AppDynamics Pro

Documentation

AppDynamics for Java

1. Tutorials for Java . . . 3
1.1 Overview Tutorials for Java . . . 3
1.1.1 Use AppDynamics for the First Time with Java . . . 3
1.2 Monitoring Tutorials for Java . . . 6
1.2.1 Tutorial for Java - Events . . . 6
1.2.2 Tutorial for Java - Flow Maps . . . 7
1.2.3 Tutorial for Java - Server Health . . . 11
1.2.4 Tutorial for Java - Transaction Scorecards . . . 14
1.3 Troubleshooting Tutorials for Java . . . 16
1.3.1 Super-Simple Java Troubleshooting using Events . . . 17
1.3.2 Tutorial for Java - Business Transaction Health Drilldown . . . 22
1.3.3 Tutorial for Java - Exceptions . . . 23
1.3.4 Tutorial for Java - Slow Transactions . . . 28
2. Install the App Agent for Java . . . 30
2.1 Multi-Agent Deployment for Java . . . 34
2.2 Java Server-Specific Installation Settings . . . 35
2.2.1 Apache Cassandra Startup Settings . . . 36
2.2.2 Apache Tomcat Startup Settings . . . 37
2.2.2.1 Tomcat as a Windows Service Configuration . . . 41
2.2.3 Glassfish Startup Settings . . . 42
2.2.4 IBM WebSphere Startup Settings . . . 44
2.2.4.1 App Agent for Java on z-OS or Mainframe Environments Configuration . . . 47
2.2.5 JBoss Startup Settings . . . 50
2.2.6 Jetty Startup Settings . . . 56
2.2.7 Oracle WebLogic Startup Settings . . . 56
2.2.8 OSGi Infrastructure Configuration . . . 60

2.2.9 Resin Startup Settings . . . 62
2.2.10 Solr Startup Settings . . . 64
2.2.11 Standalone JVM Startup Settings . . . 65
2.2.12 Tanuki Service Wrapper Configuration . . . 66
2.2.13 Tibco BusinessWorks Configuration . . . 66
2.3 Upgrade the App Agent for Java . . . 67
2.4 Uninstall the App Agent for Java . . . 67
3. Administer App Agents for Java . . . 68
3.1 App Agent for Java Configuration Properties . . . 68
3.2 App Agent for Java Diagnostic Data . . . 74
3.3 App Agent for Java Directory Structure . . . 76
3.4 App Agent for Java Performance Tuning . . . 77
3.5 Configure App Agent for Java for Batch Processes . . . 79
3.6 Configure App Agent for Java for JVMs that are Dynamically Identified . . . 80
3.7 Configure App Agent for Java in Restricted Environments . . . 81
3.8 Configure App Agent for Java in z-OS or Mainframe Environments . . . 81
3.9 Configure App Agent for Java on Multiple JVMs on the Same Machine that Serve Different Tiers . . . 83
3.10 Configure App Agent for Java on Multiple JVMs on the Same Machine that Serves the Same Tier . . . 85
3.11 Configure App Agent for Java to Use Existing System Properties . . . 87
3.12 IBM App Agent for Java . . . 89
3.13 Move an App Agent for Java Node to a New Application or Tier . . . 90
3.14 Troubleshoot App Agent for Java . . . 91
3.15 Administer App Agent for Java FAQ . . . 93
4. Configure AppDynamics for Java . . . 93
4.1 Configure Custom Exit Points (Java) . . . 93
4.1.1 Configurations for Custom Exit Points . . . 98
4.2 Code Metric Information Points (Java) . . . 101
4.3 Configure JMX Metrics from MBeans . . . 102
4.3.1 Exclude JMX Metrics . . . 108
4.3.2 Exclude MBean Attributes . . . 108
4.3.3 Configure JMX Without Transaction Monitoring . . . 109
4.3.4 Resolve JMX Configuration Issues . . . 109
4.3.5 Import or Export JMX Metric Configurations . . . 114
4.4 Configure Custom Memory Structures (Java) . . . 117
4.5 Configure Object Instance Tracking (Java) . . . 121
4.6 Configure Memory Monitoring (Java) . . . 123
4.7 Configure Multi-Threaded Transactions (Java) . . . 124
4.8 Configure Background Tasks (Java) . . . 125
4.9 Web Application Entry Points . . . 127
4.9.1 Servlet Entry Points . . . 128
4.9.1.1 Servlet Transaction Detection Scenarios . . . 137
4.9.1.1.1 Identify Transactions Based on DOM Parsing Incoming XML Payload . . . 137
4.9.1.1.2 Identify Transactions Based on POJO Method Invoked By a Servlet . . . 139
4.9.1.1.3 Identify Transactions for Java XML Binding Frameworks . . . 141
4.9.1.1.4 Identify Transactions Based on JSON Payload . . . 143
4.9.2 Struts Entry Points . . . 145
4.9.3 Web Service Entry Points . . . 146
4.9.4 POJO Entry Points . . . 148
4.9.5 Spring Bean Entry Points . . . 153
4.9.6 EJB Entry Points . . . 155
4.9.7 POCO Entry Points . . . 157
4.10 Getter Chains in Java Configurations . . . 161
5. Troubleshoot Java Application Problems . . . 163
5.1 Detect Code Deadlocks (Java) . . . 163
5.2 Troubleshoot Java Memory Issues . . . 165
5.2.1 Troubleshoot Java Memory Leaks . . . 165
5.2.2 Troubleshoot Java Memory Thrash . . . 170
6. Monitor Java Applications . . . 173
6.1 Monitor JVMs . . . 173
6.2 Monitor Java App Servers . . . 180
6.2.1 Monitor JMX MBeans . . . 181
6.3 Trace Multi-Threaded Transactions (Java) . . . 186

Tutorials for Java

This section provides tutorials for tasks in AppDynamics.

Troubleshooting Application Errors

Identifying and troubleshooting errors in your Java application.

Overview Tutorials for Java

Use AppDynamics for the First Time with Java

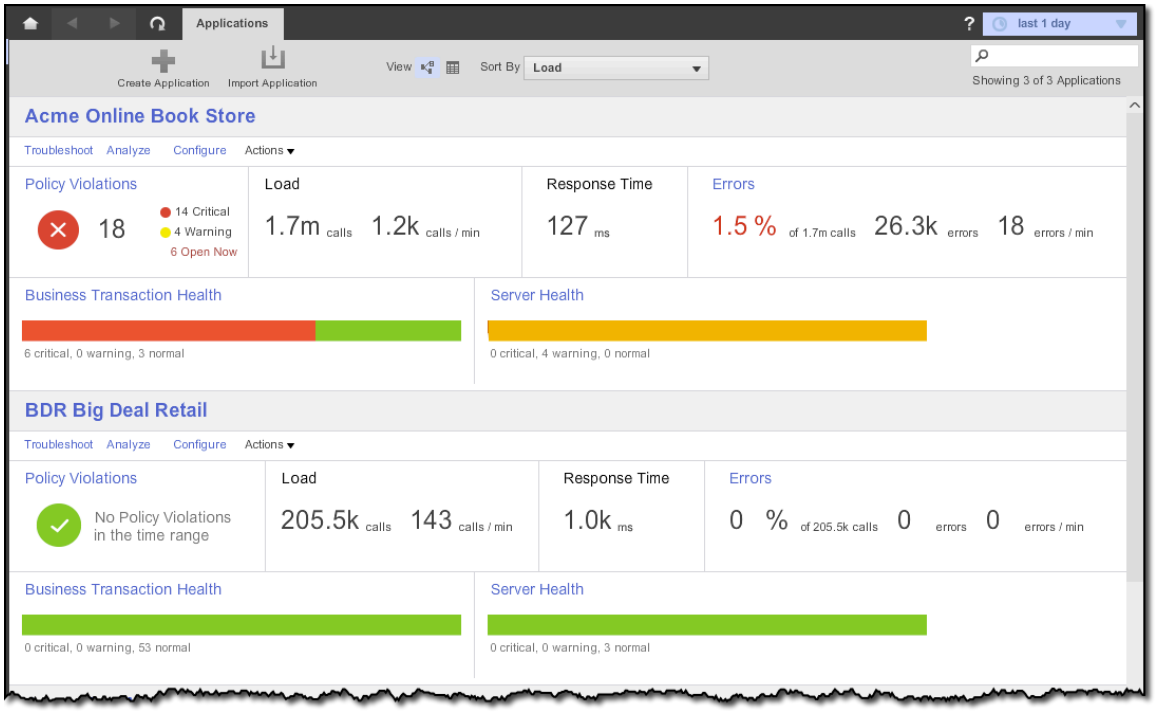
- All Applications Dashboard
- Application Dashboard
    - Time Range
    - Flow Map and KPIs
    - Events
    - Transaction Scorecard
    - Exceptions and Errors
- More Tutorials

This topic assumes that an application is already configured in AppDynamics, and uses the Acme Online application as the example. It also assumes that you have already logged in to AppDynamics.

