurnOn` directives for the tracing provided in other directives files. Most tracer groups are turned on. For more information about turning tracers on or off, see *Default tracer groups and toggles files* on page 76 and *Turning tracer groups on or off* on page 79.
toggles-typical.pbd	This file provides on/off switches in the form of `TurnOn` directives for the tracing provided in other directives files. Only a small section of tracer groups are turned on. For more information about turning tracers on or off, see *Default tracer groups and toggles files* on page 76 and *Turning tracer groups on or off* on page 79.

The Java Agent also installs application server-specific PBDs, which vary depending on the application server you are monitoring.

Default ProbeBuilder List (PBL) files

There are two sets of PBL files available with each agent:

PBL File Name	Description
default-full.pbl (default)	References PBD files in which most tracer groups are turned on. Introscope uses this set by default to demonstrate full Introscope functionality.
default-typical.pbl	A subset of tracer groups in the referenced PBD files are turned on. The typical set includes common settings, and is the set you can customize for a particular environment.

The Java Agent also installs application server-specific PBLs, which vary depending on the application server you are monitoring.

Tracer groups are found in PBD files, and referred to in PBL files. They cause the reporting of information about a set of classes. In PBD files, tracer group information is referred to by the term `flag`. For example, `TraceOneMethodIfFlagged` or `SetFlag` are defining tracer group information.

Default tracer groups and toggles files

A tracer group consists of a set of tracers that is applied to a set of classes. For example, there are tracer groups which report the response times and rates for all RMI classes.

You can refine the gathering of metrics on your systems by turning on or off certain tracer groups. This affects overhead usage, either increasing or decreasing it, depending on how you configure the tracer groups.

Tracer groups are modified in the `toggles-full.pbd` and the `toggles-typical.pbd` files, which are referred to by the `default-full.pbl` and `default-typical.pbl` files. This table lists the default tracer groups and their default configurations:

Name	Definition	Default full setting	Default typical setting
CorbaTracing	CORBA method invocations	on	on
EntityBeanTracing	Entity EJB method invocations	on	on
SessionBeanTracing	Session EJB method invocations	on	on
MessageDrivenBeanTracing	Message-driven EJB method invocations	on	on
J2eeConnectorTracing	J2EE connector information	on	on
JavaMailTransportTracing	Mail sending times	on	on
JDBCQueryTracing	JDBC queries	on	on
JDBCUpdateTracing	JDBC updates	on	on
JMSListenerTracing	JMS message processing times	on	on
JMSConsumerTracing	JMS message processing times	on	on
JMSPublisherTracing	JMS message broadcast times	on	on
JMSSenderTracing	JMS message broadcast times	on	on
JSPTracing	JSP service responses	on	on
RMIClientTracing	RMI client method invocations	on	on
RMIServerTracing	RMI server method invocations	on	on
HTTPServletTracing	HTTP servlet service responses	on	on
Note: If you are using Application Server AutoProbe, turn on this tracer group: `HTTPAppServerAutoProbeServletTracing`			
StrutsTracing	Execution times of actions in the Struts framework	on	on
InstanceCounts	Counts number of instances of object type identified with tracer group.	on	on
Note: Nothing will be traced until classes are identified with this tracer groups.			
FileSystemTracing	File system bytes written and read	on	off

Name	Definition	Default full setting	Default typical setting
JAXMListenerTracing	JAXM message sends	on	off
JNDITracing	JNDI lookup times	on	off
JSPDBTagsTagLibraryTracing	Jakarta DB Tags custom tag library for reading and writing from a SQL database	on	off
JSPIOTagLibraryTracing	Jakarta IO custom tag library for a variety of input and output tasks	on	off
JTACommitTracing	Commit times using JTA	on	off
EJBMethodLevelTracing	EJB activity at method level	on	off
SocketTracing	Network socket bandwidth	on	off
UDPTracing	Network socket bandwidth	on	off
ThreadTracing	Number of active threads by class	on	off
XMLSAXTracing	Time spent parsing XML document	on	off
XSLTTracing	XML transformation time	on	off
JSPTagLibraryTracing	Processing time of custom JSP tags	off	off

Generally, the default `toggles` PBD files should not be edited. However, you can refine the gathering of metrics by turning on or off certain tracer groups. Tracer groups can be modified in the `toggles` files by:

- Turning on/off tracer groups to save on system overhead
- Adding classes to a tracer group

Tracer groups report information only when turned on (uncommented) and are activated with the keyword `TurnOn`.

Setting toggles to gather additional metric information

The following toggles, when turned on, cause the collection of additional metrics, across all APIs, for Wily-provided tracer groups that are enabled. You must add these toggles to your full or typical toggle file to change the configuration.

Name	Definition	Default full setting	Default typical setting
DefaultStalledMethod Tracing	Stalled method tracing	on	on
DefaultConcurrent InvocationTracing	Concurrent invocation information	on	off
DefaultRateMetrics	Invocation rate metrics	off	

Turning tracer groups on or off

You can refine the gathering of metrics on your systems by turning on or off certain tracer groups.

To turn a tracer group on:

1. Locate the toggles-full.pbd or toggles-typical.pbd file (depending on which file type (`<appserver>-full.pbl` or `<appserver>-typical.pbl` is in use by AutoProbe or the Java Agent). These files are found within the <appserver home>/wily directory or *<Introscope_Home>*`/config/systempbd` directory.
2. Locate the tracer group to turn on, and uncomment the line by removing the pound sign from the beginning of the line. The directive in the following example is turned on, and will cause the tracing of all HTTP Servlets.

```
TurnOn: HTTPServletTracing
```

» **Note** Any uncommented (turned on) directive for a tracer group causes the tracer group to be used.

To turn a tracer group off:

- Comment the tracer group by placing a pound sign at the beginning of the line, as in the following example:

```
#TurnOn: HTTPServletTracing
```

Adding classes to a tracer group

You can turn on tracing for a particular class by adding the class to an existing tracer group. To identify a class as being part of a tracer group, use one of the Identify keywords.

For example, to add the class, `com.myCo.ejbentity.myEJB1`, to the tracer group, `EntityBeanTracing`:

```
IdentifyClassAs: com.myCo.ejbentity.myEJB1 EntityBeanTracing
```

The identify keywords are:

- IdentifyInheritedAs
- IdentifyClassAs
- IdentifyCorbaAs

For a list of identify keywords, see *Supplementary directives and tracers information* on page 96.

EJB subclass tracing

By default, entity and session EJB-related directives add probes only for EJBs that directly and explicitly implement the entity, session, or message-driven EJB interfaces.

Often, an application's EJBs are subclasses of classes which directly and explicitly implement the entity or session EJB interface. These are not tracked by default by Introscope.

For EJB subclasses to be tracked by Introscope, they must be added to the appropriate tracer group. To do this, add entries that refer to the direct ancestors of the EJB subclasses to be tracked.

From these models, replace `<entity.bean.ancestor.class>` or `<session.bean.ancestor.class>` with the fully-qualified class name of the immediate ancestor of the EJBs to be instrumented.

For entity EJBs:

```
IdentifyInheritedAs: <entity.bean.ancestor.class> EntityBeanTracing
```

For session EJBs:

```
IdentifyInheritedAs: <session.bean.ancestor.class> SessionBeanTracing
```

The examples below are based on this class hierarchy:

```
mySessionEJB implements javax.ejb.SessionBean
  mySessionEJBsubclass1 extends mySessionEJB
    mySessionEJBsubclass1a extends  mySessionEJBsubclass1
    mySessionEJBsubclass1b extends  mySessionEJBsubclass1
  mySessionEJBsubclass2 extends mySessionEJB
```

The tracer group, `SessionBeanTracing`, causes the tracking of `mySessionEJB`:

The following tracer traces `mySessionEJBsubclass1` and `mySessionEJBsubclass2`.

```
IdentifyInheritedAs: mySessionEJB SessionBeanTracing
```

The following tracer traces `mySessionEJBsubclass1a` and `mySessionEJBsubclass1b`.

```
IdentifyInheritedAs: mySessionEJBsubclass1 SessionBeanTracing
```

» **Note** This example does not use packages. If your code is in a package, it needs to include the package name with the class name. See *Supplementary directives and tracers information* on page 96 for more information.

EJB 3.0 annotations

The follow directive allows you to group any class containing the given class-level annotation into tracer groups. This directive supports EJB 3.0. EJBs conforming to the 3.0 specifications do not explicitly implement any well-known interface, but instead are entirely enabled via annotations. To easily identify EJB 3.0 classes, use this directive:

```
IdentifyAnnotatedClass <annotation-name> <flag-name>
```

To use this directive, create a directive class and directive parser class for the new directive. You must then add a matcher class to examine your bytecode to determine if a class contains a given annotation.

» **Note** This directive does not support method-level annotations.

Applying ProbeBuilder Directives

The way in which you apply PBDs depends on the method you choose to use. CA Wily recommends you use AutoProbe to implement your PBDs. You can also use the ProbeBuilder Wizard, or the command line ProbeBuilder to implement your PBDs.

Using AutoProbe

When you are ready to implement a PBD file, add it to the `hotdeploy` directory. AutoProbe looks for PBD files in the directory that contains the `IntroscopeAgent.profile` file (by default, this is the `<Agent_Home>/wily` directory), and the `<Agent_Home>/wily/hotdeploy` directory. AutoProbe resolves filenames relative to these directories. If you have moved the location of your `wily` directory, be sure to map the file path to the correct directory.

To implement PBDs using AutoProbe:

1. Save modified standard PBD or PBLs to the `<Agent_Home>/wily` directory.
2. Copy custom PBDs into the `<Agent_Home>/wily/hotdeploy` directory. Any PBDs added to this directory will be implemented without having to update or modify the `introscope.autoprobe.directivesFile` property in the `IntroscopeAgent.profile`.

   » **Note** If you have enabled dynamic instrumentation, the PBDs in the `hotdeploy` directory are picked up live from the folder—no reboot is required. For more information about dynamic instrumentation, see *Dynamic ProbeBuilding* on page 56.

3. Save the `IntroscopeAgent.profile`.
4. Restart the application.

Using the ProbeBuilder Wizard or command-line ProbeBuilder

When you are ready to implement a PBD file, add it to the `hotdeploy` directory. The Command-line ProbeBuilder looks for any custom directive files in the same directory where ProbeBuilder is run from, and the `<Agent_Home>/wily/hotdeploy` directory. The Command-line ProbeBuilder resolves filenames relative to these directories.

The steps to implement ProbeBuilder Directives using the ProbeBuilder Wizard or command-line ProbeBuilder are the same as using AutoProbe. See *Using AutoProbe* on page 82 for more information.

Instrumenting with new and changed PBDs

For new or changed directives to take effect, your applications must be instrumented using the latest PBDs. This process varies depending on the ProbeBuilding method you use.

Using AutoProbe on JVM 1.5 systems

You can configure dynamic instrumentation, allowing changed PBDs to take effect without application or Java Agent restart. This enables you to perform PBD corrections, or perform triage-driven instrumentation without interrupting application service. For more information see *Dynamic ProbeBuilding* on page 56.

Using AutoProbe on Pre-JVM 1.5 systems

New and changed ProbeBuilder Directive files or ProbeBuilder List files take effect the next time the application server loads the application classes.

The Java Agent automatically detects when changes have been made to directives specified by the `introscope.autoprobe.directivesFile` property and reloads the directives. As application classes are reloaded by the application server, they are re-instrumented in accordance with the latest ProbeBuilder Directives.

If your managed applications are not running when you add or change directives, when you next start the applications, they will be instrumented using the updated directives.

If your managed applications are running, you do not need to restart the application server to apply the new or changed directives, but it is necessary to load, or reload, the managed application classes.

How you cause the classes to reload depends upon the application server you use. For instance, on SAP NetWeaver 6.40, a redeploy is sufficient. Other environments may require a production redeploy.

Using the ProbeBuilder Wizard

1. The Custom Directives screen will list the PBD files you placed in the `hotdeploy` directory described in *Using the ProbeBuilder Wizard or command-line ProbeBuilder* on page 82.
2. Select the custom directives files to use. For more information on running ProbeBuilder Wizard, see *Using the ProbeBuilder wizard* on page 227.

Using the command-line ProbeBuilder

» **Important** CA Wily recommends using the command-line ProbeBuilder as your last option for Introscope-enabling your latest PBDs.

1. Stop your managed application.
2. Run the command-line ProbeBuilder or the ProbeBuilder Wizard, supplying the custom PBD and PBL files in the command line. For more information on the command-line ProbeBuilder, see *Using the command-line ProbeBuilder* on page 229).
3. Start the managed application.
4. If they are not already running, start the Enterprise Manager and the Workstation.

Creating custom tracers

You can further refine your metric collection by creating custom PBD files. Creating custom directives, by creating tracers to track application specific measurements, require the use of specific syntax and keywords. To write custom tracers, you must define:

- The directive type (indicating generically how many class(es) or method(s) to trace)
- The specific class(es) or method(s) to trace
- The type of information to trace in the class(es) or method(s) (for example, a time, a rate, or a count)
- The fully-qualified metric name (including the resource path) under which to present this information

Custom PBDs are stored in the `<Agent_Home>/wily/hotdeploy` directory. Any PBDs added to this directory will be implemented without having to update or modify the `introscope.autoprobe.directivesFile` property in the `IntroscopeAgent.profile`. If you have enabled dynamic instrumentation, the PBDs in the `hotdeploy` directory are picked up live from the folder — no reboot is required. For more information about dynamic instrumentation, see *Dynamic ProbeBuilding* on page 56.

Once a custom PBD is created, Introscope treats it as if it was an out-of-the-box PBD. You can set alerts on the metrics created, save them to SmartStor, or use them in the creation of custom dashboards in the Introscope Workstation.

» **Note** Be sure to choose methods to trace carefully, as more methods traced means more overhead.

Common custom tracer example

A `BlamePointTracer` is the most commonly used tracer. This tracer generates five separate metrics for associated methods or classes:

- Average Response Time (ms)
- Concurrent Invocations
- Errors Per Interval
- Responses Per Interval
- Stall Count

The following is an example of a `BlamePointTracer`. A `BlamePointTracer` has been set for a method called `search` in class `petshop.catalog.Catalog`. `PetShop|Catalog|search` is the name of the node under which the BlamePoint metrics will be displayed in the Introscope Investigator.

```
TraceOneMethodOfClass: petshop.catalog.Catalog search BlamePointTracer
  "PetShop|Catalog|search"
```

Tracer syntax

In addition to simple keywords that associate tracers into groups or enable/disable groups, PBD files contain tracer definitions. For Introscope to recognize and process your tracers, you must use a specific syntax when constructing custom tracers. A tracer is composed of a directive and information about the method or class to trace, in the following format:

```
<directive>: [arguments]
```

where `[arguments]` is a list, and is directive-specific. Arguments used in trace directives include `<Tracer-Group>`, `<class>`, `<method>`, `<Tracer-name>`, and `<metric-name>`.

» **Note** Depending on the directive used, only a subset of these parameters are required.

Tracer arguments	Definition
`<directive>`	There are six main directives available for custom tracing: ■ **TraceOneMethodOfClass**—traces a specified method in the specified class. ■ **TraceAllMethodsOfClass**—traces all methods in the specified class. ■ **TraceOneMethodIfInherits**—traces one method in all direct subclasses or direct interface implementations of the specified class or interface. ■ **TraceAllMethodsIfInherits**—traces all methods in all direct subclasses or direct interface implementations of the specified class or interface. **Note:** Only concrete, implemented methods can be traced and report metric data while running. An abstract method specified in a custom tracer results in no metric data being reported. ■ **TraceOneMethodIfFlagged**—traces one method if the specified class is included in a tracer group that has been enabled with the `TurnOn` keyword. ■ **TraceAllMethodsIfFlagged**—traces all methods if the specified class is included in a tracer group that has been enabled with the `TurnOn` keyword.
`<Tracer-Group>`	The group to which the tracer is associated.
`<class>`	A fully qualified class or interface name to trace. Fully qualified classes include the full assembly name of the class as well as the name, for example: `[MyAssembly]com.mycompany.myassembly.MyClass` **Note:** The assembly name must be enclosed in [] brackets.
`<method>`	The method name (e.g. `MyMethod`) OR the full method signature with return type and parameters (for example, `myMethod;[mscorlib]System.Void([mscorlib]System.Int32)`. For more information on method signatures, see *Signature differentiation* on page 89.)

Tracer arguments	Definition
`<Tracer-name>`	Specifies the tracer type to be used. For example, `BlamePointTracer`. See the *Tracer name* table below for descriptions of tracer names.
`<metric-name>`	Controls how the collected data is displayed in the Introscope Workstation. The following examples describe three ways to specify the name and location of a metric at different levels of the metrics tree. ■ **metric-name**—the metric appears immediately inside the agent node. ■ **resource:metric-name**—the metric appears inside one resource (folder) below the agent node. ■ **resource|sub-resource|sub-sub-resource:metric-name**—the metric appears more than one resource (folder) level deep below the agent node. Use pipe characters (|) to separate the resources.

This table describes tracer names and what they trace:

Tracer name	What it traces
`BlamePointTracer`	Provides a standard set of metrics including average response time, per interval counts, concurrency, stalls, and errors for a blamed component.
`ConcurrentInvocationCounter`	Reports the number of times a method has started but not yet finished. The result is reported under the metric name specified in the tracer, `<metric-name>`, in the Investigator tree. An example use of this tracer would be counting the number of simultaneous database queries.
`DumpStackTraceTracer`	Dumps a stack trace to the instrumented application's standard error for methods to which it is applied. The exception stack trace thrown by the Dump Stack Tracer is not a true exception—it is a mechanism for printing the method stack trace. This feature is useful for determining callpaths to a method. » **WARNING** This feature imposes heavy system overhead. It is strongly recommended that this tracer only be used in a diagnostic context where a sharp increase in overhead is acceptable.

Tracer name	What it traces
`MethodCPUTimer`	Average CPU time (in milliseconds) used during method execution and reports it under `<metricname>` in the metrics tree. **Note:** This tracer is supported on any operating system that supports platform monitoring.
`MethodTimer`	Average method execution time in milliseconds and reports it under the metric name specified in the tracer, `<metric-name>`, in the metrics tree.
`PerIntervalCounter`	Number of invocations per interval. This interval will change based on the view period of the consumer of the data (for example, the View pane in the Investigator). It is reported under the metric name specified in the tracer, `<metric-name>`, in the Investigator tree.

Custom method tracer examples

The following are examples of method tracers. In the following example, quotes ("") are used around the metric names because there are spaces in the metric names.

Average tracer example

This tracer tracks the average execution time of the given method in milliseconds.

```
TraceOneMethodOfClass: com.sun.petstore.catalog.Catalog search
  BlamedMethodTimer "Petstore|Catalog|search:Average Method Invocation
  Time (ms)"
```

Rate tracer example

This tracer counts the number of times the method is called per second, and reports this rate under the specified metric name.

```
TraceOneMethodOfClass: com.sun.petstore.catalog.Catalog search
  BlamedMethodRateTracer "Petstore|Catalog|search:Method Invocations Per
  Second"
```

Per interval counter tracer example

This method tracer counts the number of times the method is called per interval, and reports the per interval count under the specified metric name.

```
TraceOneMethodOfClass: com.sun.petstore.catalog.Catalog search
  PerIntervalCounter "Petstore|Catalog|search:Method Invocations Per
  Interval"
```

The interval is determined by the monitoring logic in the Enterprise Manager, such as the Graph frequency.

The preview pane in the Investigator defaults to 15 second intervals.

Counter tracer example

This tracer counts the total number of times the method is called.

```
TraceOneMethodOfClass: com.sun.petstore.cart.ShoppingCart placeOrder
  BlamedMethodTraceIncrementor "Petstore|ShoppingCart|placeOrder:Total
  Order Count"
```

Combined counter tracers example

These tracers combine incrementor and decrementor Tracers to keep a running count.

```
TraceOneMethodOfClass: com.sun.petstore.account.LoginEJB login
  MethodTraceIncrementor "Petstore|Account:Logged In Users"
TraceOneMethodOfClass: com.sun.petstore.account.LogoutEJB logout
  MethodTraceDecrementor "Petstore|Account:Logged In Users"
```

Creating advanced custom tracers

The following sections detail creating advanced customs tracers, such as single-metric tracers, skips, and combined custom tracers.

Advanced single-metric tracers

Directives and tracers track methods, classes, and sets of classes. A single-metric tracer reports a specific metric for a specific method, which is the smallest unit that Introscope can track. Single-metric tracers can be created in several ways: through the method signature, by substituting keywords, or by manipulating the metric name parameters.

Signature differentiation

Tracers can be applied to a method based on the method signature.

To trace a single instance of a method with a specific signature, append the signature to the method name (including return type) specified using the internal method descriptor format.

For example, `myMethod(Ljava/lang/String;)V` traces the instance of the method with a string argument and void return type.

For complete information about this format, see the Sun Java Virtual Machine Specification, section 4.3.3, Method Descriptors, and section 4.3.2, Field Descriptors: *http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html*.

Metric name keyword-based substitution

Keyword-based substitution allows runtime substitution of values into the metric name.

The parameters in the metric name in the tracer are substituted at runtime for the actual values into the metric name. This feature can be used with any directive.

Parameter	Runtime substitution
{method}	Name of the method being traced
{classname}	Runtime class name of the class being traced
{packagename}	Runtime package name of the class being traced
{packageandclassname}	Runtime package and class name of the class being traced

» **Note** If Introscope processes a class which does not have a package, it will replace {packagename} with the string "<Unnamed Package>".

Keyword-based substitution: Example 1

If the metric name for a tracer in the pbd file is:

```
"{packagename}|{classname}|{method}:Response Time (ms)"
```

and the tracer is applied to method myMethod with a runtime class of myClass that is in package myPackage, the resulting metric name would be:

```
"myPackage|myClass|myMethod:Response Time (ms)"
```

Keyword-based substitution: Example 2

If a tracer with a metric name in the .pbd file of

```
"{packageandclassname}|{method}:Response Time (ms)"
```

was applied to the same method, the resulting metric name would be

```
"myPackage.myClass|myMethod:Response Time(ms)"
```

» **Note** Note the . between the package and class instead of the | in the first example.

Metric-name-based parameters

You can create a single-method tracer that creates a metric name based on parameters passed to a method using the `TraceOneMethodWithParametersOfClass` keyword, using this format:

`TraceOneMethodWithParametersOfClass: <class-name> <method> <tracer-name> <metric-name>`

Parameters can be used in the metric name. This is accomplished by substituting the value of parameters for placeholder strings in the metric name. The placeholder strings to use are "{#}" where # is the index of the parameter to substitute. The indices start counting at zero. Any number of parameter substitutions can be used in any order. All parameters are converted to strings before substitution into the metric name. Object parameters other than strings should be used with caution because they are converted using the toString() method.

» **WARNING** If you are unclear about what string the parameter will be converted to, do not use it in the metric name.

Metric-name-based example

A Web site uses a class named order, with a method named process. The method has parameters for different kinds of orders, either book or music.

You can create a tracer like this:

```
TraceOneMethodWithParametersOfClass: order process(LJava/lang/string;)V
  MethodTimer "Order|{0}Order:Average Response Time (ms)"
```

This tracer produces metrics like these:

Order
  BookOrder
    Average Response Time (ms)
  MusicOrder
    Average Response Time (ms)

You can also use the TraceOneMethodWithParametersIfInherits keyword. For more information on both keywords, see *Supplementary directives and tracers information* on page 96.