This topic gives you an overview of how AppDynamics detects actual and potential problems that users may experience in your application - transactions that are slow, stalled or have errors - and helps you easily identify the root causes.

All Applications Dashboard

When you log into the Controller UI you see the All Applications dashboard.



The All Applications dashboard shows high-level performance information about one or more business applications. Load, response time, and errors are standard metrics that AppDynamics calls "key performance indicators" or "KPIs". The others are:

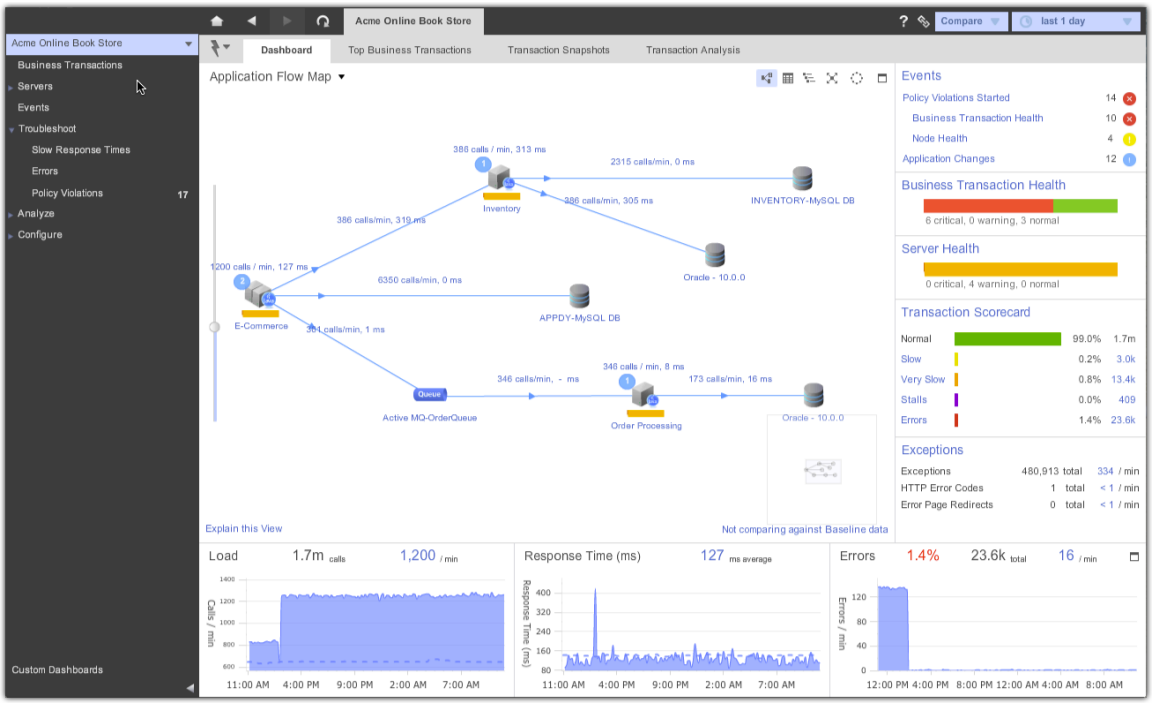
Health Rule Violations and Policies: AppDynamics lets you define a health rule, which consists of a condition or a series of conditions based on metrics exceeding predefined thresholds. You can then use health rules in policies to automate optional remedial actions to take if the conditions exist. AppDynamics also provides default health rules to help you get started.

Business Transaction Health: The health indicators are a visual summary of the extent to which a business transaction is experiencing critical and warning health rule violations. See the slow transactions tutorial.

Server Health: Additional visual indicators that track how well the server infrastructure is performing. See the server health tutorial.

Application Dashboard

Click an application to monitor, one that has some traffic running through it. The Application dashboard gives you a view of how well the application is performing.



You see the dashboard for your application. The flow map on the left gives you an overview of your servers (application servers, databases, remote servers such as message queues, etc.) and metrics for the calls between them. Click, hold and move the icons around to arrange the flow map. Use the scale slider and mini-map to change the view.

Time Range

From the time range drop-down in the upper-right corner select the time range over which to monitor - the last 15 minutes, the last couple of hours, the last couple of days or weeks. Try a few different time ranges and see how the dashboard data changes.

Flow Map and KPIs

In the flow map, click any of the blue lines to see more detail on the aggregated key performance metrics (load, average response time and errors) between the two servers. See the flow maps tutorial.

The graphs at the bottom of the dashboard show the key performance indicators over the selected time range for the entire application.

Events

An event represents a change in application state. The Events pane lists the important events occurring in the application environment. See the events tutorial.

Transaction Scorecard

The Transaction Scorecard panel shows metrics about business transactions within the specified time range, covering the percentage of instances that are normal slow, very slow, stalled or have errors. Slow and very slow transactions have completed. Stalled transactions never completed or timed out. Configurable thresholds define the level of performance for the slow, very slow and stalled categories. See the Transaction Scorecard tutorial.

Exceptions and Errors

An exception is a code-logged message outside the context of a business transaction. An error is a departure from the expected behavior of a business transaction, which prevents the transaction from working properly. See the exceptions tutorial.

More Tutorials

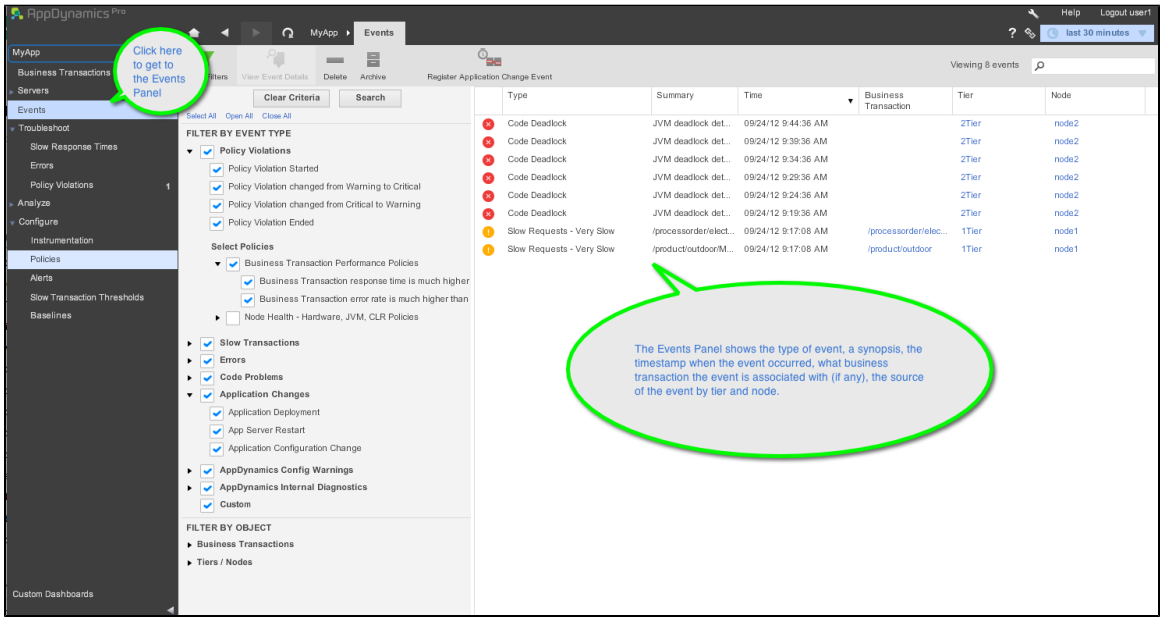
Monitoring Tutorials for Java

Tutorial for Java - Events

- Monitoring Events
- Filtering Events

Monitoring Events

1. From an application, tier or node dashboard, look at the Events panel:



2. The Events panel shows all the events that are monitored by AppDynamics. There are several types of events:

Health Rule Violation Events include business transaction health rule events such as average response time, and server health events such as Java VM Heap Utilization Thresholds. To change what triggers a health rule event, see Health Rules.

Slow Transaction Events fire when slow, very slow or stalled transactions occur. See Configure Thresholds.

Error Events trigger when application exceptions are thrown or HTTP Errors are returned. See Configure Error Detection.

Code Problem Events throw when a code deadlock is detected or a resource pool is completely utilized. For example this event fires when a JDBC connection pool is completely utilized, or a java.lang.Thread is deadlocked.

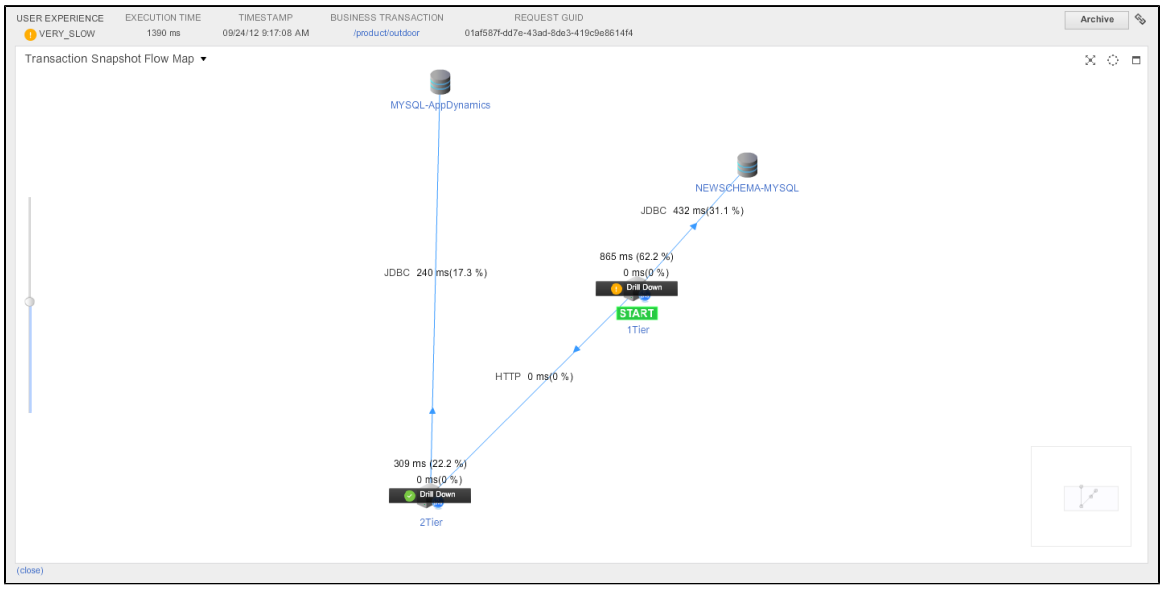
Application Change Events fire when administrative changes are made to an application tier. For example, this event is triggered when an application server instance is restarted, or when a Java VM option is modified on an application server. See Monitor Application Change Events.

AppDynamics Config Warnings occur when the AppDynamics infrastructure needs attention. For example, when the Controller detects low disk space conditions on its host this event type will fire. See AppDynamics Administration.

AppDynamics Internal Diagnostics help troubleshoot AppDynamics errors when working with AppDynamics Support. See AppDynamics Support.

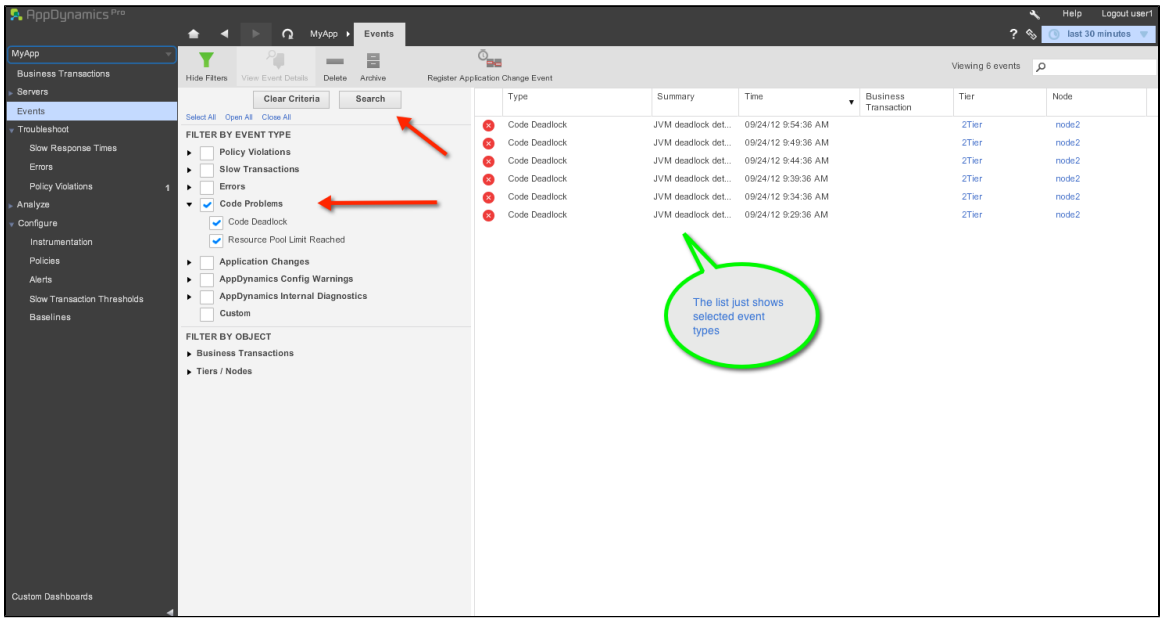
Custom events are user-defined events. See Events.

To get details click on the event in the list. For example, click on a slow request event to see the details of the request in the transaction flow map so you can start troubleshooting the slow request.



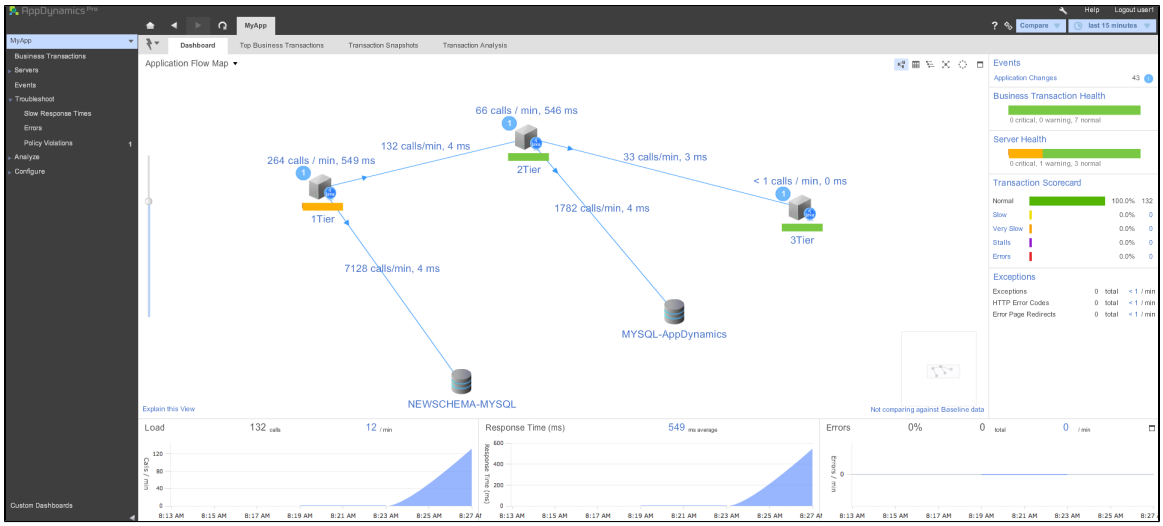
Filtering Events

You can filter events by type in the Events panel. Click on the type of event you want to see and click search. For example, to see only Deadlocks and Resource Pool Exhaustion events select the Code Problem Event and click Search.