Skip directives

Certain packages, classes, or methods can be skipped by AutoProbe or ProbeBuilder by using skip directives. By default, the Java Agent and fundamental Java classes and packages are skipped by AutoProbe or ProbeBuilder. For more information, see *Supplementary directives and tracers information* on page 96.

For a complete list of skip directives used with the Java Agent, see the *Directive & Tracer Type Definitions* guide. See *Supplementary directives and tracers information* on page 96 for more information about this guide.

Counting object instances

The InstanceCounts tracer group counts the number of instances of the particular object types associated with it (for information on associating object types with the InstanceCounts tracer group using the standard IdentifyClassAs and IdentifyInheritedAs directives, see *Adding classes to a tracer group* on page 79). Any instances explicitly allocated in your code will be counted. Subtypes will also be counted. Objects created through different mechanisms, such as deserialization or cloning, might not be counted. Tracing using this tracer group could potentially incur incremental performance (and memory) impact, depending on the number of instances counted.

Turning on InstrumentPoint directives

There are two types of directives identified by the keyword, `InstrumentPoint`: those that trace exceptions, and one that causes agent initialization when the application starts up (instead of when the first Probe is run).

Exceptions

The following directives are used to turn on tracing of exceptions either where thrown or caught. They can cause performance degradation so they are not turned on by default. To turn either of these on, uncomment the appropriate line:

```
#InstrumentPoint: ThrowException
#InstrumentPoint: CatchException
```

Agent initialization

The agent initialization instrument point directive does not cause additional overhead and is turned on by default in both full and typical PBD sets.

```
#InstrumentPoint: AgentInitialization
```

If multiple ProbeBuilder Directive files are used, any settings (such as tracer groups, Skips, InstrumentPoints, Custom Method Tracers) turned on in any file take effect.

Combining custom tracers

You can use multiple tracers that affect the same metric, in effect combining them. This is most commonly used with incrementors and decrementors.

This example creates a metric named `Logged-in Users`. With a class `user` and methods `login` and `logout`, create the following tracers:

```
TraceOneMethodOfClass user login MethodTraceIncrementor "Logged-in Users"
TraceOneMethodOfClass user logout MethodTraceDecrementor "Logged-in Users"
```

This increments the metric Logged-in Users when someone logs in and decrements Logged-in Users when someone logs out.

Instrumenting and inheritance

Introscope does not automatically instrument classes in the deeper levels of a class hierarchy in pre-1.5 JVMs.

When subclasses of a probed class more than one level deep are loaded, the new and overridden methods are not automatically instrumented. Likewise, classes that do not explicitly name a probed interface as being implemented, even though they implement the interface indirectly, will not be instrumented either.

For example, assume a class hierarchy in which `ClassB` extends `ClassA`, and `ClassC` extends `ClassB`, like so:

```
Interface/ClassA
  ClassB
    ClassC
```

When you instrument `ClassA`, `ClassB` is also instrumented because it explicitly extends `ClassA`. However, Introscope does *not* instrument `ClassC` because `ClassC` does not explicitly extend `ClassA`. To instrument `ClassC` you must explicitly identify `ClassC`.

In pre-1.5 Java environments, to ensure that subclasses are instrumented, follow the instructions in *EJB subclass tracing* on page 80.

If you run under JVM 1.5, you can configure Introscope to instrument multiple levels of subclasses of a probed class. For instructions, see *ProbeBuilding class hierarchies (JVM 1.5)* on page 59.

If you wish to instrument a private method, you may need to use more specific tracers. For more information about directives and tracers for custom PBDs, please see the *Directive & Tracer Type Definitions* document on the Wily community site.

Java 1.5 annotations

Introscope 8.0 allows the use of Java 1.5 annotations when creating custom metrics. For more information on Java 1.5 annotations, see the following articles:

- *http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html*
- *http://www.developer.com/java/other/article.php/3556176*

Use `IdentifyAnnotatedClassAs` to place the class in a tracer group, then use `TraceXYZIfFlagged` directives to instrument the methods in the class. For example:

```
SetFlag: AnnotationTracing TurnOn: AnnotationTracing
IdentifyAnnotatedClassAs: com.test.MyAnnotation AnnotationTracing
TraceAllMethodsIfFlagged: AnnotationTracing BlamePointTracer
  "Target|MyTarget|{classname}"
```

In the example, `com.test.MyAnnotation` is the annotation name. When creating your own annotations, use a term in your code. Classes containing the annotation name are identified.

Using Blame Tracers to mark blame points

Introscope's Blame Technology works in a managed Java Application to enable you to view metrics at the application tiers: the front and backends of your application. This capability, referred to as boundary blame, allows users to triage problems to the application frontend or backend.

For information about how Introscope determines frontends and backends, and about options for configuring URL Groups to control how metrics for frontends are aggregated, see *Configuring Boundary Blame* on page 133.

The following sections describe how you can use tracers to explicitly mark the frontends and backends in your application.

Blame Tracers

Introscope provides tracers for capturing front and backend metrics: FrontendMarker and BackendMarker. These tracers explicitly mark a frontend and backend, respectively.

You can use FrontendMarker and BackendMarker to instrument your own code, for instance code that accesses a backend, to cause Introscope to capture and present metrics for custom components in the Investigator tree.

If no FrontendMarker is configured, the first component in the blame stack will be the default frontend. In some environments this may not be desirable. For example, if your environment includes Introscope Browser Response Time Adapter (BRTA), by default the browser component will appear as the default frontend. In this case, you might configure servlets as frontends.

If no BackendMarker is configured, Introscope will infer a backend—any component that opens a client socket will be a default backend if none is explicitly marked.

It is useful to use BackendMarker:

- to assign a desired name to an item that Introscope detects as a backend.
- to mark custom Java sockets that Introscope does not instrument.
- for native sockets that are called through the Java Native Interface (JNI), to identify a Java/JNI bridging method as the backend.

FrontendMarker and BackendMarker are instances of BlamePointTracer which provides metrics such as average response time, per interval counts, concurrency, stalls, and errors for a blamed component. A BlamePointTracer can be applied to middle components for a more granular Blame Stack.

Blame Tracers in standard PBDs

Two of the standard PBDs provided with Introscope, `j2ee.pbd` and `sqlagent.pbd`, implement Boundary Blame Tracing.

- `HttpServletTracer` in `j2ee.pbd` is an instance of FrontendMarker.
- `SQLBackendMarker` in `sqlagent.pbd` is an instance of BackendMarker.

The following Blame Tracers used in previous versions of Introscope still exist, but are not typically used in Introscope PBDs:

- BlamedMethodTimer
- BlamedMethodRateTracer
- BlamedMethodTraceIncrementor
- BlamedMethodTraceDecrementor

Boundary Blame and Oracle backends

In the current version of Introscope, Oracle databases are not detected based on the socket connection—SQL Agent must be available for Introscope to automatically detect Oracle backends.

To enable Introscope to detect Oracle backends in the absence of SQL Agent, make the following modification to `oraclejdbc.pbd`:

In this portion of `oraclejdbc.pbd`:

```
#Socket data from the Oracle driver reports too many metrics
SkipPackagePrefixForFlag: oracle.jdbc. SocketTracing
SkipPackagePrefixForFlag: oracle.net. SocketTracing
```

comment out the skips, as shown below:

```
#Socket data from the Oracle driver reports too many metrics
#SkipPackagePrefixForFlag: oracle.jdbc. SocketTracing
#SkipPackagePrefixForFlag: oracle.net. SocketTracing
```

Supplementary directives and tracers information

For a complete list of the tracers and directives used with the Introscope Java Agent, see the *Directive & Tracer Type Definitions* guide, available on the Wily Technology Community site, here: https://community.wilytech.com/kbclick.jspa?categoryID=414&externalID=1927

To access the Wily Technology Community site, you first need to register for an account using your corporate email address, here: https://community.wilytech.com/account!default.jspa

Once you have completed the account information, CA Wily will contact you within 3-5 business days to confirm your registration. If you do not register with a corporate email address, your request for access will be denied.

SECTION II

Java Agent Operations and Management

The chapters in this section have information about Java Agent administration and operations.

CHAPTER 6

Java Agent Naming

This chapter has information about agent naming, related environmental and deployment considerations, and options for automatically naming your agents.

Understanding the Java Agent name

This section explains alternatives for assigning a name to the Java Agents in your Introscope environment.

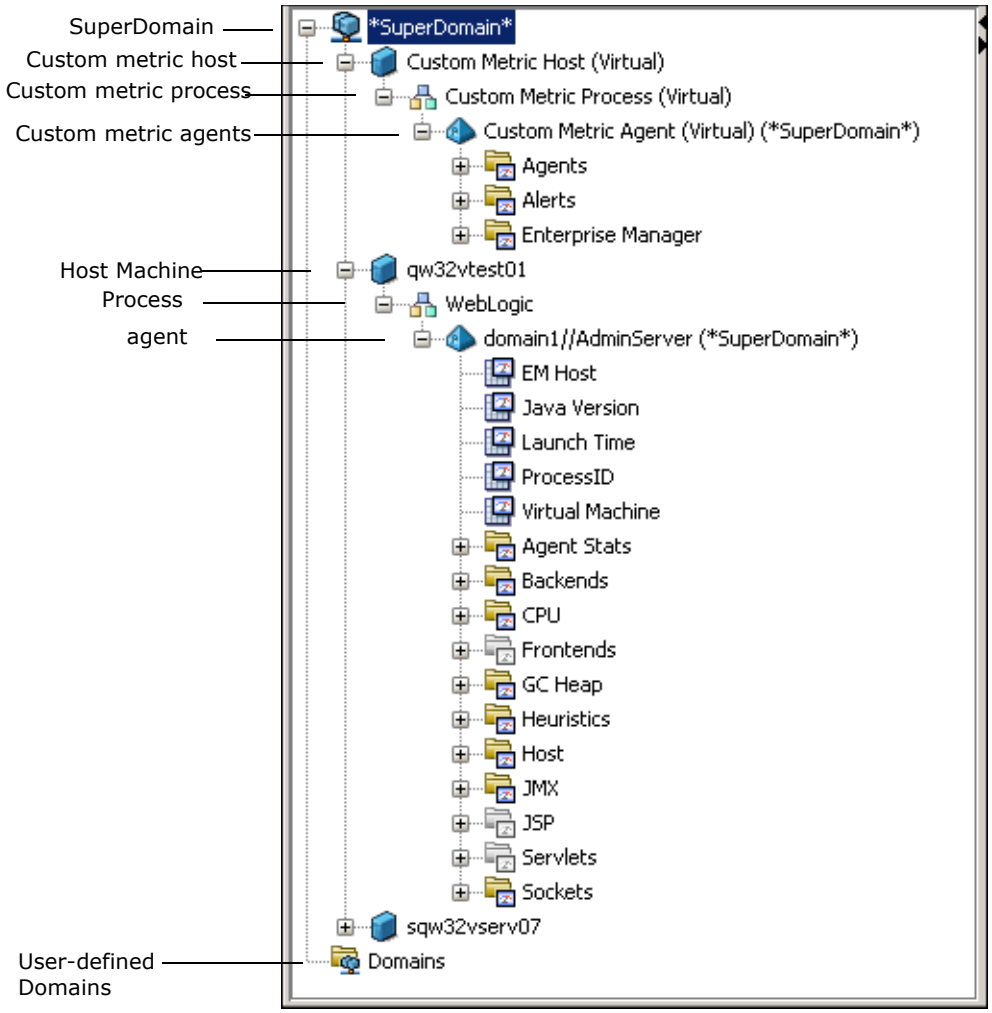
Aspects of the Java Agent name

Each Java Agent running in your Introscope environment has a name, whether you assigned one explicitly, configured a method of automatically assigning a name, or simply started up an instrumented application that the Java Agent monitors.

The Java Agent name is important because it is central to many views and presentations in the Introscope clients, and it is key to the process of associating monitoring logic with target applications.

The Investigator tree in the Workstation and WebView has a node for an Java Agent when your instrumented application is up and has started reporting metrics to the agent. The Investigator tree on the following page shows agents named `domain1//Adminserver`, running on host `qw32vtest01` under the WebLogic process.

When you configure management logic in the Workstation—for instance, Dashboards, Alerts, and Actions—the agent name is a component in the regular expressions you define that identify the applications to which the management logic applies.



How the agent determines its name

The Java Agent uses the following sequence to determine its name. If it finds a name using the first method, it accepts that name and connects to the Enterprise Manager. If it doesn't find a name using the first method, it tries the second method, and so on. If it doesn't find a name using any method, it calls itself "Unknown Agent."

Step 1 Name specified in Java system property

The agent name is defined using a Java system property on the command line. Using this method will override any other agent naming method. See *Specifying an agent name using a Java system property* on page 104.

Step 2 Name specified in System Property Key in the `IntroscopeAgent.profile`

The agent name is obtained from a Java system property specified in a property in the `IntroscopeAgent.profile`. See *Specifying an agent name using a system property key* on page 104.

Step 3 Name obtained automatically from the Application Server

If you use certain versions of WebLogic or WebSphere, the agent name can be automatically obtained from the application server using automatic agent naming functionality. You can configure a time delay, to give the agent as much time as necessary to determine its name before connecting to the Enterprise Manager. See *Obtaining an agent name from the application server* on page 104.

Step 4 Name specified explicitly in agent profile

The agent name is defined in the `IntroscopeAgent.profile`, in the property `introscope.agent.agentName`. This was the standard method for naming agents in early Introscope versions. Use this option if you already have an agent profile for every application. For more information, see *Configuring the Java Agent name* on page 40.

How Introscope resolves agent naming conflicts

The fully qualified agent name—comprised of host name, process name and agent name—is typically unique to each agent in an Introscope environment. Agents with the same agent name usually have a unique *fully-qualified* agent name because their host name and process names are likely to be different. Multiple agents will have the same fully-qualified agent name only if they reside on the same host, monitor the same process, and have the same agent name.

If an agent tries to connect to an Enterprise Manager to which an agent with the same fully-qualified agent name is already connected, the Enterprise Manager appends a unique identifier to the name of the newly connecting agent. The identifier consists of a percent (%) character and a digit. This mechanism ensures that multiple agents that connect using the same fully-qualified name can be uniquely identified for the duration of the connection. The Enterprise Manager renames the first duplicate agent to connect by appending "%1" to its agent name.

For instance, assume that two agents with the fully qualified agent name:

```
hostPA|processNIM|PodAgent
```

connect to the Enterprise Manager, one after the other. The Enterprise Manager renames the second agent:

```
PodAgent%1
```

If other agents with the same fully qualified name connect, they are renamed, in succession, `PodAgent%2`, `PodAgent%3`, `PodAgent%4`, and so on, where the digit following the percent character is the next number in sequence.

When a renamed agent disconnects, the suffix it was assigned can be re-used. For example, if `PodAgent%1` disconnects while `PodAgent` remains connected, the next agent with the fully qualified name `hostPA|processNIM|PodAgent` to connect will be renamed `PodAgent%1`.

Reuse of suffix identifier makes it possible that the Enterprise Manager might assign the same suffix to a particular agent's name from connection to connection. However, on subsequent connections, a given agent could just as well be renamed differently. Having an agent's name vary from connection to connection is problematic when querying historical data—it is preferable to configure a naming strategy that avoids the Enterprise Manager renaming agents.

Agent naming considerations for clustered applications

If you run multiple instances of the same application, Introscope attempts to resolve identical agent names, *including* custom metric agents, by appending the agent name with a character and a random number. Wily recommends, however, that you tell Introscope how to resolve the naming.

The options for resolving identical agent naming are:

- Tell Introscope that the agents in question are cloned agents by enabling cloned agent naming (described in *Enabling cloned agent naming in clustered environments* on page 109.)
- Define unique agent names yourself and make separate agent profiles for each agent (described in *Configuring unique names for application instances* on page 110.)
- Let Introscope uniquely name each agent using its own naming scheme (described in *How Introscope resolves agent naming conflicts* on page 102.)

Specifying an agent name using a Java system property

To specify an agent name using Java system property:

- On the Java command line, supply the desired name using this property:

  `com.wily.introscope.agent.`*agentName*

Specifying an agent name using a system property key

This method is the second the agent uses to look for its name. Use this method if you want the agent to be named from the value of an existing Java system property in your deployment.

To specify an agent name using the System Property Key:

1. Open the `IntroscopeAgent.profile`.
2. Under the *Agent Name* section, specify the Java system property that will provide the agent name in this property:

   ```
   introscope.agent.agentNameSystemPropertyKey
   ```

   » **Note** If the Java system property specified here doesn't exist, this property will be ignored.

3. Restart the application server.

Obtaining an agent name from the application server

You can configure the agent to extract the application server instance name automatically from the application server, and use that information to name itself. This eliminates the need to configure individual agent names in a separate agent profile file. The agent can also rename itself if there are changes in the application server environment. This enables you to deploy an agent profile across a large number of environments that might consist of a mix of application server platforms.

Application servers that support agent naming

Automatic agent Naming is supported when you use Introscope with these supported application server versions:

- WebLogic 6.1
- WebLogic 7.0
- WebLogic 8.1
- WebLogic 9.x
- WebLogic 10.0
- WebSphere 6.0.x distributed
- WebSphere 6.1.x distributed
- WebSphere 5.0.x distributed
- WebSphere 5.1 distributed
- Jboss 4.0.x
- Jboss4.2x

The name of the application server displayed in the Introscope Workstation is determined by a Java J2EE API. This sometimes causes the name of the application servers to display differently in the Workstation because all application servers implement the API differently. The names of multiple application servers may be formatted differently in the Workstation, and even the same application server name may be formatted differently from release to release.

How automatic agent naming works

When automatic agent naming is enabled, the agent starts, and looks for name information from the application server. The agent waits until an agent name is obtained before attempting to connect to the Enterprise Manager.

When the agent locates naming information, Introscope edits the information to make the agent name compliant with Introscope agent naming rules.

Agent names on supported application servers are comprised of several pieces of information, which differ according to application server.

- For WebLogic, the agent name is comprised of:

  Domain (data center) + cluster + instance (of WLS)
- For WebSphere, the agent name is comprised of:

  cell (domain) + process (instance of WAS)

When information is obtained, segments are separated by forward slashes—for example:

```
medrec/MyCluster/MedRecServer
```

Any forward slashes in the segment name are converted to underscores. For example, if a Domain is named Petstore/West, it will be converted to Petstore_West.

» **Note** When constructing the agent name that appears in Introscope, Introscope edits the information to make the agent name compliant with Introscope agent naming rules:

- characters such as pipes, colons, or percentage signs are replaced by underscores
- names that begin with any character other than a letter will have the letter "A" prepended to them
- empty names are replaced by "UnnamedAgent" (so as to be distinguishable from the "UnknownAgent" condition)

Automatic agent naming and renamed agents

Using automatic agent naming, the agent always tries to obtain the most current application-server-specific agent name. The agent periodically checks for a new name.

If a change to application server configuration results in an agent name change, the agent automatically renames itself. In the Investigator tree, the agent appears to disconnect. The disconnected agent remains in the Investigator tree, and unmounts automatically after the unmount time period has elapsed, or can be unmounted manually.

The renamed agent reconnects to the Enterprise Manager and appears in the Investigator tree. The agent logs these changes.

See *Advanced automatic agent naming options* on page 107, for information on configuring automatic agent naming properties for Enterprise Manager connection delay, and rename checking interval time.

Enabling automatic agent naming

To enable automatic agent naming:

1. In the `IntroscopeAgent.profile`, set `introscope.agent.agentAutoNamingEnabled` to `true`.
2. Make these application server-specific changes:
   - For WebLogic, create an Introscope Startup Class. See *Configuring startup class for WebLogic 8.1 or 9.0* on page 130.
   - For WebSphere, create an Introscope Custom Service. See *Configuring a custom service in WebSphere 5.0, 6.0, or 6.1* on page 131.
   - For JBoss, create an XML file. See *To deploy web application support for JBoss:* on page 35.

Advanced automatic agent naming options

You can change these automatic agent naming configurations if appropriate.

Naming options	How to change
Initial Enterprise Manager Connection Delay	When using the automatic agent naming feature, the agent waits up to a configurable amount of time before connecting to the Enterprise Manager while trying to find agent name information. The default delay is 120 seconds. **To change the delay value:** 1 Open the `IntroscopeAgent.profile`. 2 Under the **Agent Name** section, configure the desired delay in the property `introscope.agent.agentAutoNamingMaximumConnectionDelayInSeconds`. 3 Restart the application server.

Naming options	How to change
Agent Rename Check Interval	When using the automatic agent naming feature, the agent periodically checks to see if the naming information from the application server has changed. The default interval is ten minutes. **To change this interval:** **1** Open the `IntroscopeAgent.profile`. **2** Under the **Agent Name** section, configure the desired interval in the `introscope.agent.agentAutoRenamingIntervalInMinutes` property. **3** Restart the application server.
Turning Off Agent Log File Automatic Naming	By default, when the agent name is found automatically, either by information provided by a Java system property or application server, the log files associated with that agent are named automatically using that same information. However, you can turn off this automatic log naming, and continue to use the agent log name specified in the `IntroscopeAgent.profile`. **To turn off agent log file automatic naming:** **1** Open the `IntroscopeAgent.profile`. **2** Set the property, `introscope.agent.disableLogFileAutoNaming`, to a value of `true`. **3** Save the `IntroscopeAgent.profile`. **4** Restart the application server.

Disabling agent naming for WebSphere

Agent automatic naming is enabled by default for all WebSphere platforms, but is not supported in WebSphere 5.x and 6.0 for z/OS. If you use one of these WebSphere versions, you must disable the automatic agent naming feature.

1. Open the `IntroscopeAgent.profile`.
2. Set the value of the property `introscope.agent.agentAutoNamingEnabled`, to `false`.

Enabling cloned agent naming in clustered environments

If two agents exist with the same name monitoring the same host and process and are not uniquely named by a user, the name is appended with a number. Cloned agent naming enables you to correlate an agent with a particular application instance in a clustered application.

You are running cloned agents if you:

- are running agents that share a host, process, or Java Agent name with one or more other agents, or
- are running two or more agents that are using the same agent profile.

Cloned agent naming scenario

With the Java Agent cloning property turned on, if you have four Java Agents, all named `AgentX`, the Enterprise Manager names the agents `AgentX-1`, `AgentX-2`, `AgentX-3` and `AgentX-4`. If `AgentX-1` disconnects and then reconnects, it will still use `AgentX-1` as its name. With this naming, you will never have more Java Agent names in the database than the number of Java Agents originally cloned.

Enabling cloned agent naming in the agent profile

To enable cloned agent naming:

1. Stop your managed application and the Java Agent.
2. Open the `IntroscopeAgent.profile` and set the following property to true:

   ```
   introscope.agent.clonedAgent=true
   ```
3. Save the `IntroscopeAgent.profile`.
4. Restart your managed application and the Java Agent.

Configuring unique names for application instances

If you monitor multiple instances of the an application on the same machine, you can configure unique agent names explicitly:

1. Create a separate agent profile for each application.
2. Uniquely name each agent in the agent profile.
3. Specify which agent profile each application should use.

CHAPTER 7

Java Agent Monitoring and Logging

While the Introscope and the Java Agent monitors your applications, Introscope can also monitor the health and activity of the Java Agent itself. This chapter contains information on monitoring your agents health, as well as logging options for the Java Agent.

Configuring connection metrics

By default, Introscope generates metrics on the connection status of agents connected to an Enterprise Manager, which you can monitor. Java Agent connection metrics appear in the Workstation Investigator under the Enterprise Manager process (the custom metric host):

```
Custom Metric Host (Virtual) \ Custom Metric Agent (Virtual) \ Agents \
[Host_Name] \ [Agent Process Name] \ [Agent_Name] \ ConnectionStatus
```

Connection metrics have these values:

- 0—No data about the agent is available
- 1—agent is connected
- 2—agent is slow to report
- 3—agent is disconnected

An agent disconnecting also generates a "What's Interesting" event. As with other events, users can query for agent disconnects using the historical query interface. Agent disconnect events are part of the data used in assessing application health in the Overview tab in the Investigator.

Once an agent disconnects from the Enterprise Manager, Introscope continues to generate disconnected state metrics until the agent is timed out. When an agent times out, no additional connection metrics are generated or reported to the Enterprise Manager.

To configure the agent connection time out:

1. Open the `IntroscopeEnterpriseManager.properties` file located in the `<Introscope_Home>/config` directory.
2. Modify this property:

   ```
   introscope.enterprisemanager.agentconnection.metrics.agentTimeoutInMinu
     tes
   ```

   The time increment is in minutes.
3. Save the `IntroscopeEnterpriseManager.properties`

   For information about Enterprise Manager properties, see the *Introscope Configuration and Administration Guide*.

Turning off socket metrics

Metrics that trace per-socket bandwidth have a potential for high overhead. If the collection and reporting of network metrics are consuming a lot of processor or I/O time, you can turn off the reporting of the socket metric information.

To turn off reporting of socket metrics:

1. Open the `IntroscopeAgent.profile` file, located in the `<Agent_Home>\wily` directory.
2. Modify this property to have a value of `false`:

   `introscope.agent.sockets.reportRateMetrics=false`
3. Save the `IntroscopeAgent.profile`.

Configuring logging options

When the Java Agent is installed on an application server, after the server starts up a log directory is created here: `<Agent_Home>/wily/logs`. The application server process must have full read/write/execute permissions on the Wily Java Agent directory. To accomplish this, install the Java Agent on the same operating system as the user who runs the application server process. Or, install the Java Agent as a different user, then use the `chmod` command to bestow the necessary permissions.

The Java Agent has the option to run in verbose mode. Verbose mode records higher levels of details about actions and agent interactions with your environment. This information is useful in solving issues with your environment or agent functionality.

Introscope uses Log4J functionality for these functions. If you want to use other Log4J functionality, please see Log4J documentation:

http://jakarta.apache.org/log4j/docs/documentation.html.

Running the agent in verbose mode

Running the agent in verbose mode records higher levels of information to the agent log.

To run the agent in verbose mode:

1. Open the `IntroscopeAgent.profile`, located in the `<Agent_Home>\wily` directory.
2. Modify this property, replacing the existing `INFO` with `VERBOSE#com.wily.util.feedback.Log4JSeverityLevel`:

```
log4j.logger.IntroscopeAgent=VERBOSE#com.wily.util.feedback.Log4JSever
  ityLevel, console, logfile
```

3 Save the `IntroscopeAgent.profile`.

» **Note** Changes to this property take effect immediately and do not require the managed application to be restarted.

Redirecting agent output to a file

The property that controls the agent logging in verbose mode also controls where the agent log is output and the location of this log file (see *Running the agent in verbose mode* on page 113 for more information).

To redirect agent output to a file:

1 Open the `IntroscopeAgent.profile`, located in the `<Agent_Home>\wily` directory.

2 Find the property: `log4j.logger.IntroscopeAgent`

The options for this property are:

- `console`: the information in the logfile is sent to the console
- `logfile`: the information in the logfile is sent to a logfile. If this is selected, the location of the log file is configured using the `log4j.appender.logfile.File` property. See *Changing the name or location of the agent logfile*, below.

For example, if you wanted the agent to report in verbose mode to just a logfile, the property would be set to:

```
log4j.logger.IntroscopeAgent=VERBOSE#com.wily.util.feedback.Log4JSever
  ityLevel,logfile
```

If you wanted the agent to report to both a logfile and console, you would include both `logfile` and `console` in the property.

» **Note** By default the agent log, `IntroscopeAgent.log` is written to the `<Agent_Home>\wily\logs` directory. If you configured agent autonaming options, the agent log files are also automatically named, as described in *Agent log files and automatic agent naming* on page 115.

3 Save the `IntroscopeAgent.profile`.

Changing the name or location of the agent logfile

You can also change the location and name of a logfile by modifying a property.

To change the name or location of the logfile:

1. Open the `IntroscopeAgent.profile`, located in the `<Agent_Home>\wily` directory.
2. Locate the `log4j.appender.logfile.File` property.

   If `logfile` was specified in the `log4j.logger.IntroscopeAgent` property, the location of the log file is configured using the `log4j.appender.logfile.File` property. See *step 2* in *Redirecting agent output to a file* on page 114 for more information.

   » **Note** System properties (Java command line -D options) are expanded as part of the file name. For example, if a Java command starts with `-Dmy.property=Server1`, then `log4j.appender.logfile.File=logs/Introscope-${my.property}.log` is expanded to: `log4j.appender.logfile.File=logs/Introscope-Server1.log`.
3. Set the location and name of the log file, using a fully qualified path to the new location and file. For example:

   ```
   log4j.appender.logfile.File=C:/Logs/AgentLog1.log
   ```
4. Save the `IntroscopeAgent.profile`.

Agent log files and automatic agent naming

If you use the automatic agent naming functionality, by default the log files associated with an agent are named automatically using the same information used to name the agent.

Automatic agent naming affects the log file in the following way:

- If the original name of the logfile does not end in `.log`, a period and `log` is added.
- All characters that are not letters or digits will be replaced by underscores
- If advanced Log4J functionality is used, the agent logfile automatic naming capability might not work.

The following examples show how an agent logfile is named. The examples use an agent name of `DOM1//ACME42`, where `DOM1` is the WebLogic domain, and `ACME42` is the instance of the agent.

When an agent log file is created (named `AutoProbe.log` by default), if the agent name is not yet available, a timestamp is included in the filename:

```
AutoProbe.20040416-175024.log
```

Once the agent name becomes available, the logfile is renamed using the agent's automatic name:

```
AutoProbe.DOM1_ACME42.log
```

You can disable automatic log naming - see *Advanced automatic agent naming options* on page 107 for more information.

Logging considerations for WebSphere z/OS

There are some things to consider when logging in a WebSphere z/OS environment.

Tagging log output as EBCDIC

Beginning with version 5.0, WebSphere for z/OS changed its default encoding from EBCDIC CP1047 to ASCII ISO8859-1. Because z/OS is normally an EBCDIC machine, any logging data written by the Java Agent or AutoProbe must be tagged to use EBCDIC as the final output stream, instead of ASCII.

To tag data as EBCDIC instead of ASCII:

1. Open the `IntroscopeAgent.profile`, located in the `<Agent_Home>\wily` directory.
2. Add these properties to the `IntroscopeAgent.profile`:

   ```
   log4j.appender.console.encoding=IBM-1047
   log4j.appender.logfile.encoding=IBM-1047
   ```
3. Save the `IntroscopeAgent.profile`.

Eliminating startup timing issues with logging facilities

A new property has been added for WebSphere z/OS 5.0 and later, which is used to eliminate any startup timing window exposures that can occur with the Introscope logging facilities.

To eliminate the timing window exposures:

1. Open the `IntroscopeAgent.profile`, located in the `<Agent_Home>\wily` directory.
2. Add this property to the `IntroscopeAgent.profile`:

   ```
   introscope.agent.logger.delay=100000
   ```
   The value is in milliseconds, so the default delay in this example is 100 seconds.
3. Save the `IntroscopeAgent.profile`.

Managing ProbeBuilder Logs

ProbeBuilder logs the probes it added during the instrumentation process and the PBDs it used.

ProbeBuilder log name and location

The ProbeBuilder log file location is determined by where you specify Java classes with the ProbeBuilder Wizard or with the Command-Line ProbeBuilder. For a directory, the log file is located inside the destination directory. For a file, the log file is located next to the destination file.

The ProbeBuilder log file is called:

`<original-directory-or-original-file>.probebuilder.log`

`<original-directory>` or `<original-file>` is the Java class location that you specify with the ProbeBuilder Wizard or with the Command-Line ProbeBuilder.

Only the most recent log is kept; all previous log files are overwritten.

AutoProbe log name and location

AutoProbe will always attempt to log the changes it makes. By default the AutoProbe log file is named `AutoProbe.log`.

To change the name or location of the AutoProbe log:

1. Open the `IntroscopeAgent.profile`, located in the `<Agent_Home>\wily` directory.
2. Locate the `introscope.autoprobe.logfile` property and modify the log name and location, using a fully qualified file path. Non-absolute names are resolved relative to the location of the `IntroscopeAgent.profile` file.

   » **Note** When loading the agent profile from a resource on a classpath, AutoProbe is unable to write to the AutoProbe log file, because the `IntroscopeAgent.profile` file is located within a resource.

   You must restart the managed application before changes to this property take effect.
3. Save the `IntroscopeAgent.profile`.

CHAPTER 8

Using Virtual Agents to Aggregate Metrics

This chapter has information about configuring and using Virtual Agents.

Understanding Virtual Agents

You can configure multiple physical agents into a single Virtual Agent. A Virtual Agent enables an aggregated, logical view of the metrics reported by multiple agents.

A Virtual Agent is useful if you manage clustered applications with Introscope—a Virtual Agent comprised of the agents that monitor different instances of the same clustered application appears in Introscope as a single agent. This allows metrics from multiple instances of a clustered application to be presented at a logical, application level, as opposed to separately for each application instance.

You can view performance and availability data for a specific application instance, by scoping your views and interactions in terms of a single agent.

Virtual Agent requirements

A Virtual Agent can only contain agents that report to the same Enterprise Manager. If you have multiple stand-alone Enterprise Managers, you will need a Virtual Agent for each Enterprise Manager.

Agents that report to Enterprise Managers within a single cluster can belong to the same Virtual Agent, regardless of the Collector Enterprise Manager to which they report. See the *CA Wily Introscope Configuration and Administration Guide* for more information about clustering.

Consider these conditions when configuring Virtual Agents:

- An agent can be assigned to multiple Virtual Agents.
- Virtual Agents cannot include other Virtual Agents.
- If you define multiple Virtual Agents, they must have unique names, including custom metric agents—all agents in a cluster must have unique names.

Configuring Virtual Agents

You configure Virtual Agents using the `agentclusters.xml` file, located in the `<Introscope_Home>/config` directory of the Enterprise Manager to which the agents report. If you run clustered Enterprise Managers, you configure Virtual Agents using the `agentclusters.xml` file in the `config` directory of the cluster's Manager of Managers (MOM).

The sample `agentclusters.xml` below defines a Virtual Agent named `BuyNowAppCluster`, in the Introscope SuperDomain. The Virtual Agent includes all agents, on any host, whose agent name starts with `BuyNow`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<agent-clusters xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="agentclusters0.1.xsd" version="0.1"
  <agent-cluster name="BuyNowAppCluster" domain="SuperDomain" >
    <agent-specifier>.*\|.*\|BuyNow.*</agent-specifier>
    <metric-specifier>Frontends\|.*</metric-specifier>
  </agent-cluster>
</agent-clusters>
```

The root element, `<agent-clusters>`, is required. The `<agent-cluster>` element defines a Virtual Agent, and has two required attributes:

- `name`—If you define multiple Virtual Agents, each must have a unique name.
- `domain`—Assigns the Virtual Agent to an Introscope domain.

  If no domain is defined (as `domain=""`) in the agent-cluster definition, the Virtual Agent will default to the SuperDomain.

If you define multiple Virtual Agents, you define an `<agent-cluster>` element for each. The `<agent-cluster>` element requires two child elements:

- `<agent-specifier>`—Contains a regular expression that specifies the agents in the Virtual Agent, using the standard fully qualified agent name:

  `<host> | <process> | <agentName>`
- `<metric-specifier>`—Contains a prefix that specifies the metrics to collect from the agents in the Virtual Agent, in terms of resource type, or subsets of the instances of a resource type. The recommended prefixes are:
  - CPU
  - JMX
  - WebSpherePMI
  - Frontends

» **Note** While the above are the recommended prefixes, any resource can be used as a metric specifier.

The <agent-cluster> element can contain multiple <metric-specifier> stanzas. Note that a higher volume of matching metrics imposes high overhead on the Enterprise Manager, and can ultimately have an effect on Enterprise Manager capacity.

» **Note** Regular expressions and wildcard metric specifiers such as ".*" and "(.*)" are allowed, but should be used with caution. Use of wildcards can result in a high volume of metrics and a performance impact.

A sample agentclusters.xml is available in your <Introscope_Home>/config directory.

CHAPTER 9

Configuring Java Agent Failover

This chapter has information about agent failover.

Understanding agent failover

An agent that cannot connect to its Enterprise Manager, or loses connection with it, can failover to an alternative Enterprise Manager. To enable failover, you specify a list of alternative Enterprise Managers in the `IntroscopeAgent.profile` file.

When an agent configured for failover cannot connect to its default Enterprise Manager, it tries to connect to the next Enterprise Manager on the list of failover hosts. If the agent does not connect with a failover host, it cycles through the Enterprise Managers on the list until it succeeds in connecting. If the agent goes through the list without connecting to an Enterprise Manager, it waits 10 seconds before cycling through the list again.

In a basic Introscope configuration, you define the host and port settings for one Enterprise Manager. To enable agent failover, you define connection properties for backup Enterprise Managers, and a list that specifies the failover order.

Defining backup Enterprise Managers

To enable agent failover, you must define a list of backup Enterprise Managers, creating an alternate communication channel for each as described in *Configuring connection to the Enterprise Manager* on page 36.

To define backup Enterprise Managers:

1. Open the `IntroscopeAgent.profile` file, located in the `<Agent_Home>\wily` directory.
2. Locate the `Enterprise Manager Locations and Names` section of the profile.
3. Add Enterprise Managers and their connection information to this section. The following example contains the primary Enterprise Manager connection information, as well as two backup Enterprise Managers.

   » **Note** Be sure to assign each additional Enterprise Manager communication channel a unique name—do not use the name DEFAULT or the name of an existing channel when creating a new one.

   This is the primary Enterprise Manager connection information:

```
introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT=enterprise
introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT=5001
introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT=com
   .wily.isengard.postofficehub.link.net.DefaultSocketFactory
```

   This the first backup Enterprise Manager:

```
introscope.agent.enterprisemanager.transport.tcp.host.BackupEM1=voyager
introscope.agent.enterprisemanager.transport.tcp.port.BackupEM1=5002
```

```
introscope.agent.enterprisemanager.transport.tcp.socketfactory.BackupEM1=
  com.wily.isengard.postofficehub.link.net.DefaultSocketFactory
```

This is the second backup Enterprise Manager:

```
introscope.agent.enterprisemanager.transport.tcp.host.BackupEM2=space9
introscope.agent.enterprisemanager.transport.tcp.port.BackupEM2=5003
introscope.agent.enterprisemanager.transport.tcp.socketfactory.BackupEM2=
  com.wily.isengard.postofficehub.link.net.DefaultSocketFactory
```

4 Save the `IntroscopeAgent.profile`.

Defining failover connection order

After specifying the connection properties for your backup Enterprise Managers, define the order in which the agents will attempt to connect to the backup Enterprise Manager.

To specify the connection order:

1 In the `IntroscopeAgent.profile` file, locate the `introscope.agent.enterprisemanager.connectionorder` property.

2 List the Enterprise Manager communication channels—for the agent's primary Enterprise Manager as well as the backups. Put the primary Enterprise Manager first in the list. For the Enterprise Manager communication channels specified *Defining backup Enterprise Managers* on page 124, the connection order could be specified like this:

```
introscope.agent.enterprisemanager.connectionorder=DEFAULT,BackupEM1,Backu
  pEM2
```

3 Save the `IntroscopeAgent.profile`.

Configuring failback to primary Enterprise Manager

In the default agent failover scenario, if the agent loses connection to its primary Enterprise Manager, it tries to connect to the next Enterprise Manager defined in the agent profile. You can also configure the agent to periodically try to reconnect to the primary Enterprise Manager.

To configure attempts to reconnect to the primary Enterprise Manager:

1. In the `IntroscopeAgent.profile`, locate the `Enterprise Manager Failback Retry Interval` section.
2. Uncomment this property:

   `introscope.agent.enterprisemanager.failbackRetryIntervalInSeconds`

   and set the interval in which the agent will attempt to reconnect to its primary Enterprise Manager. The default interval is 120 seconds.
3. Save the `IntroscopeAgent.profile`.
4. Restart the application.

   » **Note** You must restart the managed application before changes to this property take effect.

Configuring domain/user information

To use agent failover and also have users, domains, and authentication settings defined, you must ensure that this information is in sync across the specified failover Enterprise Managers. For more information on domain and user permissions, see the *CA Wily Introscope Configuration and Administration Guide*.

SECTION III

Tailoring and Extending Data Collection

The chapters in this section have information about tailoring and extending Java Agent data collection.

CHAPTER 10

Configuring Access to Application Server Data

This chapter has information about configuring an agent to obtain management information from a WebLogic Server or WebSphere application server.

Application server management data

In WebLogic Server and WebSphere environments, the Java Agent can obtain and report management information from the application server, above and beyond the metrics resulting from instrumenting your applications. For example, you can configure an agent to:

- report JMX metrics from the application server
- report WebLogic Diagnostic Framework (WLDF) data from WebLogic 9.0
- report Performance Monitoring Infrastructure (PMI) from WebSphere
- obtain its name from the application server

The Application Overview in the Workstation (available in Introscope v7.0 and later) uses JMX and PMI metrics, if available, in application health heuristics. Enabling the Java Agent to access application server management information is not required, but it enhances the visibility provided by the Application Overview.

To enable the Java Agent to obtain and use data from the application server you configure an Introscope startup class or service in the application server, and target it at application server instances, or to an application server cluster.

Configuring startup class for WebLogic 8.1 or 9.0

This section describes how to create a startup class in WebLogic 8.1 or 9.0. For information about creating a startup class in other versions of WebLogic Server, or for more information about WebLogic Server, consult your WebLogic Server documentation.

To configure a startup class for WebLogic 8.1 or 9.0:

1. Open the WebLogic Administrative Console.
2. In the left pane, expand the Deployments folder.
3. Click the **Startup & Shutdown** folder.

   The Startup and Shutdown page opens.
4. Click **Configure a New Startup Class**.

   The **Configuration** tab is shown.
5. In the **Name** field, enter:

   ```
   Introscope Startup Class
   ```
6. In the **ClassName** field, enter:

   ```
   com.wily.introscope.api.weblogic.IntroscopeStartupClass
   ```
7. Click **Create**.

   The **Target and Deploy** tab appears.

8. Check the box(es) for the server(s) you'd like to make this startup class available to.
9. Click **Apply**. Select the "run before deploying apps" option.
10. Add the location of the `WebAppSupport.jar` to the application startup classpath.
11. Restart the application server.

Configuring a custom service in WebSphere 5.0, 6.0, or 6.1

This section describes how to create a custom service in WebSphere 5.0. To create a custom service in previous versions, or for more information, consult your WebSphere documentation.

To configure a custom service:

1. Open the WebSphere Administrative Console.
2. Select the server you'd like to configure, then:
   - For WebSphere 6.1, navigate to **Server Infrastructure > Administration > Custom Services**.
   - For WebSphere 6.0, click **Administration**, then **Custom Services**.
   - For WebSphere 5.0, click **Custom Services.**
3. Click **New** to add a new Custom Service, then:
   - For WebSphere 6.0, check the box for **Enable service at server startup**
   - For WebSphere 5.0, check the **Startup** check box.
4. In the **Classname** field, enter:

   ```
   com.wily.introscope.api.websphere.IntroscopeCustomService
   ```
5. In the **Display Name** field, enter:

   ```
   Introscope Custom Service
   ```
6. In the **Classpath** field, enter:

   ```
   <WebSphere_Home>/wily/WebAppSupport.jar
   ```
7. Click **OK**.
8. Restart the application server.

CHAPTER 11

Configuring Boundary Blame

This chapter describes default Java Agent blame reporting behaviors, and related configuration options.

Understanding Boundary Blame

Introscope's Blame Technology works in a managed Java Application to enable you to view metrics at the front and backends of your application. This capability, referred to as boundary blame, allows users to triage problems in the application front or backends.

Introscope uses the SQL statement in the SQL Agent monitoring functionality to automatically detect backends. If the SQL Agent is unavailable, Introscope automatically detects socket calls as backends, because backends such as client/server databases, JMS servers, and LDAP servers are accessed through a socket. If you have Oracle backends and do not use the Introscope SQL Agent, see *Boundary Blame and Oracle backends* on page 95.

For information about how boundary blame is presented in the Introscope Investigator, see the *Introscope Workstation User Guide*.

Using URL groups

You can use URL Groups to monitor browser response time for sets of requests whose path prefix begins with a string you define. The path prefix is the portion of the URL that follows the hostname. For example, in this URL:

```
http://burger1.com/testWar/burgerServlet?ViewItem&category=
  11776&item=5550662630&rd=1
```

the path prefix is:

```
/testWar
```

You can define a URL group for any useful category of requests that can be derived from a URL's path prefix. For example, depending on the form of your application URLs, you could define URL groups for each customer your application supports, for each major application, or for sub-applications. This enables you to monitor performance in the context of committed service levels, or for mission-critical portions of your application.

The following example is an excerpt from a Java Agent profile, showing how URL Groups are defined:

Example URL group property definitions

```
introscope.agent.urlgroup.keys=alpha,beta,gamma
introscope.agent.urlgroup.group.alpha.pathprefix=/testWar
introscope.agent.urlgroup.group.alpha.format=foo {host} bar {protocol} baz
  {port} quux {query_param:foo} red {path_substring:2:5} yellow
  {path_delimited:/:0:1} green {path_delimited:/:1:4} blue
  {path_substring:0:0}
introscope.agent.urlgroup.group.beta.pathprefix=/nofilterWar
```

```
introscope.agent.urlgroup.group.beta.format=nofilter foo {host} bar
  {protocol} baz {port} quux {query_param:foo} red {path_substring:2:5}
  yellow {path_delimited:/:0:1} green {path_delimited:/:1:4} blue
  {path_substring:0:0}
introscope.agent.urlgroup.group.gamma.pathprefix=/examplesWebApp
introscope.agent.urlgroup.group.gamma.format=Examples Web App
```

Configuring URL groups

This section provides information about the properties that configure URL Groups.

Defining keys for URL groups

The property `introscope.agent.urlgroup.keys` defines a list of the IDs, or keys, of all of your URL Groups. The key for a URL Group is referenced in other property definitions that declare an attribute of the URL group. The following example defines the keys for three URL Groups:

```
introscope.agent.urlgroup.keys=alpha,beta,gamma
```

If you define URL Groups so some URLs fall into multiple groups, the order in which you list the keys for the URL Groups in the property is important. The URL Group with the narrower membership should precede the URL Group with broader membership. For example, if the IP Group with key **`alpha`** has the path prefix `/examplesWebApp` and the URL Group with key **`delta`** has the path prefix `/examplesWebApp/cleverones`, `delta` should precede `alpha` in the `keys` parameter

Defining membership of each URL group

The property `introscope.agent.urlgroup.group.`**`groupKey`**`.pathprefix` specifies a pattern against which the path prefix of a URL is matched, defining which requests fall within the URL Group.

Example 1

This property definition assigns all requests in which the path portion of the URL starts with `/testWar` to the URL Group whose key is `alpha`:

```
introscope.agent.urlgroup.group.alpha.pathprefix=/testWar
```

Requests that match the specified `pathprefix` include:

```
http://burger1.com/testWar/burgerServlet?ViewItem&category=
  11776&item=5550662630&rd=1
http://burger1.com/testWar/burgerServlet?Command=Order&item=5550662630
```

Example 2

A company that provides call center services could monitor response time for functional areas by setting up a URL Group for each application function. If customers access services with this URL:

`http://genesystems/us/application_function/`

where *application_function* corresponds to applications such as `OrderEntry`, `AcctService`, and `Support`, the `pathprefix` property for each URL group would specify the appropriate *application_function*.

» **Note** You can use the asterisk symbol (*) as a wildcard in `pathprefix` properties.

Define name for a URL group

`The property introscope.agent.urlgroup.group.groupKey.format` determines the names under which response time metrics for a URL group whose key is *groupKey* appear in the Introscope Workstation.

Typically, the `format` property is used to assign a text string as the name for a URL. The following example causes metrics for the URL Group with key `alpha` to appear in the Workstation under the name `Alpha Group`:

```
introscope.agent.urlgroup.group.alpha.format=Alpha Group
```

Advanced naming techniques for URL groups (optional)

You can derive a URL Group name from request elements such as the server port, the protocol, or from a substring of the request URL. This is useful if your application modules are easily differentiated by inspecting the request. This section describes advanced forms of the `format` property.

Using host as URL group name

To organize metrics for a URL group under names that reflect the hostname of the HTTP server associated with requests, define the `format` parameter like this:

```
introscope.agent.urlgroup.group.alpha.format={host}
```

When `format={host}`, statistics for these requests would appear under the metric names `us.mybank.com` and `uk.mybank.com` respectively:

```
https://us.mybank.com/mifi/loanApp......
```

```
https://uk.mybank.com/mifi/loanApp.....
```

Using protocol as URL group name

To organize statistics for a URL group under names that reflect the protocol associated with requests, define the `format` parameter like this:

```
introscope.agent.urlgroup.group.alpha.format={protocol}
```

When `format={protocol}` statistics are grouped in Investigator under metric names that correspond to the protocol portion of request URLs. For example, statistics for these requests would appear under the metric name `https`:

```
https://us.mybank.com/cgi-bin/mifi/scripts......
https://uk.mybank.com/cgi-bin/mifi/scripts......
```

Using port as URL group name

To organize statistics for a URL group under names that reflect the port associated with requests, define the `format` parameter like this:

```
introscope.agent.urlgroup.group.alpha.format={port}
```

When `format={port}`, statistics are grouped under names that correspond to the port portion of request URLs. For example, statistics for these requests would appear under the name `9001`.