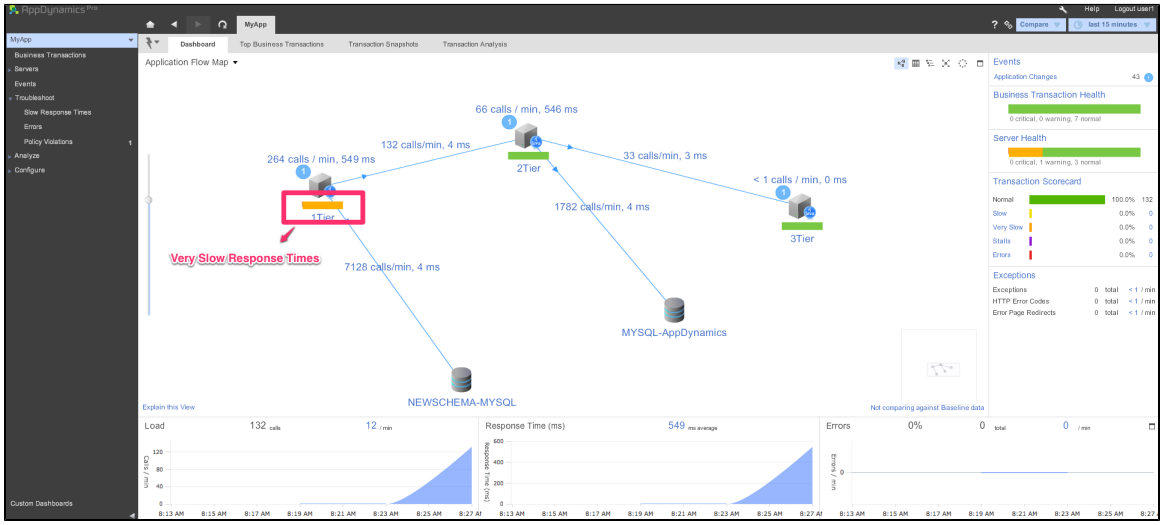


Tutorial for Java - Flow Maps

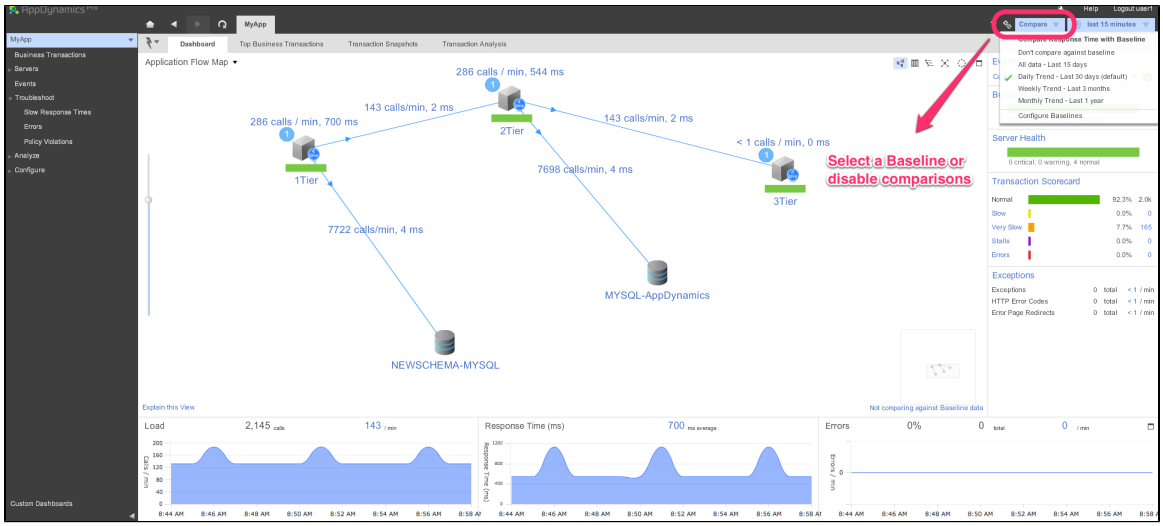
Flow maps show the health of your application, all tiers, and the communication between the tiers. It includes summary indicators such as call rates and error rates:



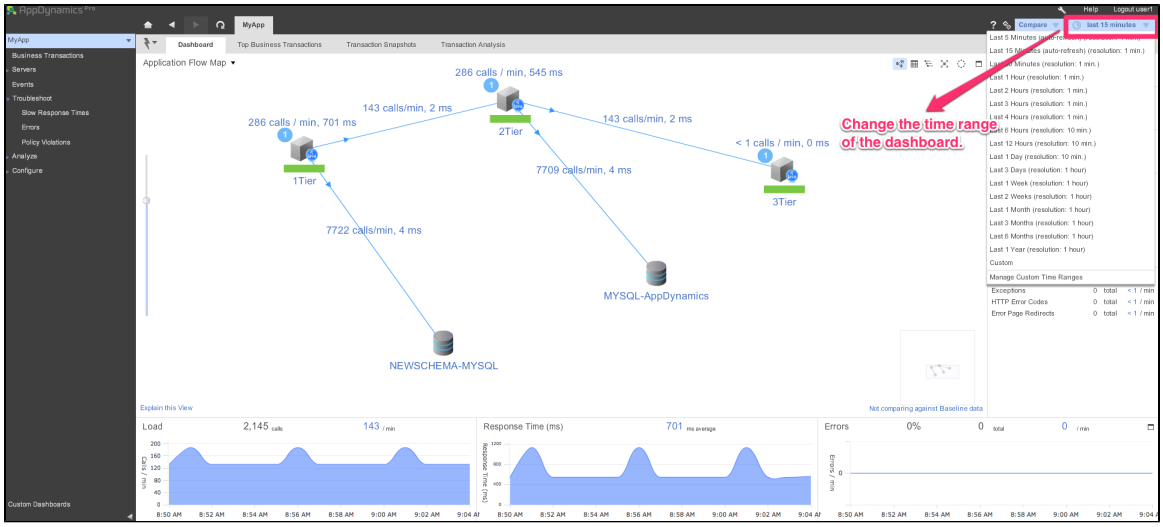
Visual indicators quickly show you problem tiers and healthy tiers. Below you can see tier 1 is experiencing very slow response times, while tier 2 and tier 3 are healthy:



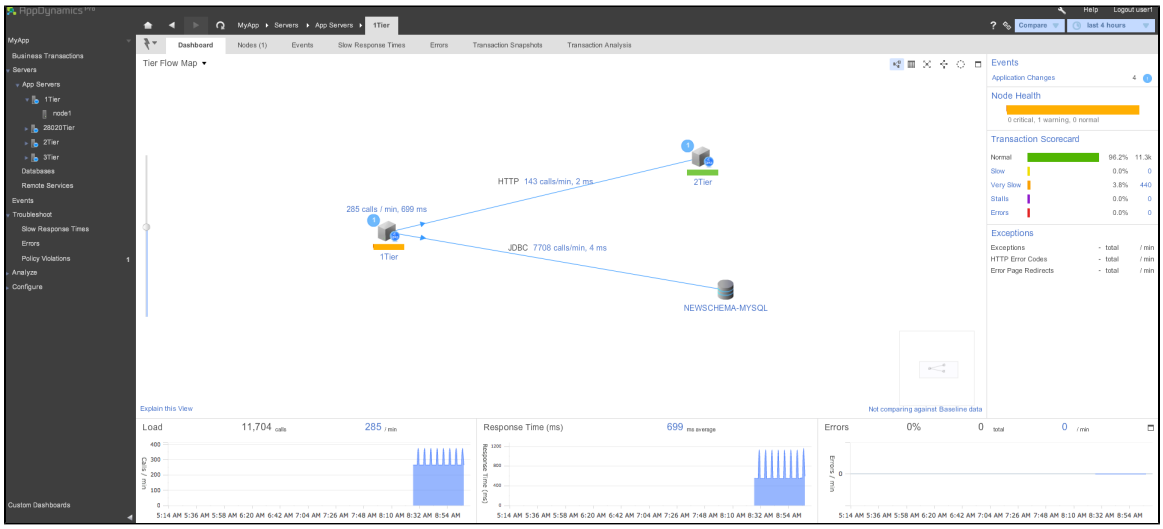
By default, the flow map computes tier health by comparing the state of the tier averaged over the last 15 minutes against the daily trend (the 30 day rolling average). You can change the time window for baseline comparison using the time window pull down menu. You can also disable baseline comparisons:



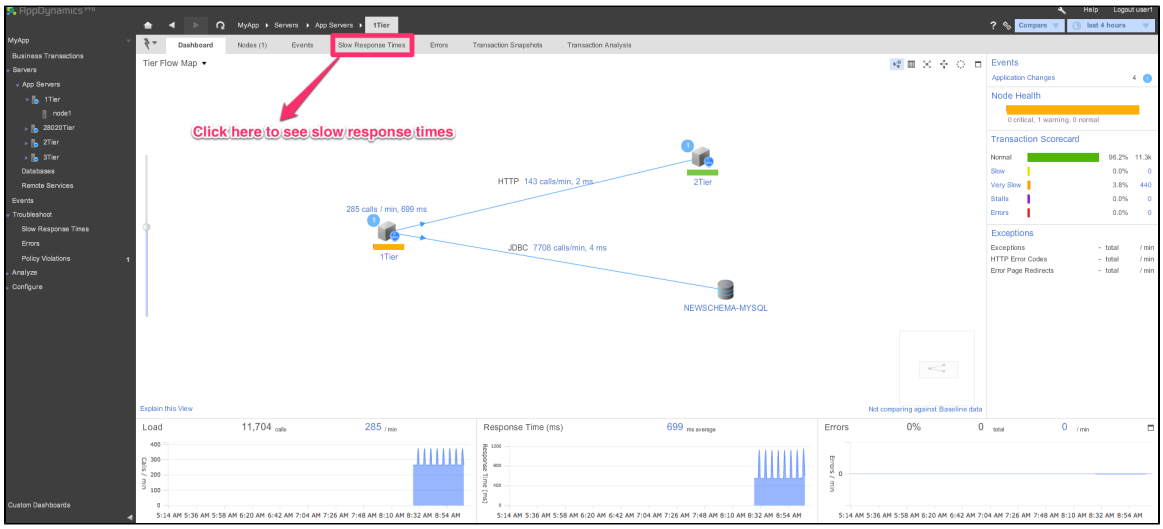
You can change the time range displayed in the flow map by changing the time window using the time window pull down menu. Changing the time range effects the entire dashboard:



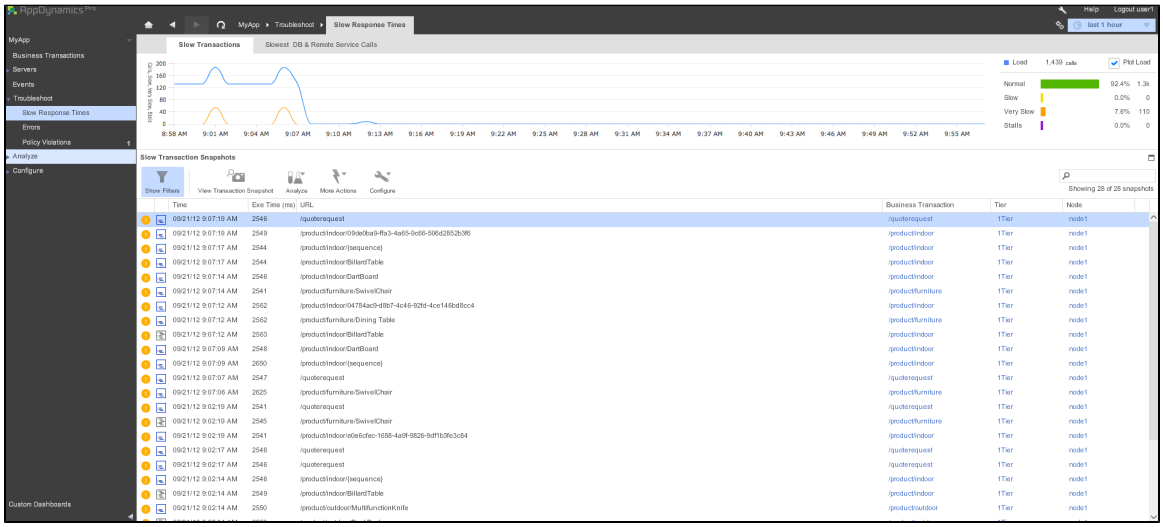
You can troubleshoot a problem system call by clicking on a the tier's name to drill down into a subset of the system involving the tier:



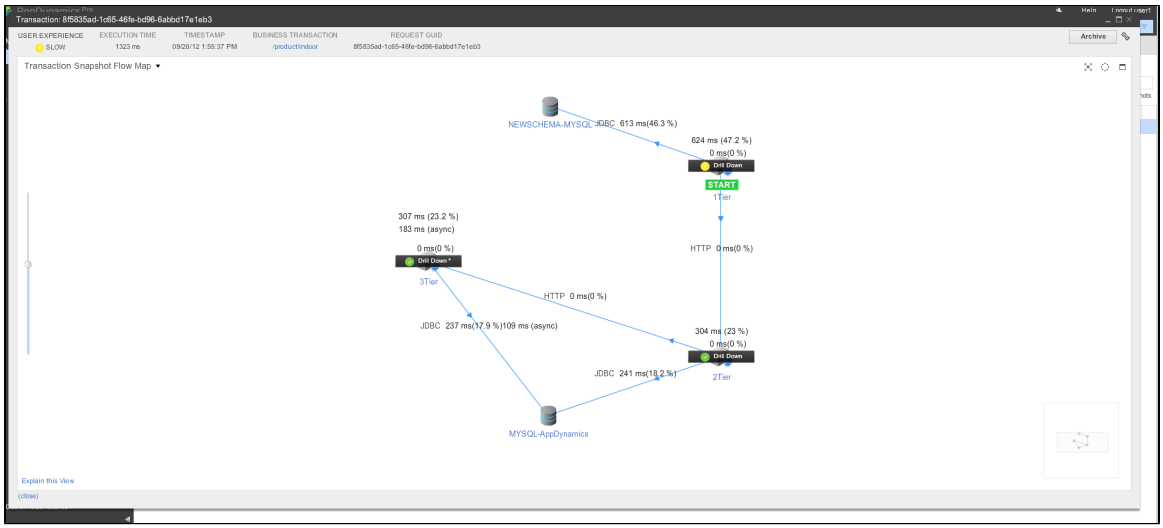
To view the slow response times in detail click on the slow response time menu:



From the slow response time pick a transaction to see a snapshot of the slow transaction:



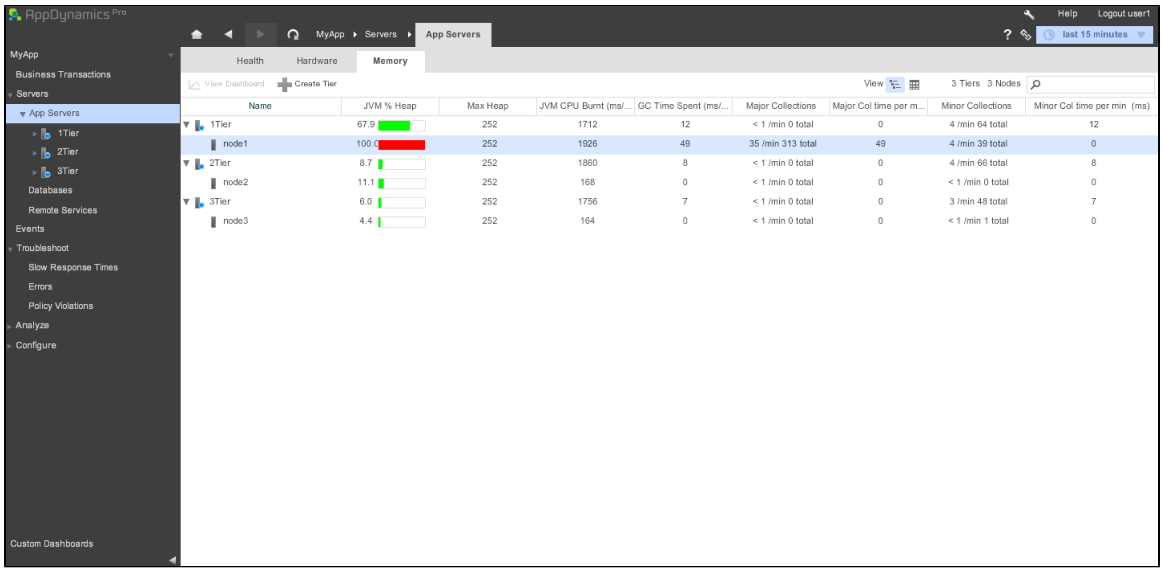
From the transaction snapshot you can troubleshoot the slow transaction:



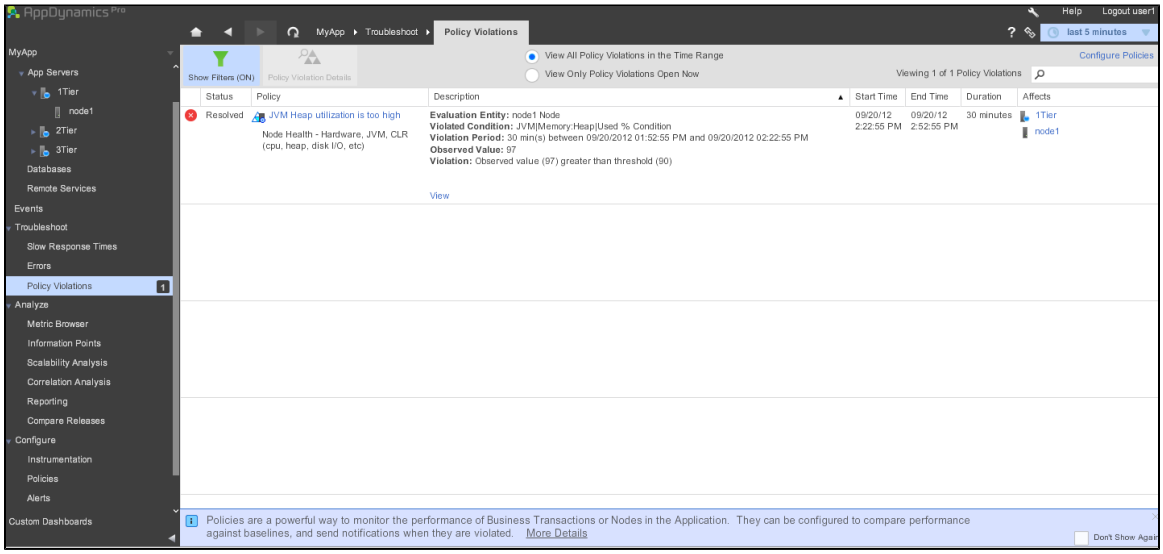
Tutorial for Java - Server Health

Node health is driven by node health policies. For example, node 1 is experiencing a JVM heap health rule violation --all of the heap is consumed.

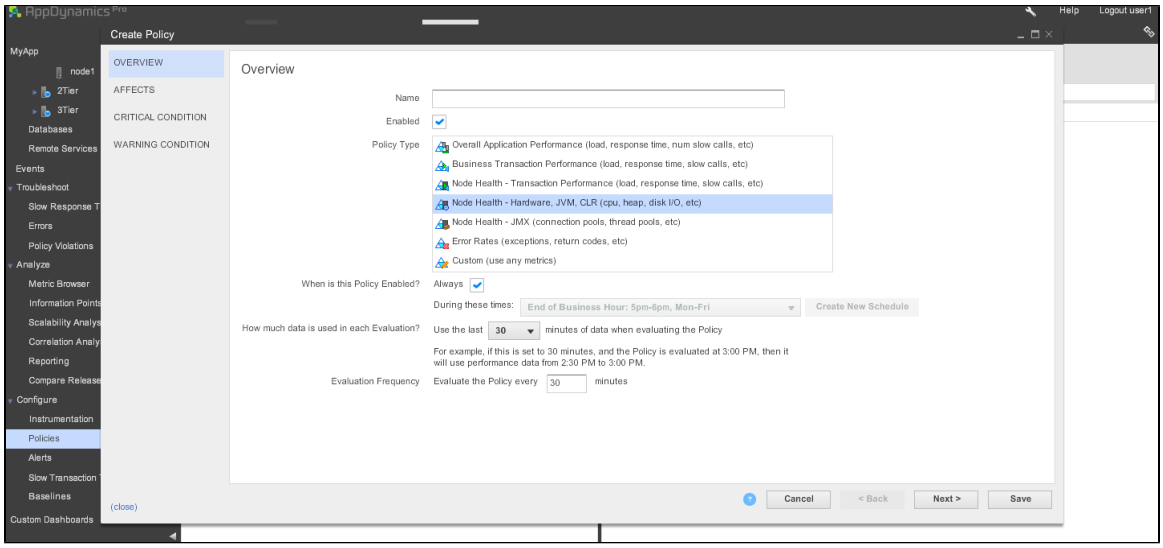
Out of the box, AppDynamics provides default policies for CPU utilization, physical memory utilization, JVM heap utilization, and CLR heap utilization. For example the default health rule for CPU utilization triggers a warning when a node exceeds 75% CPU utilization and a critical event fires when CPU utilization is 90% or above.



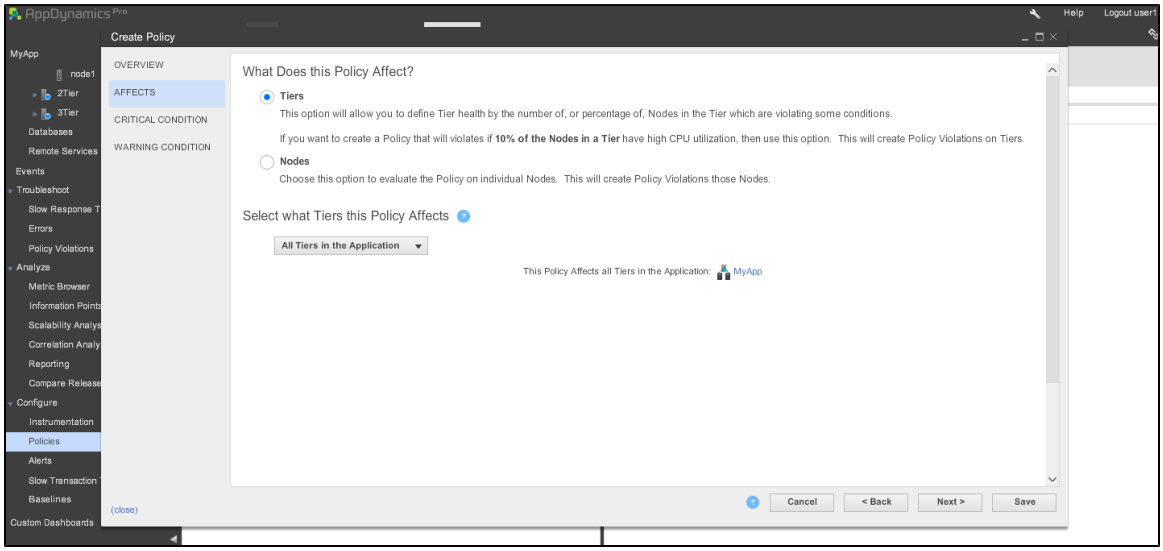
You can view the health rule violation details and the status of the violation:



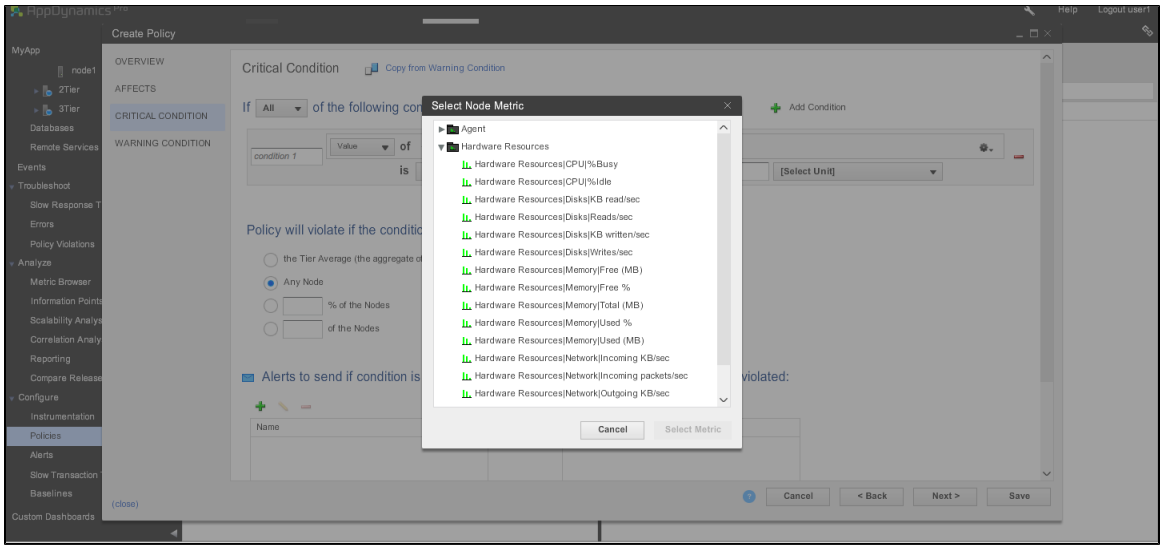
To change or add a new health rule see Configure Health Rules.



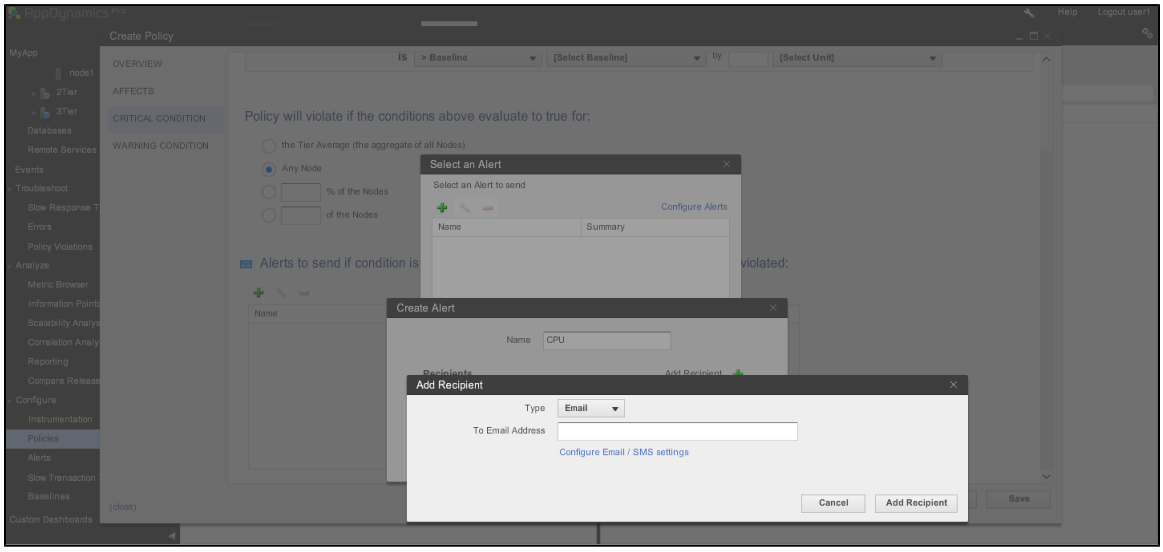
You can control the scope of a health rule. You can choose all nodes, of if you have a large cluster you may want this health rule to apply to a percentage of the nodes in a tier.