```
https://us.mybank.com:9001/cgi-bin/mifi/scripts......
https://uk.mybank.com:9001/cgi-bin/mifi/scripts......
```

Using parameter as URL group name

To organize statistics for a URL group in Investigator under metric names that reflect the value of a parameter associated with requests, define the `format` parameter like this:

```
introscope.agent.urlgroup.group.alpha.format={query_param:param}
```

When `format={query_param:`*`param`*`}` statistics are grouped in Investigator under metric names that correspond to value of the parameter specified. Requests without parameters are listed under <empty>. For example, given this parameter definition:

```
introscope.agent.urlgroup.group.alpha.format=
  {query_param:category}
```

Statistics for these requests would appear under the metric name "734"

```
http://ubuy.com/ws/shoppingServlet?ViewItem&category=734
  &item=3772&tc=photo
http://ubuy.com/ws/shoppingServlet?ViewItem&category=734
  &item=8574&tc=photo
```

Using a substring of the request path as URL group name

To organize statistics for a URL group under names that reflect a substring of the path portion of request URLs, define the `format` parameter like this:

```
introscope.agent.urlgroup.group.alpha.format=
  {path_substring:m:n}
```

where *m* is the index of the first character, and *n* is one greater than the index of the last character. String selection operates like the `java.lang.String.substring()` method. For example, given this setting:

```
introscope.agent.urlgroup.group.alpha.format=
  {path_substring:0:3}
```

Statistics for this request would appear under the metric node "/ht"

```
http://research.com/htmldocu/WebL-12.html
```

Using delimited portion of the request path as URL group name

To organize statistics for a URL group under names that reflect a character-delimited portion request URL path, define the `format` parameter like this:

```
introscope.agent.urlgroup.group.alpha.format=
{path_delimited:delim_char:m:n}
```

where *delim_char* is the character that delimits the segments in the path, `m` is the index of the first segment to select, and `n` is one greater than the index of the last segment to select. For example, given this setting:

```
introscope.agent.urlgroup.group.alpha.format=
{path_delimited:/:2:4}
```

statistics for the requests of this form:

```
http://www.buyitall.com/userid,sessionid/pageid
```

would appear under the metric name `/pageid`

Note that:

- a delimiter character counts as a segment
- the segment count starts at 0

This table shows the segments as delimited by the slash character:

Segment Index	0	1	2	3
Segment String	/	*userid,sessionid*	/	*pageid*

You can specify multiple delimiters as necessary. For example, given this setting:

```
introscope.agent.urlgroup.group.alpha.format=
{path_delimited:/,:3:4}
```

statistics for requests of the form shown above would appear under the metric name `sessionid`.

This table shows the segments as delimited by the slash and the comma character:

Segment Index	0	1	2	3	4	5
Segment String	/	*userid*	,	*sessionid*	/	*pageid*

Using multiple naming methods for URL groups

You can combine multiple naming methods in a single `format` string, as shown below:

```
introscope.agent.urlgroup.group.alpha.format=red {host} orange {protocol}
  yellow {port} green {query_param:foo} blue {path_substring:2:5} indigo
  {path_delimited:/:0:1} violet {path_delimited:/:1:4} ultraviolet
  {path_substring:0:0} friend computer
```

Running the URLGrouper

URLGrouper is a command-line utility that analyzes a web server log file in Common format, and produces BRTA property settings for a set of URL Groups. Using URLGrouper gives you a starting point for defining your own URL Groups.

» **Note** You can use the URLGRouper utility to analyze your Web server log file. URLGrouper outputs a set of property settings for potential URL Groups, based on the contents of the Web server log file. The asterisk symbol (*) can be used with URLGrouper as a wildcard.

To run URLGrouper:

1. Open a command shell.
2. Enter this command

   ```
   java -jar urlgrouper.jar logfile
   ```

   where *logfile* is the full path to your web server log file.
3. Property definitions for a set of URL Groups are output to STDOUT.
4. To configure the proposed URL Groups, copy the property statements produced by URL Grouper into the `IntroscopeAgent.profile`.

Using Blame tracers

You can use tracers to explicitly mark the frontends and backends in your application. For more information, see *Using Blame Tracers to mark blame points* on page 94.

Disabling Boundary Blame

By default, Boundary Blame is enabled. To disable boundary blame in favor of the component-level blame implemented in Introscope versions earlier than 7.0, use the `introscope.agent.blame.type` property described on *page 198*.

CHAPTER 12

Configuring Transaction Trace Options

This chapter has information about default Transaction Tracing behaviors and related configuration options.

Controlling automatic Transaction Tracing behavior

Automatic Transaction Tracing enables historical analysis of potentially problematic transaction types without explicitly running Transaction Traces. Introscope offers two types of automatic Transaction Tracing:

- Transaction Trace sampling that is enabled by default, based on your URL groupings
- Configurable automatic trace sampling that gathers trace information regardless of URL groupings

Transaction Trace component clamp

Introscope now sets a clamp (set by default to 5,000 components) to limit the size of traces. When this limit is reached, warnings appear in the log, and the trace stops.

This allows you to clamp an infinitely expanding transaction—for example when a servlet executes hundreds of object interactions and backend SQL calls. Without the clamp, Transaction Tracer views this as one transaction, continuing infinitely. Without a clamp in place, the JVM runs out of memory before the trace can be completed.

The new property for clamping infinitely expanding transactions is in the `IntroscopeAgent.profile` file:

- `introscope.agent.transactiontrace.componentCountClamp=5000`

For traces producing clamped components—those exceeding the CountClamp—traces are marked with an asterisk and have a tool tip assiociated with them, providing more information about the clamped metrics. For more information about viewing these traces, see the *Introscope Workstation User Guide*.

» **WARNING** If the Transaction Trace component clamp size is increased, the memory required for Transaction Traces may increase. Therefore, the maximum heap size for the JVM may need to be adjusted accordingly, or else the managed application may run out of memory. See the *Introscope Configuration and Administration Guide* for more information.

Transaction trace sampling

Transaction trace sampling is enabled by default. As appropriate you can disable this behavior. For more information on Transaction Trace properties, see *Transaction tracing* on page 214.

When you configure automatic trace sampling, you specify the number of transactions to trace, during a time interval you specify.

» **Note** These properties are located, by default, in the Enterprise Manager properties file. Before changing the defaults for the `sampling.perinterval` and `sampling.interval` properties, consider the potential for increased load in the Enterprise Manager with higher sampling rates. The Enterprise Manager will push this configuration to all agents connected to the Enterprise Manager. Configuring these properties in the agent will overwrite the configuration set by the Enterprise Manager for an individual agent.

To configure automatic trace sampling, modify these properties:

- `introscope.agent.transactiontracer.sampling.enabled`

  Set to `false` to disable Transaction Trace sampling. The default value is `true`.
- `introscope.agent.transactiontracer.sampling.perinterval.count`

  Specifies the number of transactions to trace, during the interval you specify. The default number of transactions is 1.
- `introscope.agent.transactiontracer.sampling.interval.seconds`

  Specifies the length of time to trace the number of transactions you specify. The default interval is every 2 minutes.

Configuring cross-process Transaction Tracing

Transaction Tracer can trace transactions that cross JVM boundaries on WebLogic Server 8 or later, or WebSphere 6.0—if the environment is comprised of compatible versions of the same vendor's application server.

Cross-process transaction tracing is supported for synchronous transactions, for instance, servlets to EJBs.

Enabling cross-process tracing in WebSphere

1. Configure web application support. Follow the instructions in *Configuring a custom service in WebSphere 5.0, 6.0, or 6.1* on page 131.
2. Turn on the work area service.

   From the administration page, servers->application servers, click on server1, click on Business Process Services, click on Work Area Service, check the "Enable service at server startup" box.
3. Set `introscope.agent.websphere.crossjvm=true` in the agent profile.

Enabling cross-process tracing in WebLogic Server

1. Configure web application support. Follow the instructions in *Configuring startup class for WebLogic 8.1 or 9.0* on page 130.

2 Add "`-Dweblogic.TracingEnabled=true`" to the java command line for starting WebLogic Server.

3 Set `introscope.agent.weblogic.crossjvm=true` in the agent profile.

Extending transaction trace data collection

The Java Agent collects basic Transaction Trace data such as Domain/Host/Process/Agent, timestamp, duration, URL, and so forth, by default.

You can configure the Introscope Transaction Tracer to obtain additional information, including User ID data for Servlet and JSP invocations, and other transaction trace data such as HTTP request headers, request parameters, and session attributes. To capture this information, you must define the criteria in the `IntroscopeAgent.profile`.

About User ID data

To configure the Java Agent to identify User IDs for Servlet and JSP invocations, you must first obtain information on how your managed application specifies user IDs. The Application Architect who developed the managed application can probably provide this information.

Introscope Transaction Tracer can identify User IDs from managed applications that store User IDs in one of these ways:

- HttpServletRequest.getRemoteUser()
- HttpServletRequest.getHeader (String key)
- HttpSession.getValue (String key), where returned object is either a String representing the UserID, or an Object whose toString() returns to the UserID

If your managed application stores User IDs using one of these methods, see *Configuring Agent to collect additional transaction trace data* on page 145, to configure Java Agent settings to collect User ID data.

About servlet request data

Using Introscope, you can collect transaction trace data that matches user-configurable parameters. For example, you can specify the Introscope Agent to collect transaction trace data for transactions that contain the **User-Agent** HTTP request header.

Introscope can record this servlet request information:

- request headers
- request parameters
- session attributes

To record this servlet request information for your managed application, see *Configuring Agent to collect additional transaction trace data* on page 145 to configure Java Agent settings to collect this data.

Configuring Agent to collect additional transaction trace data

You can configure the Java Agent to collect additional transaction trace data such as User ID, HTTP request headers, HTTP request parameters, or HTTP session attributes.

To configure the Java Agent to collect additional transaction trace data:

1. Open the agent profile, `IntroscopeAgent.profile`.
2. Locate the Transaction Tracer properties under the *Transaction Tracer Configuration* heading.

Collecting user id data

To configure the Java Agent to identify User IDs

- Configure the properties that correspond to the method your managed application uses to store User IDs.

  » **Note** Ensure that only one set of properties are not commented, or the wrong properties might be used.

  - For HttpServletRequest.getRemoteUser(), uncomment the property:

    ```
    introscope.agent.transactiontracer.userid.method=HttpServletRequest.getRemoteUser
    ```

  - For HttpServletRequest.getHeader (String key), uncomment the following pair of properties, and define a key string for the second property:

    ```
    introscope.agent.transactiontracer.userid.method=HttpServletRequest.getHeader
    introscope.agent.transactiontracer.userid.key=<application defined key string>
    ```

  - For HttpSession.getValue (String key), uncomment the following pair of properties, and define a key string for the second property:

    ```
    introscope.agent.transactiontracer.userid.method=HttpServletRequest.getValue
    introscope.agent.transactiontracer.userid.key=<application defined key string>
    ```

Collecting servlet request data

To record servlet request information such as HTTP request headers and parameters:

1. To specify the HTTP request headers for which to collect transaction trace data, uncomment this property, and specify the HTTP request header(s) to track, in a comma-separated list:

   ```
   #introscope.agent.transactiontracer.parameter.httprequest.headers=User-Agent
   ```

2. To specify the HTTP request parameters for which to collect transaction trace data, uncomment this property and specify the HTTP request parameter(s) to track, in a comma-separated list:

   ```
   #introscope.agent.transactiontracer.parameter.httprequest.parameters=parameter1,parameter2
   ```

3. To specify the HTTP session attributes for which to trace data, uncomment this property and specify the HTTP session attribute(s) to track, in a comma-separated list, for example:

   ```
   #introscope.agent.transactiontracer.parameter.httpsession.attributes=cartID,deptID
   ```

4. Restart the managed application.

Disabling the capture of stalls as Events

By default, Introscope captures transaction stalls as events in the Transaction Event database, and generates stall metrics from the detected events. Stall metrics are generated for the first and last method in the transaction. Users can view stall Events and associated metrics in the Workstation's Historical Event Viewer.

» **Note** Generated stall metrics are always available, but stall events are only visible if Introscope Error Detector is installed. Stalls are stored as ordinary errors, and will be visible in the Errors TypeView, or in the historical query viewer by querying for "type:errorsnapshot".

You can disable the capture of stalls as events, change the stall threshold, or change the frequency with which the agent checks for stalls using these properties:

- `introscope.agent.stalls.enable` controls whether the Java Agent checks for stalls and creates events for detected stalls.
- `introscope.agent.stalls.thresholdseconds` specifies the minimum threshold response time at which time a transaction is considered stalled.
- `introscope.agent.stalls.resolutionseconds` specifies the frequency that the agent checks for stalls.

CHAPTER 13

Configuring the Introscope SQL Agent

This chapter has instructions for configuring Introscope SQL Agent.

The SQL Agent overview

The Introscope SQL Agent reports detailed database performance data to the Enterprise Manager. The SQL Agent provides visibility into the performance of individual SQL statements in your application by tracking the interaction between your managed application and your database.

In the same way that the Java Agent monitors Java applications, the SQL Agent monitors SQL statements. The SQL Agent is non-intrusive, monitoring the application or database with very low overhead.

To provide meaningful performance measurements down to the individual SQL statement level, the SQL Agent summarizes performance data by stripping out transaction-specific data and converting the original SQL statements into Introscope-specific normalized statements. Since normalized statements do not include sensitive information, such as credit card numbers, this process also protects the security of your data.

For example, the SQL Agent converts this SQL query:

```
SELECT * FROM BOOKS WHERE AUTHOR = 'Atwood'
```

to this normalized statement:

```
SELECT * FROM BOOKS WHERE AUTHOR = ?
```

Similarly, SQL Agent converts this SQL update statement:

```
INSERT INTO BOOKS (AUTHOR, TITLE) VALUES ('Atwood', 'The Robber Bride')
```

to this normalized statement:

```
INSERT INTO BOOKS (AUTHOR, TITLE) VALUES (?, ?)
```

» **Note** Only text within quotation marks ('xyz') is normalized.

Metrics for normalized statements are aggregated and can be viewed in the JDBC node of the Workstation Investigator.

The SQL Agent files

When you install an Introscope agent, the agent installer automatically installs the SQL Agent. The following files are installed:

- `wily/ext/SQLAgent.jar`
- `wily/sqlagent.pbd`

» **Note** By default, agent extensions like the `SQLAgent.jar` file are installed in the `wily/ext` directory. You can change the location of the agent extension directory with the `introscope.agent.extensions.directory` property in the agent profile. If you change the location of the `/ext` directory, be sure to move the contents of the `/ext` directory as well.

Supported JDBC drivers and datasources

The SQL Agent supports the following JDBC drivers and JDBC DataSources. Configuration instructions have been simplified to apply to both a JDBC driver and a JDBC datasource.

Supported JDBC drivers	The SQL Agent supports the following JDBC drivers: ■ Oracle—classes111.zip, classes12.zip, classes111_g.zip, classes12_g.zip ■ DB2—db2java.zip ■ Sybase—jconn2.jar ■ WebLogic jDriver for Oracle—jDriver 6.1 The SQL Agent fully supports the JDBC 1.0 and 2.0 specifications, including support for all JDBC driver types, I through IV.
Supported JDBC datasources for WebSphere	In db2java.zip: ■ COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource ■ COM.ibm.db2.jdbc.DB2XADataSource In classes12.zip and classes12_g.zip: ■ oracle.jdbc.pool.OracleConnectionPoolDataSource ■ oracle.jdbc.xa.client.OracleXADataSource
Supported JDBC datasources for WebLogic	In classes12.zip: ■ oracle.jdbc.xa.client.OracleXADataSource

Configure the SQL Agent for WebSphere or WebLogic

This section describes how to configure the SQL Agent to function with WebSphere or WebLogic, using either a JDBC Driver or a JDBC DataSource.

The SQL Agent supports:

- WebSphere Application Server (WAS) 4.0 and higher
- WebLogic Server 6.1 and higher

If other applications use a JDBC DataSource or driver that has been instrumented, you need to add the `Agent.jar` file to the classpaths for those applications. Once this is done, the applications may show up as "Unknown Processes" in the Workstation. This will not affect the functionality or performance of your application and can be ignored.

WebSphere Application Server (WAS) configuration

There are two steps to configuring the SQL Agent for WebSphere Application Server:

Step 1 Configure the JDBC DataSource or driver in WebSphere. See *Configure the JDBC DataSource or Driver in WebSphere* on page 151.

Step 2 Instrument the JDBC DataSource or Driver. See *Instrument the JDBC DataSource or Driver* on page 151.

» Important If you used Application Server AutoProbe or Manual ProbeBuilder to instrument your application, you must complete the configuration procedures. If you used **JVM AutoProbe**, *no further configuration is required*.

WebLogic Server configuration

To configure the SQL Agent for WebLogic Server, you must intrstrument the JDBC DataSource or driver. For more information, see *Instrument the JDBC DataSource or Driver* on page 151.

» Important If you used Application Server AutoProbe or Manual ProbeBuilder to instrument your application, you must complete the configuration procedures. If you used **JVM AutoProbe**, *no further configuration is required*.

Configure the JDBC DataSource or Driver in WebSphere

In your WebSphere environment, configure your JDBC DataSource or driver to work with SQL Agent and WebSphere. For more information, refer to your WebSphere documentation.

Instrument the JDBC DataSource or Driver

The following instructions assume:

- Introscope and ProbeBuilder are installed in your environment.
- you have write permission in the directory that contains the driver file.
- you are able to use the ProbeBuilder Wizard application either on Windows or via X-windows on UNIX. For instructions on command-line use of ProbeBuilder, see the *Using the command-line ProbeBuilder* on page 229.

To instrument a JDBC DataSource or Driver:

1. Shut down WebSphere or WebLogic.
2. Copy the `sqlagent.pbd` file (the default location is `<Introscope_Home>/wily`) to the `<Introscope_Home>\config\custompbd` directory.
3. Locate the file in WebSphere or WebLogic containing the Java classes that implement your JDBC DataSource or driver.
4. Run the ProbeBuilder Wizard, located in the `<Introscope_Home>` directory:
   - `Introscope ProbeBuilder Wizard.exe` on Windows
   - `IntroscopeProbeBuilderWizard` on UNIX
5. At the Welcome screen, click **Next**.
6. On the **Select Original Java Bytecode** screen, select the file containing the JDBC DataSource to instrument. For example, if using a file containing an Oracle JDBC DataSource, the name of the file would be `classes12.zip`.

   » **Note** Before instrumenting any file, save a backup copy in another location.
7. On the **Destination Location** screen, click **Next** to name the resulting instrumented file. For example, if using the file containing an Oracle JDBC DataSource, the name of the resulting file would be `classes12.isc.zip`. Accept the default save location for the resulting file.
8. On the **System Directives** screen, select the **SYSTEM** directives for either WebSphere or WebLogic, depending on which system you are configuring. Click **Next**.
9. On the **Custom Directives** screen, select the `sqlagent.pbd` along with any other custom PBDs used in your deployment.

10 Click **Add Probes**. This creates a copy of the file containing the JDBC DataSource, instrumenting the JDBC DataSource in the copy. The new copy will be located in the same directory as the original.

11 On the Finished screen, click **Exit**.

In your JDBC driver directory you should find the file containing the instrumented JDBC DataSource or driver. To use this file to see the JDBC metrics in Introscope do **one** of the following:

- Set the original file aside and rename the instrumented one, as in the following example:

```
c:\> rename classes12.zip classes12.orig.zip
c:\> rename classes12.isc.zip classes12.zip
```

» **Note** If the file to be renamed is in use, do not attempt to rename it until you first shut down any application or database that is actively using the file containing the JDBC DataSource.

- Change your application server's CLASSPATH setting to point to the new instrumented file.

» **Note** If other applications use this same JDBC DataSource or driver file, you will need to put the `Agent.jar` (located in `<WebSphere_Home>\wily` or `<WebLogic_Home>/wily`) in the classpath for those applications. Otherwise, applications that reference the JDBC DataSource will fail at runtime.

12 Restart your administration or application server.

SQL statement normalization

Some applications may generate an extremely large number of unique SQL statements. If technologies like EJB 3.0 are in use, the likelihood of long unique SQL statements increases. Long SQL statements can contribute to a metric explosion in the agent, leading to poor performance as well as other system problems.

How poorly written SQL statements create metric explosions

If your SQL Agent is showing a large and increasing number of unique SQL metrics even though your application uses a small set of SQL statements, the problem could be in how the SQL statement was written.

In general, the number of SQL Agent metrics should approximate the number of unique SQL statements. A common reason this becomes a problem is because of how comments are used in SQL statements. For example, in this statement,

```
"/* John Doe, user ID=?, txn=? */ select * from table..."
```

the SQL Agent creates the following metric:

```
"/* John Doe, user ID=?, txn=? */ select * from table..."
```

Note that the comment is part of the metric name. While the comment is useful for the database administrator to see who is executing what query, the SQL Agent does not parse the comment in the SQL statement. Therefore, for each unique user ID, the SQL Agent creates a unique metric, potentially causing a metric explosion. The database that executes the SQL statements does not see these metrics as unique because it ignores the comments.

This problem can be avoided is by putting the SQL comment in single quotes, as shown:

```
"/*' John Doe, user ID=?, txn=? '*/ select * from table..."
```

The SQL Agent then creates the following metric where the comment no longer causes a unique metric name:

```
"/* ? */ select * from table..."
```

Example 1

When looking at this path under an agent node in the Investigator Backends|{backendName}|SQL|{sqlType}|sql you notice that temporary tables are being accessed like this:

```
SELECT * FROM TMP_123981398210381920912 WHERE ROW_ID = ?
```

All the additional digits on the `TMP_ table` name are unique and steadily growing causing a metric explosion.

Example 2

You have been alerted to a potential metric explosion and your investigation brings you to a review of this SQL statement:

```
#1 INSERT INTO COMMENTS (COMMENT_ID, CARD_ID, CMMT_TYPE_ID,
  CMMT_STATUS_ID,CMMT_CATEGORY_ID, LOCATION_ID, CMMT_LIST_ID,
  COMMENTS_DSC, USER_ID,LAST_UPDATE_TS) VALUES (?, ?, ?, ?, ?, ?, ?, "CHANGE
  CITY FROM CARROLTON,TO CAROLTON, _ ", ?, CURRENT)
```

In studying the code, you notice that `"CHANGE CITY FROM CARROLTON, TOCAROLTON, _ "` recurs as a dizzying array of cities.

Example 3

You have been alerted to a potential metric explosion and your investigation brings you to a review of this SQL statement:

```
CHANGE COUNTRY FROM US TO CA _ CHANGE EMAIL ADDRESS FROM TO BRIGGIN @ COM _ "
```

In studying the code, you notice CHANGE COUNTRY results in an endless list of countries. In addition, the placement of the quotes for countries results in people's e-mail addresses getting inserted into SQL statements. Here's the source of metric explosion as well as other negative consequences.

SQL statement normalization options

To address long SQL statements, the SQL Agent includes the following normalizers for use:

- *Default SQL statement normalizer*, below
- *Custom SQL statement normalizer* on page 154
- *Regular expression SQL statement normalizer* on page 156
- *Command-line SQL statement normalizer* on page 160

Default SQL statement normalizer

The standard SQL statement normalizer is on by default in the SQL Agent. It normalizes text within single quotation marks ('xyz'). For example, the SQL Agent converts this SQL query:

```
SELECT * FROM BOOKS WHERE AUTHOR = 'Atwood'
```

to this normalized statement:

```
SELECT * FROM BOOKS WHERE AUTHOR = ?
```

Metrics for normalized statements are aggregated and can be viewed in the Workstation Investigator.

Custom SQL statement normalizer

The SQL Agent allows users to add extensions for performing custom normalization. To do so, you create a DLL file containing a normalization scheme that is implemented by the SQL Agent.

To apply a SQL statement normalizer extension:

1 Create an extension DLL file.

» **Note** The entry point class for the SQL normalizer extension file has to implement com.wily.introscope.agent.trace.ISqlNormalizer interface.

Making a DLL extension file involves creating a manifest file that contains specific keys for the SQL normalizer extension, which are detailed in step 2 below. However, for your extension to work, other general keys are required. These keys are the type you would use to construct any extension file. The extension file you create relates to database SQL statement text normalization, for example metrics under the `Backends|{backendName}|SQL|{sqlType}|{actualSQLStatement}` node. The `{actualSQLStatement}` is normalized by the SQL normalizer.

2. Place the following keys in the manifest of the created extension:

   - `com-wily-Extension-Plugins-List:testNormalizer1`

   » **Note** The value of this key can be anything. In this instance, `testNormalizer1` is used as an example. Whatever you specify as the value of this key, use it in the following keys as well.

   - `com-wily-Extension-Plugin-testNormalizer1-Type: sqlnormalizer`
   - `com-wily-Extension-Plugin-testNormalizer1-Version: 1`
   - `com-wily-Extension-Plugin-testNormalizer1-Name: normalizer1`

     Should contain the unique name of your normalizer, for example normalizer1.
   - `com-wily-Extension-Plugin-testNormalizer1-Entry-Point-Class: <Thefully-qualified classname of your implementation of ISQLNormalizer>`

3. Place the extension file you created in the `<Agent_Home>/wily/ext` directory.
4. In the `IntroscopeAgent.profile`, locate and set the following property:

   `introscope.agent.sqlagent.normalizer.extension`

   Set the property to the com-wily-Extension-Plugin-{plugin}-Name from your created extension's manifest file. The value of this property is case-insensitive. For example:

   `introscope.agent.sqlagent.normalizer.extension=normalizer1`

   » **Important** This is a hot property. Changes to the extension name will result in re-registration of the extension.

5. In the `IntroscopeAgent.profile`, you can optionally add the following property to set the error throttle count:

   `introscope.agent.sqlagent.normalizer.extension.errorCount`

   For more information about errors and exceptions, see *Exceptions*, below.

   » **Note** If the errors thrown by the custom normalizer extension exceeds the error throttle count, the extension is disabled.

6. Save the `IntroscopeAgent.profile`.
7. Restart your application.

Exceptions

If the extension you created throws an exception for one query, the default SQL statement normalizer uses the default normalization scheme for that query. When this happens, an ERROR message is logged, saying an exception was thrown by the extension, and a DEBUG message is logged with stack trace information. However, after five such exceptions are thrown, the default SQL statement normalizer disables the your created extension and stops attempting to use the created extension for future queries until the normalizer is changed.

Null or empty strings

If the extension you created returns a null string or empty string for a query, the StatementNormalizer uses the default normalization scheme for that query and logs an INFO message saying the extension returned a null value. However, after five such null or empty strings have been returned, the StatementNormalizer stops logging messages, but will attempt to continue to use the extension.

Regular expression SQL statement normalizer

The SQL Agent ships with an extension that normalizes SQL statements based on configurable regular expressions (regex). This file, `wily.RegexSqlNormalizer.ext.dll`, is located in the `<Agent_Home>/wily/ext` directory. The normalizer extension uses `Systems.Test.RegularExpressions` namespace classes.

For examples on how to use the regular expression SQL statement normalizer, see *Regular expression SQL statement normalizer examples* on page 158.

To apply the regular expressions extension:

1. Open the `IntroscopeAgent.profile`.
2. Locate and set the following properties:
   - `introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer`

     Specifies the name of the SQL normalizer extension that will be used to override the preconfigured normalization scheme. When enabling the regular expressions extension, set this property to RegexSqlNormalizer.
   - `introscope.agent.sqlagent.normalizer.regex.keys=key1`

     This property specifies the regex group keys, which are evaluated in the order they are listed. This property is required to enable the regular expressions extension. There is no default value.
   - `introscope.agent.sqlagent.normalizer.regex.key1.pattern=A`

This property specifies the regex pattern that is used to match against the SQL statements. All valid regular expressions allowed by the System.Test.RegularExpressions namespace classes can be used here. This property is required to enable the regular expressions extension. There is no default value.

- `introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=B`

  This property specifies the replacement string format. All valid regex allowed by the System.Test.RegularExpressions namespace classes can be used here. This property is required to enable the regular expressions extension. There is no default value.

- `introscope.agent.sqlagent.normalizer.regex.matchFallThrough=false`

  If this property is set to true, SQL strings are evaluated against all the regex key groups. The implementation is chained. Hence, if the SQL strings match multiple key groups, the normalized SQL output from group1 is fed as input to group2, and so on.

  If the property is set to false, as soon as a key group matches the SQL string, the normalized SQL output from that group is returned. The MatchFallThrough property does not enable or disable the extension.

  For example, if you had a SQL string like: Select * from A where B, you would set the following properties:

  ```
  introscope.agent.sqlagent.normalizer.regex.keys=key1,key2
  introscope.agent.sqlagent.normalizer.regex.key1.pattern=A
  introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=X
  introscope.agent.sqlagent.normalizer.regex.key2.pattern=B
  introscope.agent.sqlagent.normalizer.regex.key2.replaceFormat=Y
  ```

  If `introscope.agent.sqlagent.normalizer.regex.matchFallThrough=`**`false`**, then the SQL is normalized against key1 regex. Output from that regex will be `Select * from X where B`. This SQL will be returned.

  If `introscope.agent.sqlagent.normalizer.regex.matchFallThrough=`**`true`**, then the SQL is normalized against key1 regex first. The output from that regex is `Select * from X where B`. This output is then fed to `key2 regex`. The output from `key2 regex` is `Select * from X where Y`. This will be the SQL returned.

» **Note** This property is not required to enable the regular expressions extension.

- `introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false`

  This property specifies whether the pattern match is case sensitive. The default value is false. This property is not required to enable the regular expressions extension.

- `introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false`

If this property is set to false, it will replace the first occurrence of the matching pattern in the SQL with the replacement string. If this property is set to true, it will replace all occurrences of the matching pattern in the SQL with the replacement string.

For example, if you have a SQL statement like Select * from A where A like Z, you would set the properties as follows:

```
introscope.agent.sqlagent.normalizer.regex.key1.pattern=A
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=X
```

If `introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=`**`false`**, it will result in a normalized SQL statement: `Select * from X where A like Z`.

If `introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=`**`true`**, it will result in a normalized SQL statement: `Select * from X where` **`X`** `like Z`.

The default value is false. This property is not required to enable the regular expressions extension.

» **Note** If none of the regular expression patterns match the input SQL, the RegexNormalizer will return a null string. The statement normalizer will then use the default normalization scheme.

3 Save the `IntroscopeAgent.profile`.

» **Important** All properties listed above are hot, meaning changes to these properties take effect once you have saved the IntroscopeAgent.profile. Changes to these properties do not require IIS restart.

Regular expression SQL statement normalizer examples

The three examples below can help you understand how to implement the regular expression SQL statement normalizer.

Example 1

Here is a SQL query before regular expression SQL statement normalization:

```
INSERT INTO COMMENTS (COMMENT_ID, CARD_ID, CMMT_TYPE_ID,CMMT_STATUS_ID,
  CMMT_CATEGORY_ID, LOCATION_ID, CMMT_LIST_ID,COMMENTS_DSC, USER_ID,
  LAST_UPDATE_TS) VALUES(?, ?, ?, ?, ?, ?,?, ''CHANGE CITY FROM CARROLTON,
  TO CAROLTON, _ ", ?, CURRENT)
```

Here is the desired normalized SQL statement:

```
INSERT INTO COMMENTS (COMMENT_ID, ...) VALUES (?, ?, ?, ?, ?, ?,?, CHANGE
  CITY FROM ( )
```

Here is the configuration needed to the `IntroscopeAgent.profile` file to result in the normalized SQL statement shown above:

```
introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=true
introscope.agent.sqlagent.normalizer.regex.keys=key1,key2
introscope.agent.sqlagent.normalizer.regex.key1.pattern=(INSERT INTO
COMMENTS \\(COMMENT_ID,)(.*)(VALUES.*)''(CHANGE CITY FROM \\().*(\\))
introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=$1 ...)
  $3$4 $5
introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false
introscope.agent.sqlagent.normalizer.regex.key2.pattern='[a-zA-Z1-9]+'
introscope.agent.sqlagent.normalizer.regex.key2.replaceAll=true
introscope.agent.sqlagent.normalizer.regex.key2.replaceFormat=?
introscope.agent.sqlagent.normalizer.regex.key2.caseSensitive=false
```

Example 2

Here is a SQL query before regular expression SQL statement normalization:

```
SELECT * FROM TMP_123981398210381920912 WHERE ROW_ID =
```

Here is the desired normalized SQL statement:

```
SELECT * FROM TMP_ WHERE ROW_ID =
```

Here is the configuration needed to the `IntroscopeAgent.profile` file to resultin the normalized SQL statement shown above:

```
introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=true
introscope.agent.sqlagent.normalizer.regex.keys=key1
introscope.agent.sqlagent.normalizer.regex.key1.pattern=(TMP_)[1-9]*
introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=$1
introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false
```

Example 3

If you want to normalize a SQL statement like: `Select .... ResID1, CustID1` where `ResID1=.. OR ResID2=.. n times OR CustID1=.. OR n times`, you could set the properties like this:

```
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=true
introscope.agent.sqlagent.normalizer.regex.keys=default,def
introscope.agent.sqlagent.normalizer.regex.default.pattern=(ResID)[1-9]
introscope.agent.sqlagent.normalizer.regex.default.replaceAll=true
introscope.agent.sqlagent.normalizer.regex.default.replaceFormat=$1
introscope.agent.sqlagent.normalizer.regex.default.caseSensitive=true
introscope.agent.sqlagent.normalizer.regex.def.pattern=(CustID)[1-9]
introscope.agent.sqlagent.normalizer.regex.def.replaceAll=true
introscope.agent.sqlagent.normalizer.regex.def.replaceFormat=$1
introscope.agent.sqlagent.normalizer.regex.def.caseSensitive=true
```

Command-line SQL statement normalizer

If the regular expression SQL normalizer is not in use, and you have SQL statements that enclose values in the where clause with double quotes (" "), use the following command-line command to normalize your SQL statements:

```
-DSQLAgentNormalizeDoubleQuoteString=true
```

» **Important** You can use the regular expressions SQL normalizer instead of this command to normalize SQL statements in double quotes. See *Regular expression SQL statement normalizer* on page 156 for more information.

Turning off statement metrics

Some applications may generate an extremely large number of unique SQL statements, causing a metric explosion in the SQL Agent. You can turn off SQL statement metrics in the SQL Agent.

» **Note** You will not lose backend or top-level JDBC metrics if you turn off statement metrics.

To turn off statement metrics:

1. Open the `sqlagent.pbd` file.
2. Remove **{sql}** from the trace directives you wish to turn off.
3. Save the `sqlagent.pbd` file.

Turning off Blame metrics

In a standard deployment of the SQL Agent, Blame metric data is collected by default. However, to reduce data overhead and reduce the number of metrics generated, you can turn Blame metric data off for the SQL Agent.

Note: If Blame metric generation is turned off, the SQL Agent data will not appear in Transaction Tracer viewer.

To turn off Blame metric data generation:

1. Open the `IntroscopeAgent.profile`.
2. Locate the property, `introscope.agent.sqlagent.useblame`.
3. Change the value to `false`:

   ```
   introscope.agent.sqlagent.useblame=false
   ```
4. Save your changes to the `IntroscopeAgent.profile`.
5. Restart the managed application.

SQL metrics

The SQL Agent metrics appear under the **Backends** node in the Introscope Workstation Investigator. SQL statement metrics can be found under the **Backends|<backendName>|SQL** node.

» **Note** `Average Response Time (ms)` will only display queries that return a data reader, i.e. queries executed via the `ExecuteReader()` method. This metric represents the average time spent in the data reader's `Close()` method.

Metric types specific to SQL data include:

- Connection Count—The number of live connection objects in memory.

  A connection is opened when a driver's `connect()` method is invoked, and closed when the connection invocation is closed via the `close()` method. The SQL Agent maintains weak references to Connections in a Set. When the Connection objects are garbage collected, the counts reflect the changes.

- Average Result Processing Time (ms)—The average processing time of a query.

  This metric represents the average time spent processing a ResultSet from the end of the `executeQuery()` call to the invocation of the ResultSet's `close()` method.

» **Note** Instrumented XADataSources may not report commit or rollback metrics. Other instrumented DataSources may not report commit or rollback metrics unless those metrics contain data.

CHAPTER 14

Enabling JMX Reporting

This chapter contains information about enabling the Java Agent to report JMX data.

Introscope Java Agent JMX support

Introscope can collect management data that application servers or Java applications expose as JMX-compliant MBeans, and present the JMX data in the Investigator metric tree.

Introscope supports any MBean built to the Sun JMX specification. For more information on the Sun JMX specification, see *http://java.sun.com/products/JavaManagement/*.

Introscope converts the JMX data to Introscope metric format and displays it in the Investigator under the following Resource:

```
<Domain>|<Host>|<Process>|<Agent>|JMX|
```

Introscope support for WebLogic 9.0 JMX metrics

WebLogic versions prior to WebLogic 9.0 provided only a single MBeanServer as a source of JMX metrics. WebLogic 9.0 provides three:

- RuntimeServiceMBean: per-server runtime metrics, including active effective configuration
- DomainRuntimeServiceMBean: domain-wide runtime metrics
- EditServiceMBean: allows user to edit persistent configuration

Introscope polls only the RuntimeServiceMBean, because it is the only one that supports local access (an efficiency issue), and because it contains most of the data expected to be relevant.

Default JMX metric conversion process

This section describes the process Introscope uses, by default, to convert JMX Metrics for display in the Investigator. This method is used to convert an MBean if:

- You use WebLogic 9.0, or
- You have not configured valid primary keys, as described in *Using primary key conversion to streamline JMX metrics* on page 165.

» **Note** If you specify primary keys that no MBeans match, Introscope will use the default conversion method.

In the default conversion method, Introscope displays both the name and the value of the attribute, and lists the pairs alphabetically in the metric tree.

```
Domain>|<Host>|<Process>|<Agent>|JMX|<domain name>|
<key1>=<value1>|<key2>=<value2>:<metric>
```

For example, given an WebLogic MBean with these characteristics:

Domain name	Key/Value Pairs	Metric Names
WebLogic	category=server, type=jdbc	connections

If no primary keys are specified in `introscope.agent.jmx.name.primarykeys`, the MBean attributes in the table above would be converted to the following Introscope metric:

```
<Domain>|<Host>|<Process>|<Agent>|JMX|Weblogic|category=server|type=jdbc:connections
```

Note that the key/value pairs are displayed alphabetically in the Introscope metric.

Using primary key conversion to streamline JMX metrics

You can optionally configure the order in which metrics appear under the JMX node by defining, in the agent profile, a Primary Key—those parts of an MBean's ObjectName that uniquely identify it.

If you do not configure primary key conversion, Introscope converts the JMX data as described in *Default JMX metric conversion process* on page 164. With the default conversion, metrics are listed alphabetically under the JMX node in Investigator.

This method of converting JMX data to Introscope metrics results in streamlined metric names, and allows you to control order of key/value pair information in the generated metrics.

The behavior is configured in the `introscope.agent.jmx.name.primarykeys` property in the agent profile. Values in the `primarykeys` property should specify the parts of an MBeans JMX `ObjectName` that uniquely identify an MBean. For example, a WebLogic MBean's ObjectName contains a *Type* key that specifies the kind of MBean, and a *Name* key that specifies the name of the resource the MBean represents. The key/value pairs in an ObjectName can vary for different types of MBeans.

Introscope converts and presents the MBeans identified by value of the `introscope.agent.jmx.name.primarykeys` property according to these rules:

- Only the key value information is displayed, not the key name.
- Values are ordered in the sequence defined in the `primarykeys` property.
- Values are case-sensitive.

For example, given a WebLogic MBean with these characteristics:

Domain name	MBean ObjectName Key/ Value Pairs	Metric Names
WebLogic	category=server, type=jdbc	connections

If you configure:

```
introscope.agent.jmx.name.primarykeys=type,category
```

the connections attribute appears in the Investigator tree in this structure:

```
<IntroscopeDomain>|<Host>|<Process>|<Agent>|JMX|Weblogic|jdbc|server:co
  nnections
```

» **Note** WebLogic 9.0 does not have universally available primary keys, so for WebLogic 9.0 Introscope uses the key/value pair metric naming convention found in the Default Conversion Method described in *Default JMX metric conversion process* on page 164. As a result, the JMX Metric tree for WebLogic 9.0 will have a different structure than the metric tree for other WebLogic versions.

Managing metric volume with JMX filters

Defining JMX filters determines what JMX MBean information will be collected and displayed in Introscope. If no filters are set, *all* JMX MBean information will be reported by the agent to the Enterprise Manager, increasing system overhead.

Filters are set in the `introscope.agent.jmx.name.filter` property in the agent profile, `IntroscopeAgent.profile`. Filters are keywords, entered as comma-separated strings in the property. Introscope 6.1 and higher supports filter strings that contain the asterisk (*) and question mark (?) wildcard characters.

Introscope matches the filter strings to JMX-generated Introscope metrics. If it finds a match, the metrics that match are reported to Introscope.

To limit the volume of metrics returned, define filter strings as narrowly as possible. For instance, if you define a filter string that matches an MBean attribute that exists on multiple MBeans, metrics from *each* of those MBeans will be reported. If you are only interested in an attribute on selected MBeans, you can qualify the attribute name with the MBean name in your filter string.

For example, assume you wish to capture the `MessagesCurrentCount` attribute value for the `JMSDestinationRuntime` MBean.

If the fully qualified metric name for `MessagesCurrentCount` is:

```
*SuperDomain*|host-name|Process|Agent-name|JMX|comp-1|
JMSDestinationRuntime|comp-2:MessagesCurrentCount
```

define `introscope.agent.jmx.name.filter` in the `IntroscopeAgent.profile` as:

```
JMX|comp-1|JMSDestinationRuntime|comp-2:MessagesCurrentCount
```

JMX filters for WebLogic

In the `IntroscopeAgent.profile` file for WebLogic, the following keywords are already defined:

- ActiveConnectionsCurrentCount
- WaitingForConnectionCurrentCount
- PendingRequestCurrentCount
- ExecuteThreadCurrentIdleCount
- OpenSessionsCurrentCount

Configuring JMX reporting

How you configure Introscope to support JMX depends upon the application server you use. This section describes how to configure Introscope to collect and present JMX data from WebLogic Server and WebSphere 5.0.

To configure JMX reporting, you must complete the following steps in this order:

1. *Enable JMX support in the agent profile*.
2. *Define primary keys for JMX data conversion*.
3. *Define JMX filters*.
4. *Configure startup class or service*.
5. *Add permissions to Java 2 security policy (WebSphere 5.0.x)*.

These are the steps:

To	Do this
Enable JMX support in the agent profile	1 Shut down the managed application if it is running. 2 For WebSphere agents only, in `IntroscopeAgent.profile` set `introscope.agent.jmx.enable` to `true`. (The default value is `false` in the WebSphere agent profile.)

To	Do this

Define primary keys for JMX data conversion	**1** In `IntroscopeAgent.profile`, configure primary keys.
	■ For WebLogic 9.0, comment out:
	`introscope.agent.jmx.name.primarykeys`
	■ For other WebLogic versions, uncomment:
	`introscope.agent.jmx.name.primarykeys`
	■ For WebSphere 6.0, uncomment:
	`introscope.agent.jmx.name.primarykeys=J2EEServer, Application, j2eeType,JDBCProvider,name,mbeanIdentifier`
	2 If you modify the value of the property, values must be case-sensitive, and multiple keys must be separated by commas.
	3 Continue to the next step, *Define JMX filters*, in the next cell of this table.

Define JMX filters	**1** In `IntroscopeAgent.profile`, make sure the `introscope.agent.jmx.name.filter` property is uncommented. **2** Enter desired strings, separated by commas, in the property. In order for Introscope to properly match filtered strings, the strings must be spelled exactly and case sensitive **3** Save changes. **4** Restart the managed application.
Configure startup class or service	To enable JMX data, you must configure an Introscope startup class in WebLogic Server, or a custom service in WebSphere. For instructions, see *Configuring startup class for WebLogic 8.1 or 9.0* on page 130.

Add permissions to Java 2 security policy (WebSphere 5.0.x)	If you use WebSphere 5.0.x, add these lines to the `<WebSphere home>/properties/server.policy` file:
	`// permissions for Introscope JMX support`
	`grant codeBase "file:${was.install.root}/-" {`
	`permission com.tivoli.jmx.MBeanServerPermission "*";`
	`permission com.tivoli.jmx.MBeanPermission "*";`
	`permission com.tivoli.jmx.AllMBeanPermission "*";`
	`};`

Enabling JSR-77 data for WAS 6.x

This section provides instructions for configuring Introscope to collect, retain, and report metrics for JSR-77 JMX MBean objects under WebSphere 6.0 and later.

JSR-77, the J2EE Management Specification, abstracts the manageable parts of the J2EE architecture and defines an interface for accessing management information.

JSR-77 support requires a JVM version 1.4 or later. If the JVM is 1.4, the application server must also support JSR-77.

» Related Knowledge Base article(s): For more information about viewing JMX metrics on WAS 6.1, please see the Knowledge Base article *Viewing JMX Metrics on WAS 6.1*. This article has been updated with Introscope 8.0 specific information.

To enable JSR-77 support:

1. Shut down the managed application if it is running.
2. Configure a WebSphere Custom Service, as described in *Configuring a custom service in WebSphere 5.0, 6.0, or 6.1* on page 131.
3. In the `IntroscopeAgent.profile`, verify that:

   ```
   introscope.agent.jmx.enable=true
   ```
4. In the `IntroscopeAgent.profile`, enable JSR-77 by setting:

   ```
   introscope.agent.jmx.name.jsr77.disable=false
   ```
5. Configure the primary keys method of metric conversion by uncommenting this property in the `IntroscopeAgent.profile`:

   ```
   introscope.agent.jmx.name.primaryKeys=J2EEServer,Application,
   j2eeType,JDBCProvider,name,mbeanIdentifier
   ```

   » Note Only the `IntroscopeAgent.profile` provided with Introscope for WebSphere contains this property definition.

   For more information see *Using primary key conversion to streamline JMX metrics* on page 165.
6. To specify the JSR-77 Metrics to report, uncomment and set this property to identify desired metrics:

   ```
   introscope.agent.jmx.name.filter
   ```

   Although filtering is not required, it is highly recommended. For more information see *Managing metric volume with JMX filters* on page 166.
7. To specify specific Mbean attributes to exclude in JSR-77 metrics, uncomment this property, and update as desired to exclude additional attributes:

   ```
   introscope.agent.jmx.ignore.attributes=server
   ```

CHAPTER 15

Configuring Platform Monitoring

This chapter has instructions for configuring Introscope Platform Monitors.

Understanding platform monitors

Platform monitors enable the Java Agent to report system metrics, including CPU statistics, to the Enterprise Manager. Platform monitors are included with the Introscope Agent installers.

Platform monitors on all operating systems except Windows Server 2003 and AIX are automatically enabled upon Java Agent installation. Windows Server 2003 and AIX platform monitors require a minimal configuration to work.

Introscope can monitor these operating systems:

- Solaris
- Windows Server 2003
- Windows 2000 Professional/Server/Advanced Server/Datacenter Server
- Windows XP Professional
- AIX 4 or 5
- RedHat Enterprise Linux 3.0 or 4.0

The platform metrics generated are:

- `ProcessID`
- `Processor Count` - the number of CPUs
- `Utilization %` (process) - for the Java Agent process, the percentage of total capacity of all processors this process is using. Regardless of how many processors there are, this metric generates only one number.
- `Utilization %` (aggregate) - for this processor, its total utilization (as a percentage) by all processes in the system. Each processor is shown as a Resource in the Investigator tree.

Enabling platform monitors on Windows Server 2003

To run platform monitors on Windows Server 2003, you must have *admin* privileges.

Enabling platform monitors on AIX

To enable platform monitors on AIX:

1. After Java Agent installation, make sure these files are installed in the `wily/ext` directory:
   - introscopeAIX4Stats.jar

- libIntroscopeAIX4Stats.so
- introscopeAIX5Stats.jar
- libIntroscopeAIX5Stats.so

2 Install the Perfstat Library.

- AIX 5: Install the patch/fix **APAR IY30022** from IBM at:
  - http://www.ibm.com/support/docview.wss?uid=isg1IY30022
- AIX 4.3.3 and higher: A Perfstat Library has been created to work with AIX 4.3.3. Install the following packages from ftp://ftp.software.ibm.com/aix/fixes/v4/os:
  - bos.perf.libperfstat
  - bos.perf.perfstat
- AIX 4: Bring your system up to 4.3.3 and then install the above packages.

» **Note** Restart your machine to ensure the patches have taken effect.

Disabling platform monitors

To disable platform monitors on any platform, move the .jar file from the `/wily/ext` directory to another directory.

This table shows the location of platform monitor files installed with a Java Agent installer.

For this platform	The platform monitor files are located in
Solaris	■ wily/ext/introscopeSolarisAmd32Stats.jar ■ wily/ext/introscopeSolarisAmd64Stats.jar ■ wily/ext/introscopeSolarisSparc32Stats.jar ■ wily/ext/introscopeSolarisSparc64Stats.jar ■ wily/ext/libIntroscopeSolarisAmd32Stats.so ■ wily/ext/libIntroscopeSolarisAmd64Stats.so ■ wily/ext/libIntroscopeSolarisSparc32Stats.so ■ wily/ext/libIntroscopeSolarisSparc64Stats.so
Windows Server 2003 all Windows 2000 platforms XP Professional	■ wily\ext\introscopeWindowsStats.jar ■ wily\ext\introscopeWindowsStats.dll

For this platform	The platform monitor files are located in
AIX	■ wily/ext/introscopeAIX5PSeries32Stats.jar ■ wily/ext/introscopeAIX52PSeries64Stats.jar ■ wily/ext/introscopeAIX53PSeries64Stats.jar ■ wily/ext/libIntroscopeAIX5PSeries32Stats.so ■ wily/ext/libIntroscopeAIX52PSeries64Stats.so ■ wily/ext/libIntroscopeAIX53PSeries64Stats.so
RedHat Enterprise Linux	■ wily/ext/introscopeRedHatStats.jar ■ wily/ext/libIntroscopeRedHatStats.so

Troubleshooting platform monitoring

In most cases, the platform monitor successfully detects the operating system and runs if the operating system is supported. In rare cases where this does not occur, you can explicitly specify your operating system in the Java Agent profile to ensure that the platform monitor runs.

To specify your operating system in the IntroscopeAgent.profile:

1. Open `IntroscopeAgent.profile`.
2. Under the *Platform Monitor Configuration* heading, locate the `introscope.agent.platform.monitor.system` property and enter the value for your operating system. Acceptable values are:
   - `SolarisAmd32`
   - `SolarisAmd64`
   - `SolarisSparc32`
   - `SolarisSparc64`
   - `HP-UXItanium`
   - `HP-UXParisc32`
   - `AIX5PSeries32`
   - `AIX53PSeries64`
   - `AIX52PSeries64`

   For example: `introscope.agent.platform.monitor.system=SolarisAmd32`
3. Restart the managed application.

On Windows platforms, the Java Agent logfile will sometimes contain an error similar to the following:

```
11/28/06 08:29:55 AM PST [ERROR] [IntroscopeAgent] An error occurred polling
   for platform data
```

If the error is infrequent, it is likely caused by a transient error originating from Windows itself, and is harmless. On platforms other than Windows, or in the case that the error happens all the time, this error indicates something more serious and should be reported to CA support for Introscope.

CHAPTER 16

Configuring WebSphere PMI

This chapter has instructions for configuring the Introscope Agent to report WebSphere PMI metrics.

Java Agent support for WebSphere PMI

Introscope can provide WebSphere performance data by extracting WebSphere Performance Monitoring Infrastructure (PMI) metrics via the PMI interface provided with WebSphere 5.1 and higher.

You must first enable PMI data collection in WebSphere before the data will be available to Introscope. In WebSphere, all performance monitor settings are off by default.

These PMI metrics can then be displayed as Introscope metrics. Users can filter which metric categories to bring into Introscope, depending on their needs.

To enable PMI reporting:

- enable PMI in WebSphere
- enable PMI in the Introscope Agent profile
- configure an Introscope Custom Service in WebSphere

Enabling PMI in WebSphere

This section describes how to turn on WebSphere Performance Monitor Settings.

To enable PMI in WebSphere 6.0/5.0.x:

1. See your WebSphere 6.0/5.0.x documentation for instructions on enabling Performance Monitoring Settings, and enabling Performance Monitoring for each desired metric category.
2. If you are running WebSphere 5.0.x and 6.0, you must modify the Java 2 Security Policy for PMI. Edit the file, `<WebSphere home>/properties/server.policy` to include the lines:

```
// permissions for Introscope PMI support
  grant codeBase "file: <Introscope_Home directory>/-" {
  permission java.security.AllPermission;
  };
```

Using PMI with Introscope on z/OS

There are several ways to obtain additional WebSphere-specific performance metrics on z/OS. One solution is to use Wily's PowerPack for z/OS WebSphere product, which provides WebSphere-specific PBDs and metrics. This product uses Wily tracer technology and is a low overhead method of obtaining WebSphere-specific metrics, and does not require you to enable PMI in WebSphere for z/OS.

In addition, Introscope supports PMI on WebSphere z/OS, but this approach consumes more system resources.

Configuring PMI in Introscope

After you turn on Performance Monitoring Settings in WebSphere, you must enable PMI data collection in Introscope, and enable the metric categories you'd like to see reported.

To configure PMI collection, use the following steps:

1. Shut down your managed application.
2. Open the `IntroscopeAgent.profile`.
3. Locate the property, `introscope.agent.pmi.enable`, under the *WebSphere PMI Configurations* heading, and verify it is set to `true`.
4. Introscope can report data from the following high-level PMI metric categories. These categories are represented by commented-out properties under the *WebSphere PMI Configuration* heading. Four categories—`threadPool`, `servletSessions`, `connectionPool`, and `j2c`—are set to `true` by default.
   - WebSphere 5.0.x PMI Categories:

```
introscope.agent.pmi.enable.threadPool=true
introscope.agent.pmi.enable.servletSessions=true
introscope.agent.pmi.enable.connectionPool=true
introscope.agent.pmi.enable.j2c=true
introscope.agent.pmi.enable.bean=false
introscope.agent.pmi.enable.transaction=false
introscope.agent.pmi.enable.webApp=false
introscope.agent.pmi.enable.jvmRuntime=false
introscope.agent.pmi.enable.jvmpi=false
introscope.agent.pmi.enable.system=false
introscope.agent.pmi.enable.cache=false
introscope.agent.pmi.enable.orbPerf=false
introscope.agent.pmi.enable.j2c=false
introscope.agent.pmi.enable.webServices=false
introscope.agent.pmi.enable.wlm=false
```

5. For each high level metric category you want to report, enter a value of `true`.
6. Save the changes.
7. Restart the managed application.

Viewing WebSphere Agent PMI data

After you've enabled PMI collections in Introscope, available PMI metrics will be displayed in the following location in the Investigator tree:

`<Domain>|<Host>|<Process>|<Agent>|WebSpherePMI`

CHAPTER 17

Enabling WebLogic Diagnostic Framework

This chapter has instructions for configuring the Introscope Agent to report WebLogic Diagnostic Framework (WLDF) metrics.

Java Agent support for WebLogic Diagnostic Framework (WLDF)

The WebLogic Diagnostic Framework (WLDF) is a monitoring and diagnostic framework that defines and implements a set of services that run within the WebLogic Server® process and participate in the standard server life cycle. Using WLDF, you can create, collect, analyze, archive and access diagnostic data generated by a running server and the applications deployed within its containers. This data provides insight into the run-time performance of servers and applications and enables you to isolate and diagnose faults when they occur.

WLDF is a new feature in WebLogic 9.0. In previous releases of WebLogic Server, access to diagnostic data by monitoring agents—which were developed by customers or third-party tools vendors—was limited to JMX attributes, and changes to monitoring agents required server shutdown and restart. However, WLDF enables dynamic access to server data through standard interfaces, and the volume of data accessed at any given time can be modified without shutting down and restarting the server.

For more information on WLDF, see *http://e-docs.bea.com/wls/docs90/wldf_configuring/index.html*.

Understanding WLDF Metric conversion

Introscope WLDF Metric conversion is similar to that in JMX metric conversion. Where JMX MBeans have multiple Attributes (metrics), WLDF has a set of Data Accessors, each with multiple Columns (metrics). Introscope converts WLDF Columns to Introscope metrics.

Information in Data Accessors is defined by a domain name and one or more key/value pairs. Introscope converts this WLDF information into Introscope-specific metric format and displays it in the Investigator under the following Resource:

```
<Domain>|<Host>|<Process>|<Agent>|WLDF|
```

Introscope converts Data Accessor Columns using the following method:

- key and value information is displayed
- key/value pairs are placed in alphabetical order in the Introscope-generated metrics.

The following example shows the syntax used:

```
<Domain>|<Host>|<Process>|<Agent>|WLDF|<domain
  name>|<key1>=<value1>|<key2>=<value2>:<metric>
```

For example, this table shows the information for the BYTECOUNT Column from the HTTPAccessLog Data Accessor:

Domain name	Key/Value pairs	Metric names
WebLogic	Name=HTTPAccessLog, Type=WLDFDataAccessRuntime=Accessor, WLDFRuntime=WLDFRuntime	BYTECOUNT

The Data Accessor information in the table above would be converted to the following Introscope metric:

```
<Domain>|<Host>|<Process>|<Agent>|WLDF|Weblogic|Name=HTTPAccessLog
   |Type=WLDFDataAccessRuntime=Accessor|WLDFRuntime=WLDFRuntime:BYTECOUNT
```

Note that the key/value pairs are displayed alphabetically in the Introscope metric.

Enabling WLDF reporting

By default, WLDF reporting is not enabled in Introscope.

To obtain WLDF data:

1. Shut down the managed application if it is running.
2. Configure a WebLogic Startup Class, as described in *Application server management data* on page 130.
3. In the `IntroscopeAgent.profile`:

   ```
   introscope.agent.wldf.enable=true
   ```

APPENDIX A

Java Agent Properties

This appendix describes procedures and properties associated with the Java Agent. The procedures and properties are:

Configuring IntroscopeAgent.profile location

The agent refers to properties in the IntroscopeAgent.profile for its basic connection and naming properties. When you install an agent, the agent profile is installed in the <AppServerHome>/wily directory.

Introscope looks for the agent profile in these locations, in this sequence:

- location defined in the system property com.wily.introscope.agentProfile
- location defined in com.wily.introscope.agentResource
- *<working directory>/wily* directory

» **Note** When adding a path on a Windows machine, you must escape a backslash (\) with another backslash (each one doubled), such as C:\\Introscope\\lib\\Agent.jar.

To change the location of the IntroscopeAgent.profile:

1. Define the new location using one of these methods:
   - define a system property on the Java command line with the -D option to specify the full path to the location of the IntroscopeAgent.profile file: com.wily.introscope.agentProfile
   - Make the IntroscopeAgent.profile available in a resource on the classpath. Set com.wily.introscope.agentResource to specify the path to the resource containing the agent profile.

   » **Note** If you change the location of the IntroscopeAgent.profile, the AutoProbe log location will also have to be changed. For more infomation, see *Managing ProbeBuilder Logs* on page 117.
2. Move your ProbeBuilder directives (PBD and PBL files) to the same location as the agent profile—they are referenced relative to the profile location.

   If you use Sun ONE, you must add the new location of the agent profile to the Sun ONE server.xml file

To change the location of the IntroscopeAgent.profile for Sun ONE:

1. To add Introscope information to startup scripts for Sun ONE 7.0, log in as Administrator or Root.
2. Open the server.xml file, in <SunOne_Home>/domains/domain1/server1/config/
3. Add a line to the jvm-options stanza in server.xml:

```
<jvm-options>
-Dcom.wily.introscope.agentProfile=SunOneHome/wily/
  IntroscopeAgent.profile
</jvm-options>
```

Command-line property overrides

In Introscope 8.0, you can override specific properties of the Enterprise Manager, agents, Workstation, and WebView using the command line. With regard to the Java Agent, this is useful when you have a clustered environment with multiply copies of an agent being shared and you want to tailor some of the agent settings for each application being monitored.

These steps assume you have installed and configured an agent on the application server to be monitored, and that the agent successfully connects to the Enterprise Manager.

To override agent properties using the command line:

1. Open the file where you modified the Java command to start the agent.

   The location of this file varies depending on the application server you use in your environment. For more information, see *Configuring Access to Application Server Data* on page 129.

2. Add a `-D` command to override a property. For example, you can add the following command to make the agent also use the `weblogic-full.pbl` file:

   ```
   -Dintroscope.autoprobe.directivesFile=weblogic-full.pbl
   ```

   Place this command next to other `-D` commands in the open file.

   » **Note** When you use this command to override hot deployable properties, the property is no longer hot deployable. Also, if you modify the property at a later time in the configuration file, you will receive a warning message in the Workstation stating you modified an overridden property and your change will have no effect. To avoid this, remove the override command before modifying the property in a configuration file.

3. Save the file.

4. Restart the agent.

   In the example used above, you would now see the additional WebLogic metrics in the agent node in the Workstation.

   » **Important** System properties become part of the property space of Introscope properties, allowing things like `java.io.tmpdir` to be visible to anything using IndexedProperties.

Agent failover

If the Java Agent loses connection with its primary Enterprise Manager, these properties specify which Enterprise Manager the agent will failover to, and how often it will try to reconnect to its primary Enterprise Manager.

introscope.agent.enterprisemanager.connectionorder

Usage The connection order of backup Enterprise Managers the agent uses if it is disconnected from its default Enterprise Manager.

Options Names of other Enterprise Managers the agent can connect to.

Default default

Example `introscope.agent.enterprisemanager.connectionorder=DEFAULT`

Notes Use a comma separated list.

introscope.agent.enterprisemanager.failbackRetryIntervalInSeconds

Usage Number of seconds between attempts by the agent to reconnect to its primary Enterprise Manager.

Options

Default Commented out; 120

Example
```
#introscope.agent.enterprisemanager.failbackRetryIntervalI
  nSeconds=120
```

Notes

Agent HTTP tunneling

You can configure agents to send information using tunneling technology, enabling agents to connect to an Enterprise Manager remotely. To do this, the agent must be configured to connect to the Enterprise Manager's embedded Web server, where the HTTP tunneling Web service is hosted.

To configure HTTP tunneled communication in `IntroscopeAgent.profile` as a new agent connection, specify:

- The host name of machine running the Enterprise Manager—see *introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT* on page 196.
- The connection port to the Enterprise Manager Web server. This is the value for the `introscope.enterprisemanager.webserver.port` property specified in the `IntroscopeEnterpriseManager.properties` for the Enterprise Manager to which the agent will connect. See the Introscope Properties Files section of the *Introscope Configuration and Administration Guide* for information about `introscope.enterprisemanager.webserver.port`.
- The HTTP tunneling socket factory. Specify this client socket factory:

  `com.wily.isengard.postofficehub.link.net.HttpTunnelingSocketFactory`

Agent HTTP tunneling—proxy server

These properties only apply to agents configured to tunnel over HTTP and must connect to an Enterprise Manager using a proxy server. For more information, see *Configuring a proxy server for HTTP tunneling* on page 38.

introscope.agent.enterprisemanager.transport.http.proxy.host

Usage Specify the proxy server host name.

Options

Default Commented out; not specified.

Example

Notes You must restart the managed application before changes to this property take effect.

introscope.agent.enterprisemanager.transport.http.proxy.port

Usage Specify the proxy server port.

Options

Default Commented out; not specified.

Example

Notes You must restart the managed application before changes to this property take effect.

introscope.agent.enterprisemanager.transport.http.proxy.username

Usage If the proxy server requires the agent to authenticate it, specify the username for authentication.

Options

Default Commented out; not specified.

Example

Notes You must restart the managed application before changes to this property take effect.

introscope.agent.enterprisemanager.transport.http.proxy.password

Usage If the proxy server requires the agent to authenticate it, specify the password for authentication.

Options

Default Commented out; not specified.

Example

Notes You must restart the managed application before changes to this property take effect.

Agent HTTPS tunneling

You can configure agents to send information using HTTPS, enabling agents to connect to an Enterprise Manager remotely.

introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT

Usage

Options

Default localhost

Example
```
introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT=localhost
```

Notes

introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT

Usage

Options

Default 8444

Example
```
introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT=8444
```

Notes

introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT

Usage

Options

Default
```
com.wily.isengard.postofficehub.link.net.HttpsTunnelingSocketFactory
```

Example
```
introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT=com.wily.isengard.postofficehub.link.net.HttpsTunnelingSocketFactory
```

Notes

Agent metric aging

Agent metric aging periodically removes dead metrics from the agent memory cache. A dead metric is a metric that has no new data reported in a given amount of time. This helps the agent improve performance and avoid potential metric explosions.

» **Note** A metric explosion happens when an agent is inadvertently set up to report more metrics than the system can handle. In this case, Introscope is bombarded with such a large number of metrics that performance gets very slow or the system cannot function at all.

Metrics that are in a group are removed only if all metrics in the group are considered candidates for removal. Currently, only `BlamePointTracer` group and `MetricRecordingAdministrator` metrics are removed as a unit; other metrics are removed individually.

The `MetricRecordingAdministrator` metric has APIs that can create a metric group. These APIs are:

- `getAgent().IAgent_getMetricRecordingAdministrator.addMetricGroup`
  String component, collection metrics. The component name is the metric resource name of the metric group. The metrics must be under the same metric node in order to qualify as a group. The metrics are a collection of `com.wily.introscope.spec.metric.AgentMetric` data structures. You can only add `AgentMetric` data structures to this Collection.
- `getAgent().IAgent_getMetricRecordingAdministrator.getMetricGroup`
  String component. Based on the component name which is the metric resource name, you can get the Collection of metrics.
- `getAgent().IAgent_getMetricRecordingAdministrator.removeMetricGroup`
  String component. The metric group is removed based on the component which is the metric resource name.
- `getAgent().IAgent_getDataAccumulatorFactory.isRemoved`
  Checks if the metric is removed. You use this API if you keep an instance of an accumulator in your extension. If the accumulator is removed because of metric aging then you will be holding onto a dead reference.

» Important » Important If you create an extension that uses a `MetricRecordingAdministrator` API (for example, for use with CA Wily product), be sure to delete your own instance of an accumulator. When a metric ages out because it has not been invoked, and then after a time data does become available for that metric, if you are using an old accumulator instance, the accumulator will not create new metric data points for that metric. To avoid this situation, do not delete your own instance of an accumulator and use instead the `getDataAccumulatorFactory` API.

Configuring agent metric aging

Agent metric aging is on by default. You can choose to turn off this capability using the property *introscope.agent.metricAging.turnOn* on page 192. If you remove this property from the IntroscopeAgent.profile, agent metric aging is turned off by default.

Agent metric aging runs on a heartbeat in the agent. The heartbeat is configured using the property *introscope.agent.metricAging.heartbeatInterval* on page 192. Be sure to keep the frequency of the heartbeat low. A higher heartbeat will impact the performance of the agent and Introscope.

During each heartbeat, a certain set of metrics are checked. This is configurable using the property *introscope.agent.metricAging.dataChunk* on page 193. It is also important to keep this value low, as a higher value will impact performance. The default value is 500 metrics to be checked per heartbeat. Each of the 500 metrics is checked to see if it is a candidate for removal. For example, if you set this property to check chunks of 500 metrics per heartbeat, and you have a total of 10,000 metrics in the agent memory, then it will take longer with lower impact on performance to check all 10,000 metrics. However, if you set this property to a higher number, you would check all 10,000 metrics faster, but with possibly high overhead.

A metric is a candidate for removal if the metric has not received new data after certain period of time. You can configure this period of time using the property *introscope.agent.metricAging.numberTimeslices* on page 193. This property is set to 3000 by default. If a metric meets the condition for removal, then a check is performed to see if all the metrics in its group are candidates for metric removal. If this requirement has also been met then the metric is removed.

» Note For metrics that do not have a metric group then the rule does not apply.

Based on the rules outlined above, it may take a significant amount of time for metrics to be removed.

Use the following properties to configure agent metric aging. In all of the properties, if any unrecognised values are used, the default value will be used instead.

introscope.agent.metricAging.turnOn

Usage	Turns on or off agent metric aging.
Options	`True or False`
Default	`True`
Example	`introscope.agent.metricAging.turnOn=true`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.metricAging.heartbeatInterval

Usage	The time interval when metrics are checked for removal, in seconds.
Options	
Default	`1800`

introscope.agent.metricAging.heartbeatInterval

Example	`introscope.agent.metricAging.heartbeatInterval=1800`
Notes	You must restart the managed application before changes to this property take effect.

introscope.agent.metricAging.dataChunk

Usage	During each interval, the number of metrics that are checked.
Options	
Default	`500`
Example	`introscope.agent.metricAging.dataChunk=500`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.metricAging.numberTimeslices

Usage	The number of intervals to check without any new data before making it a candidate for removal.
Options	
Default	`3000`
Example	`introscope.agent.metricAging.numberTimeslices=3000`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.metricAging.metricExclude.ignore.0

Usage	To exclude metrics from being removed. Add the metric name or metric filter to the list.
Options	comma seperated list; use the * wildcard
Default	
Example	`introscope.agent.metricAging.metricExclude.ignore.0=Threads*`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

Agent metric clamp

This property allows you to configure the Java Agent to approximately clamp the number of metrics sent to the Enterprise Manager. If the number of metrics pass this metric clamp value then no new metrics will be created.

introscope.agent.metricClamp

Usage	Configures the agent to approximately clamp the number of metrics sent to the Enterprise Manager.
Options	
Default	5000

introscope.agent.metricClamp

Example `introscope.agent.metricClamp=5000`

Notes
- If the property is not set then no metric clamping will occur. Old metrics will still report values.
- Changes to this property take effect immediately and do not require the managed application to be restarted.

Agent naming

The following are Java Agent naming properties. For more information on Java Agent naming, see *Java Agent Naming* on page 99.

introscope.agent.agentAutoNamingEnabled

Usage Specifies whether agent autonaming will be used to obtain the Java Agent name for supported application servers.

Options True or False

Default False

Example `#introscope.agent.agentAutoNamingEnabled=false`

Notes
- You must restart the managed application before changes to this property take effect.
- Requires the Startup Class to be specified for WebLogic; requires Custom Service to be specified for WebSphere.
- Set to true, and not commented out in agent profiles shipped with supported application servers

introscope.agent.agentAutoNamingMaximumConnectionDelayInSeconds

Usage Specifies the amount of time in seconds the agent waits for naming information before connecting to the Enterprise Manager.

Options

Default 120

Example
```
introscope.agent.agentAutoNamingMaximumConnectionDelayIn
  Seconds=120
```

Notes

introscope.agent.agentAutoRenamingIntervalInMinutes

Usage Specifies the time interval in minutes during which the agent will check to see if it has been renamed.

Options

Default 10

Example `introscope.agent.agentAutoRenamingIntervalInMinutes=10`

Notes

introscope.agent.disableLogFileAutoNaming

Usage Disables automatic naming of an agent's log files—the default behavior when an agent is configured for autonaming.

Options

Default

Example `introscope.agent.disableLogFileAutoNaming=`

Notes

introscope.agent.agentName

Usage Name of Agent.

Options For any installation, if the value of this property is invalid or if this property is deleted from the profile, the agent name will be `Unknown Agent`.

Default

Example `#introscope.agent.agentName=AgentName`

Notes

- In the agent profile provided with application server-specific agent installers, the default reflects the application server, for instance `WebLogic Agent`.
- In the agent profile provided with the default agent installer, the property value is `AgentName`, and the line is commented out.

introscope.agent.agentNameSystemPropertyKey

Usage Specifies which Java system property will contain agent name.

Options

Default Not specified.

Example `introscope.agent.agentNameSystemPropertyKey`

Notes You must restart the managed application before changes to this property take effect.

introscope.agent.clonedAgent

Usage

Options

Default false

Example `introscope.agent.clonedAgent=false`

Notes Set to true when running identical copies of an Application on the same machine.

introscope.agent.customProcessName

Usage Specify the process name as it should appear in the Introscope Enterprise Manager and Workstation.

Options

Default

introscope.agent.customProcessName

Example `introscope.agent.customProcessName=CustomProcessName`

Notes
- You must restart the managed application before changes to this property take effect.
- In the agent profile provided with application server-specific agent installers, the default reflects the application server, for instance "WebLogic."
- In the agent profile provided with default agent installer, the property is commented out.

introscope.agent.disableLogFileAutoNaming

Usage Specifies whether to disable automatic naming of agent log files when using AutoNaming options.

Options

Default false

Example `introscope.agent.disableLogFileAutoNaming=false`

Notes

Agent thread priority

The following property controls the priority of agent threads

introscope.agent.thread.all.priority

Usage Controls the priority of agent threads.

Options You can set this from 1 (low) to 10 (high).

Default Commented out;5.

Example `#introscope.agent.thread.all.priority=5`

Notes

Agent to Enterprise Manager connection

The following properties controls the agent connection to the Enterprise Manager.

introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT

Usage The host name of machine running the Enterprise Manager.

Options

Default `localhost`

Example

Notes

introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT

Usage The port on the Enterprise Manager machine that listens for the agent.

Options

introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT

Default `5001`

Example

Notes

introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT

Usage Change this property to use a different client socket factory.

Options

Default `com.wily.isengard.postofficehub.link.net.DefaultSocketFactory`

Example

Notes

AutoProbe

The following properties configure AutoProbe.

introscope.autoprobe.directivesFile

Usage Specifies Directives files for AutoProbe. For more information, see *ProbeBuilder Directives* on page 73.

Options

Default Varies by installer.

Example

Notes

introscope.autoprobe.enable

Usage

Options True or False

Default `true`

Example `introscope.autoprobe.enable=true`

Notes When this property is set to false, it turns off the automatic insertion of Probes into an application's bytecode. It does not turn off the agent or agent reporting.

introscope.autoprobe.logfile

Usage Name and location of AutoProbe log file.

Options

Default `AutoProbe.log`

Example `introscope.autoprobe.logfile=AutoProbe.log`

Notes

introscope.autoprobe.hierarchysupport.enabled

Usage For agents that run under JDK 1.5 using AutoProbe and dynamic instrumentation, you can use this property to enable instrumentation of classes that extend a supertype or interface.

Options True or False

Default `False`

Example `introscope.autoprobe.hierarchysupport.enabled=false`

Notes

introscope.autoprobe.hierarchysupport.runOnceOnly

Usage If you have enabled instrumentation of classes that extend a supertype or interface, you can use this property to control whether the utility that enables the feature runs only once, or at a specified interval.

Options True or False

Default `True`

Example `introscope.autoprobe.hierarchysupport.enabled=false`

Notes Logging properties related to dynamic instrumentation are defined in *Logging* on page 206.

Blame

The following property configures Boundary Blame.

introscope.agent.blame.type

Usage Add this property with a value of standard to turn off boundary blame. For information about boundary blame, see the *Introscope Workstation User Guide*.

Options

Default

Example

Notes

CPU utilization

The following property configures CPU utilization.

introscope.agent.disableAggregateCPUUtilization

Usage When multiple Java Agents are deployed on a single machine and CPU overhead is high, this property can be set to true to reduce overall CPU overhead on the machine.

Options True or False

Default `Uncommented; True`

Example `introscope.agent.disableAggregateCPUUtilization=true`

Notes

Cross-process tracing in WebLogic Server

The following property configures cross-process tracing in WebLogic Server.

introscope.agent.weblogic.crossjvm

Usage

Options True or False

Default `Commented out; True`

Example `#introscope.agent.weblogic.crossjvm=true`

Notes

Dynamic instrumentation

These properties enable changes to PBDs to take effect without restarting the application server or the agent process. This is a very CPU intensive operation, and it is highly recommended to use configuration to minimize the classes that are being redefined. PBD editing is all that is required to trigger this process.

introscope.autoprobe.dynamicinstrument.enabled

Usage Enables dynamic ProbeBuilding for agents that run under JDK 1.5, and use AutoProbe.

Options True or False

Default `False`

Example `introscope.autoprobe.dynamicinstrument.enabled=false`

Notes For more information about dynamic instrumentation, see *Dynamic ProbeBuilding* on page 56.

autoprobe.dynamicinstrument.pollIntervalMinutes

Usage For agents that run under JDK 1.5 using AutoProbe and dynamic instrumentation, this property determines the frequency with which the agent polls for new and changed PBDs.

Options

autoprobe.dynamicinstrument.pollIntervalMinutes

Default 1

Example autoprobe.dynamicinstrument.pollIntervalMinutes=1

Notes

introscope.autoprobe.dynamicinstrument.classFileSizeLimitInMegs

Usage Some classloader implementations have been observed to return huge class files.This is to prevent memory errors.

Options

Default 1

Example introscope.autoprobe.dynamicinstrument.classFileSizeLimitInMegs=1

Notes You must restart the managed application before changes to this property take effect.

introscope.autoprobe.dynamic.limitRedefinedClassesPerBatchTo

Usage Re-defining too many classes at a time might be very CPU intensive. In cases where the changes in PBDs trigger a re-definition of a large number of classes, this batches the process at a comfortable rate.

Options

Default 10

Example introscope.autoprobe.dynamic.limitRedefinedClassesPerBatchTo=10

Notes

ErrorDetector

The following properties configure interaction with ErrorDetector.

introscope.agent.errorsnapshots.enable

Usage Enable the agent to captures transaction details about serious errors.

Options True or False

Default True

Example

Notes

- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- Requires Introscope Error Detector.

introscope.agent.errorsnapshots.throttle

Usage The maximum number of error snapshots that the agent can send in a 15-second period.

Options

Default 10

introscope.agent.errorsnapshots.throttle

Example `introscope.agent.errorsnapshots.throttle=10`

Notes
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- Requires Introscope Error Detector.

introscope.agent.errorsnapshots.ignore.<index>

Usage This indexed property allows you to specify error messages to ignore. Error snapshots will not be generated or sent for errors with messages matching these filters. You may specify as many as you like (using .0, .1, .2 ...). You may use wildcards (*).

Options

Default Example definitions are provided, and commented out, as shown below. The following are examples only.

Example
```
#introscope.agent.errorsnapshots.ignore.0=*com.company.HarmlessException*
#introscope.agent.errorsnapshots.ignore.1=*HTTP Error Code: 404*
```

Notes
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- Requires Introscope Error Detector.

Extensions

The following property configures agent extensions.

introscope.agent.extensions.directory

Usage Specifies the location of all extensions to be loaded by the agent.

Options

Default `ext`

Example `introscope.agent.extensions.directory=ext`

Notes Non-absolute names are resolved relative to the location of the `IntroscopeAgent.properties` file.

JMX

The following properties configure JMX metrics.

introscope.agent.jmx.enable

Usage Enables collection of JMX Metrics.

Options True or False.

Default Varies by agent version.

Example

Notes

introscope.agent.jmx.ignore.attributes

Usage Controls which (if any) JMX MBean attributes are to be ignored. Create a comma-separated list of desired keywords.

Options

Default Commented out; `server`.

Example `#introscope.agent.jmx.ignore.attributes=server`

Notes If an MBean attribute name matches one on the list, the attribute will be ignored. Leave the list empty to include all MBean attributes.

introscope.agent.jmx.name.filter

Usage This property uses a comma-separated list of filter strings to determine what JMX data Introscope collects and displays.

Introscope reports JMX-generated metrics that match a filter string. Filter strings can contain the asterisk (*) and question mark (?) wildcard characters:

- * matches zero or more characters
- ? matches a single character.

To match a literal * or ?, escape the character with \\.

Examples:

- `ab\\*c` matches a metric name that contains `ab*c`
- `ab*c` matches a metric name that contains `abc, abxc, abxxc` etc.
- `ab?c` matches a metric name that contains `abxc`
- `ab\\?c` matches a metric names that contains `ab?c`

where `x` is any character.

Options

Default Commented out.

For WebLogic:

```
ActiveConnectionsCurrentCount,WaitingForConnectionCurrentC
  ount,PendingRequestCurrentCount,ExecuteThreadCurrentIdle
  Count,OpenSessionsCurrentCount,j2eeType
```

introscope.agent.jmx.name.filter

Example

```
#introscope.agent.jmx.name.filter=ActiveConnectionsCurrentCount,WaitingForConnectionCurrentCount,PendingRequestCurrentCount,ExecuteThreadCurrentIdleCount,OpenSessionsCurrentCount,j2eeType
```

Notes

introscope.agent.jmx.name.jsr77.disable

Usage This property controls whether or not Introscope collects and reports full JSR77 data, including complex JMX data.

Options True or False

Default `True`

Example

Notes To enable JSR 77 reporting, set this property to false.

The property `introscope.agent.jmx.name.jsr77.enable` was removed in Introscope 6.1.

introscope.agent.jmx.name.primarykeys

Usage User-defined order of MBean information, and simplifies name conversion.

Options

Default Commented out in default `IntroscopeAgent.profile` file.

Example

```
#introscope.agent.jmx.name.primarykeys=J2EEServer
```

Notes Options for WebLogic:

- Type
- Name

Comment out this property if using WebLogic Server 9.0.

Options for WebSphere:

- J2EEServer
- Application
- j2eeType
- JDBCProvider
- name
- mbeanIdentifier

LeakHunter

The following properties configure agent interaction with LeakHunter.

introscope.agent.leakhunter.collectAllocationStackTraces

Usage Specifies whether to collect allocation stack trace information.

Options True or False

Default False

Example

Notes
- Turning on this option has the potential to create higher system overhead, in CPU usage and memory.
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.

introscope.agent.leakhunter.enable

Usage Enables LeakHunter functionality.

Options True or False

Default True

Example

Notes Turning on this option has the potential to create higher system overhead, in CPU usage and memory.

introscope.agent.leakhunter.leakSensitivity

Usage Controls the sensitivity of LeakHunter.

Options Must be integer value from 1-10.

Default 5

Example `introscope.agent.leakhunter.leakSensitivity=5`

Notes Higher sensitivity will report more collections as potential leaks.

introscope.agent.leakhunter.leakSensitivity

Usage Controls the sensitivity of LeakHunter.

Options Must be integer value from 1-10.

Default 5

Example `introscope.agent.leakhunter.leakSensitivity=5`

Notes Higher sensitivity will report more collections as potential leaks.

introscope.agent.leakhunter.logfile.append

Usage Specifies whether to replace the log file or add information to an existing log file on application restart.

Options True or False

Default `False`

introscope.agent.leakhunter.logfile.append

Example `introscope.agent.leakhunter.logfile.append=false`

Notes
- `False` replaces the log file.
- `True` adds information to an existing log file.

introscope.agent.leakhunter.logfile.location

Usage Location of the `LeakHunter.log` file. The filename is relative to the directory that contains the agent profile.

Options

Default LeakHunter.log (This locates the log file in the same directory as the agent profile.)

Example
```
introscope.agent.leakhunter.logfile.location=
  LeakHunter.log
```

Notes If this property is commented out or left blank, no log file will be written.

introscope.agent.leakhunter.timeoutInMinutes

Usage Period (in minutes) during which Introscope LeakHunter looks for new potential leaks.

Options Must be a positive integer (no negative numbers).

Default 120

Example `introscope.agent.leakhunter.timeoutInMinutes=120`

Notes A value of zero means no timeout.

introscope.agent.leakhunter.ignore

Usage Use this to ignore any class matching any supplied patterns.

Options A comma-separated list of class matching patterns.

Default none

Example `introscope.agent.leakhunter.ignore=`

Notes
- Some collections cannot be used with LeakHunter. In order for a collection to be LeakHunter-safe, it must be safe to call `size()` at any time, from any thread.
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- You can used the "*" wildcard.

Logging

The following properties configure agent logging options.

log4j.appender.logfile.File

Usage Specifies the name and location of `IntroscopeAgent.log` file. The filename is relative to the directory that contains the agent profile.

Options

Default `IntroscopeAgent.log`

Example `log4j.appender.logfile.File=IntroscopeAgent.log`

Notes

log4j.logger.IntroscopeAgent

Usage Amount of logging detail for the `IntroscopeAgent.log`.

Options Level of detail value can be:

- `INFO`

or

- `VERBOSE#com.wily.util.feedback.Log4JSeverityLevel`

Destination value can be:

- `console`
- `logfile`
- both `console` and `logfile`

Default INFO, console, logfile

Example `log4j.logger.IntroscopeAgent=INFO,console,logfile`

Notes

log4j.logger.IntroscopeAgent.inheritance

Usage Controls log level and destination for log messages about classes that require instrumentation.

Options To configure logging of classes that have not been instrumented because they extend a supertype or interface, set this property to: `INFO, pbdlog`

For information about inheritance class logging see *Controlling directive logging* on page 60.

Default None

Example `log4j.logger.IntroscopeAgent.inheritance=INFO,pbdlog`

Notes

log4j.appender.pbdlog.File

Usage Identifies a log file for messages about classes that require instrumentation.

Options To configure logging of classes that have not been instrumented because they extend a supertype or interface set to: `pbdupdate.log`

Default None

log4j.appender.pbdlog.File

Example `log4j.appender.pbdlog.File=pbdupdate.log`

Notes

log4j.appender.pbdlog

Usage Specifies a package for logging messages about classes that require instrumentation.

Options To configure logging of classes that have not been instrumented because they extend a supertype or interface, set this property to: `com.wily.introscope.agent.AutoNamingRollingFileAppender`

Default None

Example
```
log4j.appender.pbdlog=com.wily.introscope.agent.AutoNamin
  gRollingFileAppender
```

Notes

log4j.appender.pbdlog.layout

Usage Specifies rules for logging messages about classes that require instrumentation.

Options To configure logging of classes that have not been instrumented because they extend a supertype or interface, set this property to: `com.wily.org.apache.log4j.PatternLayout`

Default None

Example
```
log4j.appender.pbdlog.layout=com.wily.org.apache.log4j.Pat
  ternLayout
```

Notes

log4j.appender.pbdlog.layout.ConversionPattern

Usage Specifies rules for logging messages about classes that require instrumentation.

Options To configure logging of classes that have not been instrumented because they extend a supertype or interface, set this property to: `%d{M/dd/yy hh:mm:ss a z} [%-3p] [%c] %m%n`

Default None

Example
```
log4j.appender.pbdlog.layout.ConversionPattern=%d{M/dd/yy
  hh:mm:ss a z} [%-3p] [%c] %m%n
```

Notes

log4j.additivity.IntroscopeAgent.inheritance

Usage Causes the directives for multiple level inheritance to be logged only in the `pbdupdate.log` file.

Options `True or False`

Default True

log4j.additivity.IntroscopeAgent.inheritance

Example `log4j.additivity.IntroscopeAgent.inheritance=true`

Notes To configure the logging of multiple level inheritance directives in the `pbdupdate.log` only, add this property to the agent profile and set to `false`.

Metric count

The following property affects where you will see the `Metric Count` metric in the Investigator. By default, this metric is no longer displayed under the `Agent Stats` node in the Investigator; it is now displayed as `Metric Count` under the `Custom Metric Agent` node.

If you want to see the `Metric Count` metric under the `Agent Stats` node, add this property to the `IntroscopeAgent.profile`.

introscope.ext.agent.metric.count

Usage

Options True or False

Default Not present in the `IntroscopeAgent.profile;` False

Example `introscope.ext.agent.metric.count=true`

Notes Add this property to the `IntroscopeAgent.profile` and set it to true to see the 'Metric Count' metric under the 'Agent Stats' node.

Platform monitoring

The following property configures platform monitoring metrics.

introscope.agent.platform.monitor.system

Usage Name of operating system to load a platform monitor for.

Options Solaris, RedHat2.1, RedHat3.0, Windows, AIX, Linux

Default Commented out; `Solaris`

Example `#introscope.agent.platform.monitor.system=Solaris`

Notes

Socket metrics

The following property configures socket metrics.

introscope.agent.sockets.reportRateMetrics

Usage Enables reporting of individual Socket's Input/Output Bandwidth rate metrics.

Options True or False

Default True

introscope.agent.sockets.reportRateMetrics

Example `introscope.agent.sockets.reportRateMetrics=true`

Notes

SQL Agent

The following properties configure aspects of the SQL Agent. For more information, see *Configuring the Introscope SQL Agent* on page 147.

introscope.agent.sqlagent.sql.useblame

Usage The following setting configures SQL Agent to optionally participate in the Introscope blame stack, thus creating blame metrics.

Options True or False

Default True

Example `#introscope.agent.sqlagent.sql.useblame=true`

Notes You must restart the managed application before changes to this property take effect.

introscope.agent.sqlagent.sql.maxlength

Usage Limits how much of a SQL statement appears in the Investigator tree for SQL Agent metrics, in bytes.

Options

Default 990

Example `introscope.agent.sqlagent.sql.maxlength=990`

Notes Does not appear in `IntroscopeAgent.profile`. To change the value, add the property to the agent profile.

The following property is used to set the custom SQL Agent normalizer extension:

introscope.agent.sqlagent.normalizer.extension

Usage Limits how much of a SQL statement appears in the Investigator tree for SQL Agent metrics, in bytes.

Options the name of the sql normalizer extension that will be used to override the preconfigured normalization scheme.

Default RegexSqlNormalizer

Example
```
introscope.agent.sqlagent.normalizer.extension=RegexSqlNor
  malizer
```

Notes If you use the default setting, you also must configure the regular expressions SQL statement normalizer properties below.

The following properties are used to set the regular expressions SQL statement normalizer:

introscope.agent.sqlagent.normalizer.regex.matchFallThrough

Usage This property if set to true will make sql strings to be evaluated against all the regex key groups.

Options True or False

Default false

Example
```
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=false
```

Notes Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.keys

Usage This property specifies the regex group keys.

Options

Default key1

Example
```
introscope.agent.sqlagent.normalizer.regex.keys=key1
```

Notes Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.pattern

Usage This property specifies the regex pattern that will be used to match against the SQL.

Options All valid regex allowed by `java.util.Regex` package can be used here.

Default .*call(.*\)\.FOO(.*\)

Example
```
introscope.agent.sqlagent.normalizer.regex.key1.pattern=.*call(.*\)\.FOO(.*\)
```

Notes Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.replaceAll

Usage This property if set to 'false' will replace the first occurrence of the matching pattern in the sql with the replacement string. If set to 'true' it will replace all occurrences of the matching pattern in the sql with replacement string.

Options True or False

Default false

Example
```
introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false
```

Notes Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat

Usage This property specifies the replacement string format.

Options All valid regex allowed by `java.util.Regex` package `java.util.regex.Matcher` class can be used here.

Default $1

Example
```
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=$1
```

Notes Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive

Usage This property specifies whether the pattern match is sensitive to case.

Options true or false

Default false

Example
```
introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false
```

Notes Changes to this property take effect immediately and do not require the managed application to be restarted.

SSL communication

The agent can connect to the Enterprise Manager over SSL. Use the following properties to configure that communication.

introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT

Usage

Options

Default localhost

Example
```
introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT=localhost
```

Notes

introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT

Usage

Options

Default 5443

Example
```
introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT=5443
```

Notes

ntroscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT

Usage

Options

Default com.wily.isengard.postofficehub.link.net.SSLSocketFactory

Example
```
ntroscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT=com.wily.isengard.postofficehub.link.net.SSLSocketFactory
```

Notes

introscope.agent.enterprisemanager.transport.tcp.truststore.DEFAULT

Usage Location of a truststore containing trusted Enterprise Manager certificates. If no truststore is specified, the agent trusts all certificates.

Options Either an absolute path or a path relative to the agent's working directory.

Default

Example
```
introscope.agent.enterprisemanager.transport.tcp.truststore.DEFAULT=
```

Notes On Windows, backslashes must be escaped. For example: C:\\keystore

introscope.agent.enterprisemanager.transport.tcp.trustpassword.DEFAULT

Usage The password for the truststore

Options

Default

Example
```
introscope.agent.enterprisemanager.transport.tcp.trustpassword.DEFAULT=
```

Notes

introscope.agent.enterprisemanager.transport.tcp.keystore.DEFAULT

Usage Location of a keystore containing the agent's certificate. A keystore is needed if the Enterprise Manager requires client authentication.

Options Either an absolute path or a path relative to the agent's working directory.

Default

Example
```
introscope.agent.enterprisemanager.transport.tcp.keystore.DEFAULT=c:\\keystore
```

Notes On Windows, backslashes must be escaped. For example: C:\\keystore

introscope.agent.enterprisemanager.transport.tcp.keypassword.DEFAULT

Usage The password for the keystore

Options

Default

Example
```
introscope.agent.enterprisemanager.transport.tcp.keypassword.DEFAULT=MyPassword768
```

Notes

introscope.agent.enterprisemanager.transport.tcp.ciphersuites.DEFAULT

Usage Set the enabled cipher suites.

Options A comma-separated list of cipher suites.

Default

Example
```
introscope.agent.enterprisemanager.transport.tcp.ciphersuites.DEFAULT=SSL_DH_anon_WITH_RC4_128_MD5
```

Notes If not specified, use the default enabled cipher suites.

Stall metrics

For more information on stall metric properties, see *Disabling the capture of stalls as Events* on page 146.

introscope.agent.stalls.thresholdseconds

Usage Specifies the minimum threshold response time at which time a transaction is considered stalled

Options

Default 30

Example
```
introscope.agent.stalls.thresholdseconds=30
```

Notes This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.

introscope.agent.stalls.resolutionseconds

Usage Specifies the frequency that the agent checks for stalls.

Options

Default 10

Example
```
introscope.agent.stalls.resolutionseconds=10
```

Notes This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.

introscope.agent.stalls.enable

Usage Controls whether the agent checks for stalls and creates events for detected stalls.

Options True or False

Default True

Example `introscope.agent.stalls.enable=true`

Notes

Transaction tracing

The following properties are for Transaction Tracing. For more information on Transaction Tracing, see *Configuring Transaction Trace Options* on page 141.

introscope.agent.transactiontracer.parameter.httprequest.headers

Usage Specifies (in comma-separated list) HTTP request header data to capture. Use a comma separated list.

Options

Default Commented out; `User-Agent`

Example
```
#introscope.agent.transactiontracer.parameter.httprequest.
  headers=User-Agent
```

Notes The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the desired header names.

introscope.agent.transactiontracer.parameter.httprequest.parameters

Usage Specifies (in comma-separated list) HTTP request parameter data to capture.

Options

Default Commented out; generic parameters.

Example
```
#introscope.agent.transactiontracer.parameter.httprequest.
  parameters=parameter1,parameter2
```

Notes The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the desired parameter names.

introscope.agent.transactiontracer.parameter.httpsession.attributes

Usage Specifies (in comma-separated list) HTTP session attribute data to capture.

Options

Default Commented out; generic parameters.

introscope.agent.transactiontracer.parameter.httpsession.attributes

Example
```
#introscope.agent.transactiontracer.parameter.httpsession.
  attributes=attribute1,attribute2
```

Notes The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the desired parameter names.

introscope.agent.transactiontracer.userid.key

Usage User-defined key string.

Options

Default Commented out; generic parameters.

Example
```
#introscope.agent.transactiontracer.parameter.httpsession.
  attributes=attribute1,attribute2
```

Notes The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the correct value if, in your environment, user IDs are accessed using `HttpServletRequest.getHeader` or `HttpServletRequest.getValue`.

For more information, see *introscope.agent.transactiontracer.userid.method*, below.

introscope.agent.transactiontracer.userid.method

Usage Specifies the method that returns User IDs. The Agent profile includes a commented out property definition for each of the three allowable values.

Uncomment the appropriate statement, based on whether user ID is accessed by getRemoteUser, getHeader, or getValue.

Options Allowable values are:

- `HttpServletRequest.getRemoteUser`
- `HttpServletRequest.getHeader`
- `HttpServletRequest.getValue`

Default Commented out; see options above.

Example The `IntroscopeAgent.profile` includes a commented out property definition for each of the three allowable values.

Notes

introscope.agent.transactiontrace.componentCountClamp

Usage Limiting the number of traces.

Options

Default 5000

introscope.agent.transactiontrace.componentCountClamp

Example `introscope.agent.transactiontrace.componentCountClamp=5000`

Notes
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- When the set limit is reached, warnings appear in the log, and the trace stops.

URL grouping

These properties are for configuring URL Groups for frontend metrics. For more information, see *Using URL groups* on page 134.

introscope.agent.urlgroup.keys

Usage Configuration settings for Frontend naming.

Options

Default `Default`

Example `introscope.agent.urlgroup.keys=default`

Notes If a URL address belongs to two URL Groups, the order in which you list the keys for the URL Groups in this property is important. The URL Group defined by the narrower pattern should precede the URL Group specified by the broader pattern.

For example, if the URL Group with key alpha contains a single address, and the URL Group with key beta includes all addresses on the network segment that contains the address in the first URL Group, alpha should precede beta in the keys parameter.

introscope.agent.urlgroup.group.default.pathprefix

Usage Configuration settings for frontend naming.

Options

Default `*`

Example `introscope.agent.urlgroup.group.default.pathprefix=*`

Notes

introscope.agent.urlgroup.group.default.format

Usage Configuration settings for Frontend naming.

Options

Default `Default`

Example `introscope.agent.urlgroup.group.default.format=default`

Notes

WebSphere PMI

The following properties configure WebSphere PMI metrics.

introscope.agent.pmi.enable

Usage Enables collection of data from WebSphere PMI.
Options True or False
Default True
Example `introscope.agent.pmi.enable=true`
Notes

introscope.agent.pmi.enable.bean

Usage Enables collection of PMI bean data.
Options True or False
Default False
Example `introscope.agent.pmi.enable.bean=false`
Notes

introscope.agent.pmi.enable.cache

Usage Enables collection of data about the effectiveness of WebSphere caching layers.
Options True or False
Default False
Example `introscope.agent.pmi.enable.cache=false`
Notes For WebSphere 5.0.x only.

introscope.agent.pmi.enable.connectionPool

Usage Enables collection of PMI connectionPool data.
Options True or False
Default True
Example `introscope.agent.pmi.enable.connectionPool=true`
Notes

introscope.agent.pmi.enable.j2c

Usage Enables collection of J2EE connector data.
Options True or False
Default False
Example `introscope.agent.pmi.enable.j2c=false`
Notes For WebSphere 5.0.x only.

introscope.agent.pmi.enable.jvmpi

Usage Enables collection of PMI jvmpi data.

Options True or False

Default False

Example `introscope.agent.pmi.enable.jvmpi=false`

Notes For data to be provided to this module, JVMPI must be turned on in WebSphere.

introscope.agent.pmi.enable.jvmRuntime

Usage Enables collection of PMI JVM runtime data.

Options True or False

Default False

Example `introscope.agent.pmi.enable.jvmRuntime=false`

Notes For data to be provided to this module, JVMPI must be turned on in WebSphere.

introscope.agent.pmi.enable.orbPerf

Usage Enables collection of performance statistics about the embedded Object Request Broker (ORB).

Options True or False

Default False

Example `introscope.agent.pmi.enable.orbPerf=false`

Notes For WebSphere 5.0.x only.

introscope.agent.pmi.enable.servletSessions

Usage Enables collection of PMI servletSessions data.

Options True or False

Default True

Example `introscope.agent.pmi.enable.servletSessions=true`

Notes

introscope.agent.pmi.enable.system

Usage Enables collection of operating system-level data.

Options True or False

Default False

Example `introscope.agent.pmi.enable.system=false`

Notes For WebSphere 5.0.x only.

introscope.agent.pmi.enable.threadPool

Usage Enables collection of PMI threadPool data.

Options True or False

Default True

introscope.agent.pmi.enable.threadPool

Example `introscope.agent.pmi.enable.threadPool=true`

Notes

introscope.agent.pmi.enable.transaction

Usage Enables collection of PMI transaction data.

Options True or False

Default False

Example `introscope.agent.pmi.enable.transaction=false`

Notes

introscope.agent.pmi.enable.webApp

Usage Enables collection of PMI webApp data.

Options True or False

Default False

Example `introscope.agent.pmi.enable.webApp=false`

Notes

introscope.agent.pmi.enable.webServices

Usage Enables collection of SOAP and web services data.

Options True or False

Default False

Example `introscope.agent.pmi.enable.webServices=false`

Notes For WebSphere 5.0.x only.

introscope.agent.pmi.enable.wlm

Usage Enables collection of Workload Management (WLM) data on load balancing and failover of WebSphere applications.

Options True or False

Default False

Example `introscope.agent.pmi.enable.wlm=false`

Notes For WebSphere 5.0.x only.

Wily CEM integration

For information on configuring Java Agents for Wily CEM Integration, see the *CA Wily CEM Integration Guide*.

WLDF metrics

The following properties configure WLDF metrics.

introscope.agent.wldf.enable

Usage	Enables collection of WLDF metrics.
Options	True or False
Default	False
Example	`introscope.agent.wldf.enable=false`
Notes	For WebSphere 5.0.x only.

APPENDIX B

Using the Introscope PBD Generator

You can use the Wily PBD Generator tool to instrument custom Java class files for use by Java Agents.

This section includes these topics:

About the Wily PBD Generator

The Wily PBD Generator utility can create a PBD file from Javadoc tags with which you have annotated your Java code, to facilitate the instrumentation of custom Java class files for use by the Java Agent.

The PBD Generator examines a set of Java source files, and instruments the methods in the classes that contain the Javadoc tag `@instrument`.

Using the PBD Generator tool, you can:

- automate building of PBD files, to eliminate potential for errors that might be introduced by creating PBD files manually.
- integrate PBD generation into your build systems to create and update PBD files automatically and incorporate any changes to the Java source.

You configure the PBD Generator by integrating it into an Apache Ant target using the `WilyPBDGenerator.jar` file, then running it as an Ant Javadoc task.

Configuring the PBD Generator

This tool is intended to be incorporated into Ant-based build systems, as a Javadoc task in an Ant target.

This sample Javadoc task illustrates the use of this tool in Ant:

```
<javadoc sourcepath="/src/engineering/products/introscope/source"
     destdir="/src/engineering/products/introscope/source/generatedpbd"
     maxmemory="512m"
     packagenames="com.wily.introscope.console.thornhill.ui.util"
     verbose="false"
     private="true">
   <doclet name="com.wily.util.build.javadoc.PBDInstrumentDoclet"
        path="/Wily/tools/WilyPBDGenerator.jar">
     <param name="-d" value="/src/engineering/products/introscope/source/
       generatedpbd"/>
   </doclet>
</javadoc>
```

Required PBD Generator parameters

These key PBD Generator parameters are required:

This parameter	Determines
sourcepath	the root directory of the Java source tree
destdir	the directory path of the PBD file that will be output from the tool

This parameter	Determines
packagenames	a comma-separated list of the Java packages to be examined for instrumentation
doclet path	the path to find the PBD Generator jar file, which contains this tool
param name="-d"	this must contain the same value as `destdir`

Using the PBD Generator

Before you can use the PBD Generator, you insert special Javadoc tags into the Java source files to be instrumented.

The syntax for the JavaDoc tag is:

```
@instrument <valid metric prefix> <optional tracer name>
```

where:

`<valid metric prefix>` is any valid Introscope metric prefix—a string without a colon character (:). Pipe characters (|) are acceptable.

`<optional tracer name>` can be BlamePointTracer, FrontendMarker or BackendMarker. The default is BlamePointTracer if the tracer name is missing.

APPENDIX C

Manual ProbeBuilding

This appendix provides instructions for manually instrumenting your applications. Manual ProbeBuilding is a non-dynamic method of instrumenting your applications.

Before you begin

When you run ProbeBuilder manually, it instruments classes on disk before the application server is run. You use manual ProbeBuilding when your environment does not support AutoProbe, or you prefer not to use AutoProbe.

» **WARNING** Introscope supports two other methods of instrumenting applications. CA Wily recommends you use these other methods before using manual ProbeBuilding. These methods are:

- Using JVM AutoProbe. See *Configuring ProbeBuilder options* on page 41 for more information..
- Using AutoProbe for application servers. See *AutoProbe for Application Servers* on page 63 for more information..

Manual ProbeBuilding **should not** be used with other methods of instrumentation, and *should be used as a last resort*.

Contact Wily Technical Support at 1-888-GET-WILY ext. 1 or support@wilytech.com if you are not sure you should use manual ProbeBuilding.

The instructions in this appendix assume you have performed these installation and configuration tasks:

☐ Installed the Java Agent. See *Installing the Java Agent* on page 22 for more information.

☐ Configured Java Agent connection properties. See *Configuring connection to the Enterprise Manager* on page 36 for more information.

☐ Configured the Java Agent name. See *Configuring the Java Agent name* on page 40 for more information.

☐ Configured options for ProbeBuilder. See *Configuring ProbeBuilder options* on page 41 for more information.

Manual ProbeBuilding options

There are two ways to instrument your bytecode manually:

- **The ProbeBuilder Wizard**—a GUI dialog for running ProbeBuilder. Follow the instructions in *Using the ProbeBuilder wizard* on page 227.
- **The Command-line ProbeBuilder**—A command-line interface for environments without a windowing system. Follow the instructions in *Using the command-line ProbeBuilder* on page 229.

Using the ProbeBuilder wizard

If your machine has a windowing environment, use the instructions in this section to add probes to your bytecode. These instructions assume that you have:

- Installed the Java Agent. See *Installing the Java Agent* on page 22 for more information.
- Selected the ProbeBuilder Wizard option from the Enterprise Manager installation process. See the *CA Wily Introscope Configuration and Administration Guide* for information on how to install the Enterprise Manager.
- Configured basic agent settings as described in *Installing and Configuring the Java Agent* on page 19.

To use the ProbeBuilder Wizard:

1. If you have custom PBDs, add them to the `<Introscope_Home>/config/custompbd` directory of the Enterprise Manager.

   » **Important** This directory is not the same as the `<Agent_Home>/wily/hotdeploy` directory used by the Java Agent to deploy custom PBDs. If you have custom PBDs in the `hotdeploy` directory and are now using the ProbeBuilder Wizard, you must copy the PBDs you want to use from the hotdeploy directory to the Enterprise Manager `config/custompbd` directory.

2. Launch the ProbeBuilder Wizard from your `<Introscope_Home>` directory. On the Welcome screen, click **Next**.
3. Enter or browse to your bytecode directory and click **Next**.

   » **Note** You can also select `.jar` files or individual `.class` files.

4. Click **Select Java Bytecode** to enter your desired directory and click **Next**.
5. Enter the name and location for the new directory to contain the instrumented code, or click **Browse** to select a location. The default name is the original directory with the suffix "isc."

   » **Note** If you select a directory that already exists, ProbeBuilder Wizard will display a dialog asking if you want to overwrite the directory. Click **Overwrite** to overwrite the existing files as necessary, or click **Cancel** to go back to the previous dialog to select another location.

6. Click **Next** if you did not overwrite an existing directory.
7. In the System Directives window, locate the set of system directives which correspond to your environment (if your application server isn't listed, use the **Default Java** selection) and click **Next**.

» **Note** You have a choice of either using system directives files in which most tracer groups are turned on (FULL) or only a subset of tracer groups are turned on (TYPICAL). For more information on full and typical system directives files and the what kind of information they provide, see *Default tracer groups and toggles files* on page 76.

8 If you installed custom directives files in your `config/hotdeploy` directory, they appear in the Custom Directives window. Check the box next to any custom directives you want to use with this bytecode, then click **Add Probes**.

» **Note** For information on creating custom directives, see *ProbeBuilder Directives* on page 73.

Introscope adds Probes to the specified bytecode. This operation may take several minutes.

9 When the **Finished** window opens, click **Exit** or **Add Additional Probes** to add Probes to another directory.

Update application startup script

After adding probes to the bytecode, update the application startup script to specify the location of the instrumented code and the Java Agent.

To specify the location of the instrumented code:

1 Edit the classpath of the application startup script to include locations of the directories containing the instrumented code created with the ProbeBuilder. Make sure this reference precedes the reference to the original code in the classpath.

2 Edit the classpath in the application startup script to include the path to the `<Introscope_Home>/lib/Agent.jar`.

For example, edit an existing classpath:

```
<your_application_path>/classes:/<your_application_path>/lib/app.jar
  MainClass
```

to look like this:

```
<your-applicationpath>.isc/classes:/<your-applicationpath>.isc/lib/
  app.jar:/Introscope_Home/lib/Agent.jar MainClass
```

3 Save the changes.

4 Start your application with the new startup script.

Using the command-line ProbeBuilder

If your machine does not have a windowing environment, use the instructions in this section to add probes to your bytecode.

In Introscope 8.0, the command-line ProbeBuilder has been migrated to Java Development Kit (JDK) 1.4.2. This affects users who:

- have a managed application running under JDK 1.3.
- cannot run their applications with AutoProbe.
- have never instrumented their applications before OR need to upgrade the Java Agent to 8.0.
- cannot install JDK 1.4 on their servers.

If one or more of the above conditions apply to your environment, use a version of ProbeBuilder from Introscope 7.1 or earlier.

Adding Probes to bytecode

The command-line ProbeBuilder is activated by the following Java command. This example uses the `/wily` directory, so paths are relative to the that directory. The syntax for the Java command depends on your Java version and other settings in your environment.

Note: If you are processing a very large `.jar` file, you may need to increase the memory in your JVM memory settings. Consult your JVM documentation for instructions.

The ProbeBuilder classpath varies, depending on whether you installed it using an agent installer or the full Introscope installer.

This is an example of invoking ProbeBuilder from the agent directory:

```
java -cp ext/ProbeBuilder.jar
  com.wily.introscope.api.IntroscopeProbeBuilder
  -directives default.pbl,stream.pbd
  -origdir /usr/myApp/classes
  -destdir /usr/myApp/classes.isc –verbose
```

This is an example of invoking ProbeBuilder from the Enterprise Manager directory:

```
java -cp lib/ProbeBuilder.jar
  com.wily.introscope.api.IntroscopeProbeBuilder
  -directives default.pbl,stream.pbd
  -origdir /usr/myApp/classes
  -destdir /usr/myApp/classes.isc –verbose
```

These are the command line ProbeBuilder commands

Option	Result
-help -h -?	Displays a help screen
-directives (or -pbd) comma-separated-file-list	REQUIRED: a comma-separated list of ProbeBuilder Directives files (`.pbd`), ProbeBuilder list files (`.pbl`), or directories to scan. Select either a full or typical .pbl file, depending on how much information you want gathered (see *Full or typical tracing options* on page 56). If you used the default Java Agent installer, you will see all possible default PBD and PBL files in the `\wily` directory. However, if you used an application-server specific Java Agent installer, you will only see PBD and PBL files specific to that application server.
Select one set of the next three pairs to specify your original code location and your instrumented code destination, using either directories, jar files, or classes.	
-origdir directory	Specifies the original directory (including subdirectories)
-destdir directory	Specifies the destination directory
-origjar file	Specifies the original archive, including `.jar`, `.zip`, `.war`, `.rar` and `.ear` archives
-destjar file	Specifies the destination archive
-origclass file	Specifies the original class
-destclass file	Specifies the destination class
-skipitems	Skips any files that are not Java bytecode files or archive files (`.jar` and `.zip` files). If **-skipitems** is not set, the non-Java bytecode files are copied to the destination, unchanged.
-prompt	Turns on an interactive text UI when problems arise.

Editing the classpath

After adding probes to the bytecode, update the classpath of the application startup script to reflect the locations of the instrumented code and the Java Agent.

To update the classpath:

1. Edit the classpath of the application startup script to include locations of the directories containing the instrumented code created with the ProbeBuilder. Make sure this reference precedes the reference to the original code in the classpath.
2. Edit the classpath in the application startup script to include the path to the `Agent.jar` file.

   For example, you might edit a classpath that looks like this:

```
<your_application_path>/lib/app.jar MainClass
```

to look like this:

```
<your-applicationpath>.isc/lib/app.jar:/<ApplicationServer_Home>/wily/
  Agent.jar MainClass
```

3. Save the changes.
4. Start your application with the new startup script.

Running instrumented code

There are three ways to point to instrumented code instead of your original code:

- In classpaths, replace original class paths with instrumented code paths.

  The instructions in this chapter directed you to perform this process when you instrumented your application for the first time.

- Prepend paths to classpaths.

  If only part of the application's code was instrumented, you could place the instrumented code paths before the original-code paths (prepend) in the classpath.

  If you do this, instrumented code loads and reports performance data. Non-instrumented code still loads and works normally, but does not report performance data.

- Place instrumented code in original classpath.

  Use this method when classpaths are set in many places, or to conduct an evaluation. Be careful using this method in a production environment, when with this method it is easy to forget whether you are using the original or the instrumented code.

  - Move the original code to a new location. Leave the classpaths unchanged. Then move the instrumented code to the original location.
  - On a UNIX machine, you could also create a symbolic link from the current location of the instrumented code to the original location.

Switching back to non-instrumented code

If you want to switch back to using non-instrumented code, undo the steps of modifying classpaths to run instrumented code, as described below:

- If you put the paths to your instrumented code into the Java classpaths, then replace paths to the instrumented code in Java classpaths with original paths.
- If you added paths to the instrumented code in front of the paths to the original code, remove prepended paths to classpaths

Remove the prepended portion of the classpath so that only the original classpath remains.

- If you put instrumented code in the original classpath, then remove the Instrumented code from the original path and place the original code in the original classpath.

  If you used symbolic links on a UNIX system, point the symbolic link to the original directory or remove the link and move the code into the original classpath.

The ProbeBuilder Wizard.lax file

The `ProbeBuilder Wizard.lax` file is located here:

`<Introscope home>/Introscope ProbeBuilder Wizard.lax`

lax.nl.current.vm

VM to use the next time the ProbeBuilder is started. Can be set to any installed JDK or Default: JRE version 1.3.

Varies by operating system

lax.stderr.redirect

Standard Error Output. Leave blank for no output, **console** to send to a console window, or any path to a file to save to the file.

Default: blank

lax.stdin.redirect

Standard Input. Leave blank for no input, **console** to read from the console window, or any path to a file to read from that file.

Default: blank

lax.stdout.redirect

Standard Output. Leave blank for no output, **console** to send to a console window, or any path to a file to save to the file.

Default: blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Z