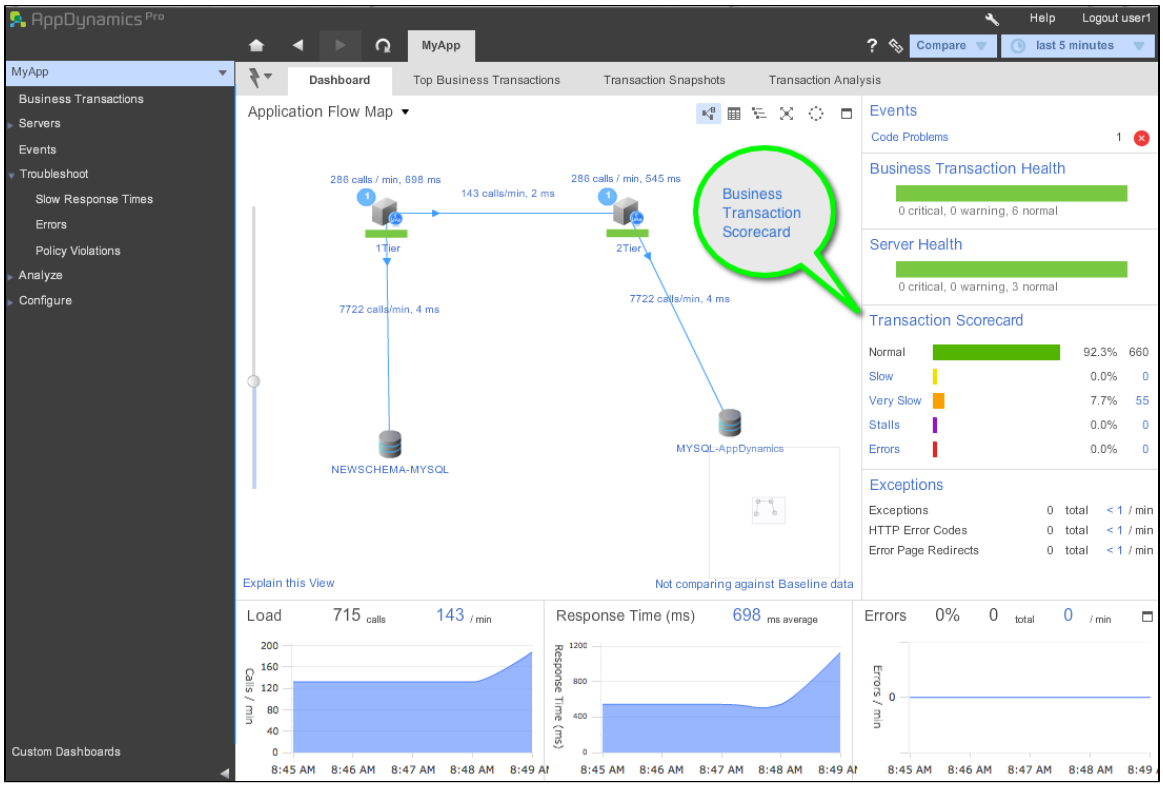
If you do not see a metric provided out of the box, you can Add Metrics Using Custom Monitors or create a JMX metric from MBeans.



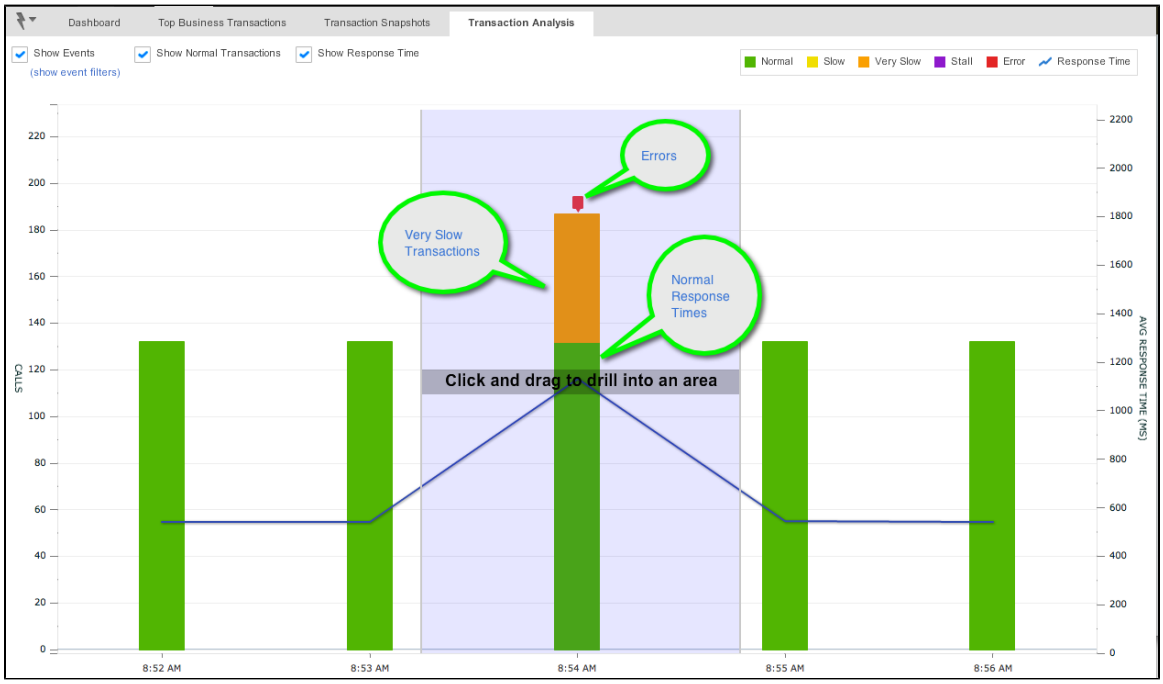
Health rule violations can be used in policies to trigger alerts that can notify Administrators via email or SMS systems.



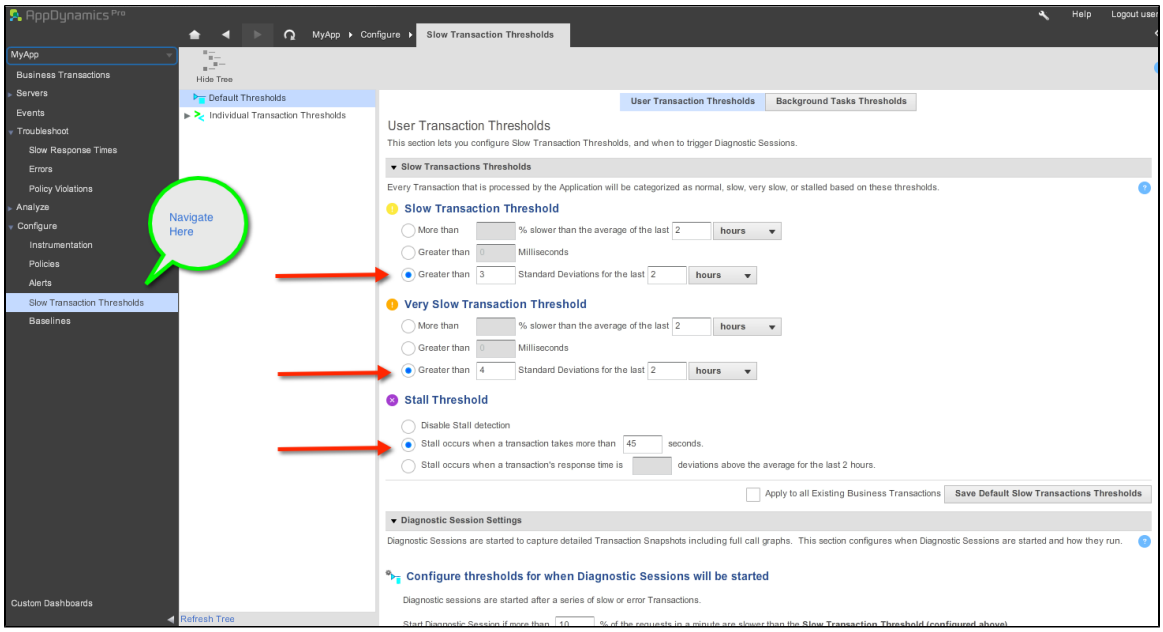
Tutorial for Java - Transaction Scorecards



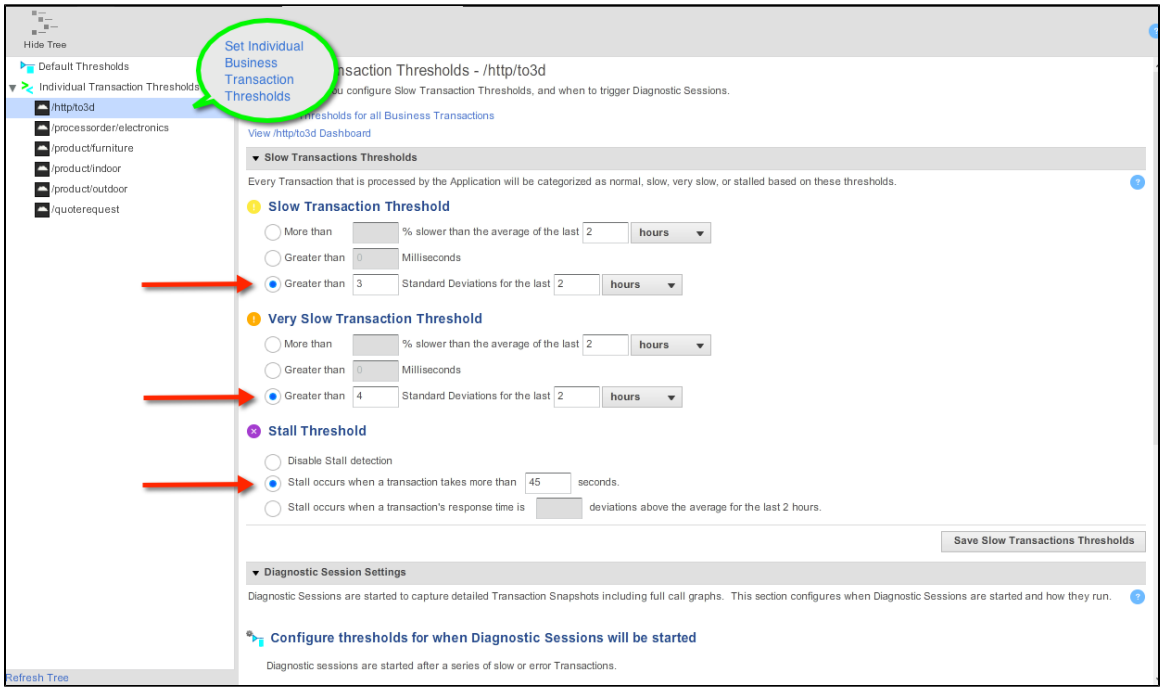
Transactions are categorized as Normal, Slow, Very Slow, Stalled, or Errors, which are determined by thresholds and the AppDynamics error detection subsystem. Thresholds can be static or dynamic; dynamic thresholds are based on historical data. The Transaction Analysis Histogram shows the distribution over time.



Default Transaction Baselines can be viewed here:



If you want to change the thresholds for all or individual transactions see the transaction threshold policy configurations (see below). See Thresholds.



To troubleshoot slow transactions, see Tutorial for Java - Slow Transactions and Troubleshoot Slow Response Times.

By Default, errors are determined when HTTP Error Codes are returned and by default AppDynamics instruments Java error and warning methods such as logger.warn and logger.error. AppDynamics captures the exception stack trace and automatically correlates it with the request. To learn how to change this, for example to turn down the "noise" or add redirect error pages, see Configure Error Detection.

Troubleshooting Tutorials for Java

Super-Simple Java Troubleshooting using Events

- Super-Simple Troubleshooting using the Events List
  - How to Set up the Events List
  - How to Know Something is Not Quite Right
  - How to Investigate
    - Drill down to an Error
    - Drill down to a stalled Business Transaction
    - Drill down to a slow or very slow Business Transactions
    - Drill down to an Application Server Exception
    - Drill down to a Code Deadlock
    - Drill down to Application Change Events

Super-Simple Troubleshooting using the Events List

How to Set up the Events List

1. From an application dashboard, click Events (either the menu item or the Events pane label).

2. In the Events window, use the filter criteria to pick which events you want to monitor. Click Search.

3. Set the time range.

4. Watch and look.

How to Know Something is Not Quite Right

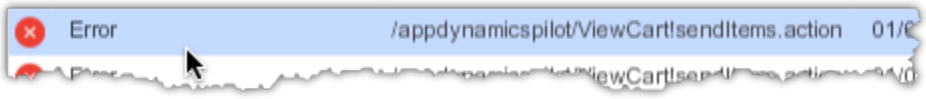
You see:
Red (critical, policy violation)
Purple (warning, stall)
Orange (warning, very slow)
Yellow (warning, slow)

How to Investigate

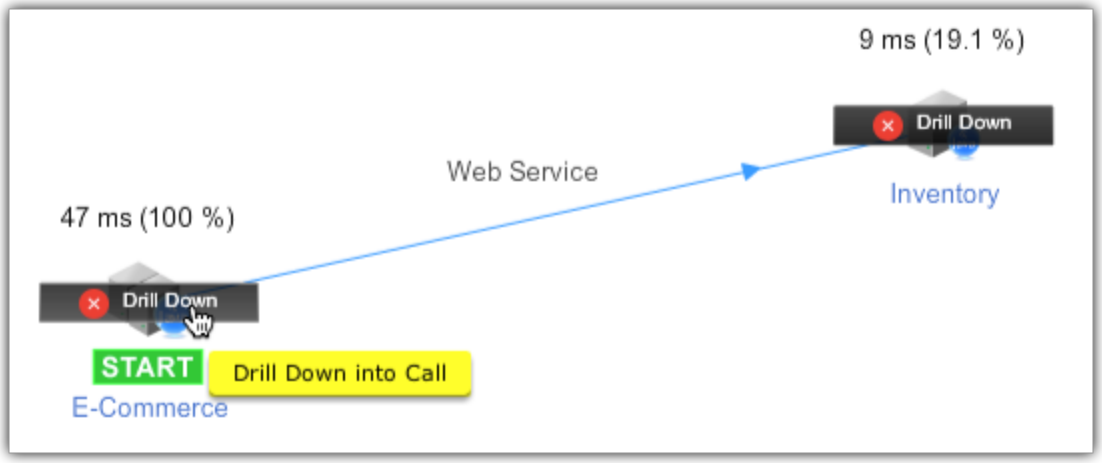
You drill down to the root cause of the problem in different ways depending on the type of event.

Drill down to an Error

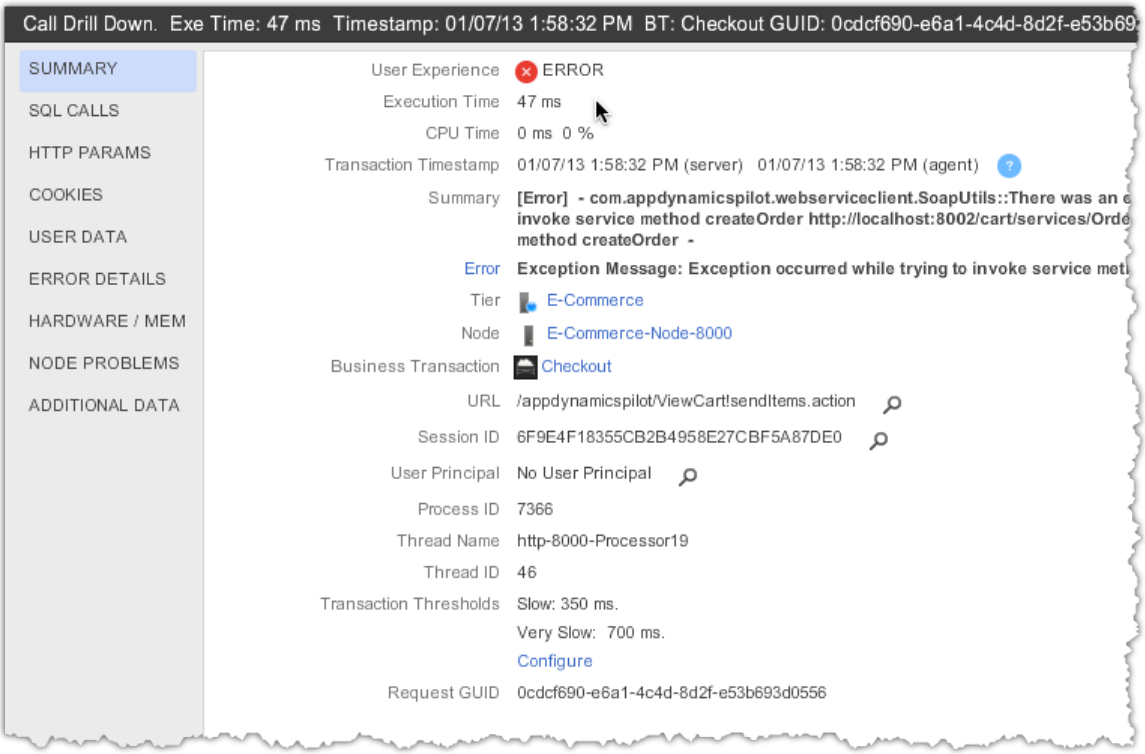
1. In the Events window click an Error.



2. In the Transaction Flow Map click the Drill Down icon. If there are multiple drill down icons, select the one with the transaction that takes the most time.



3. In the Call Drill Down window click the Summary tab.

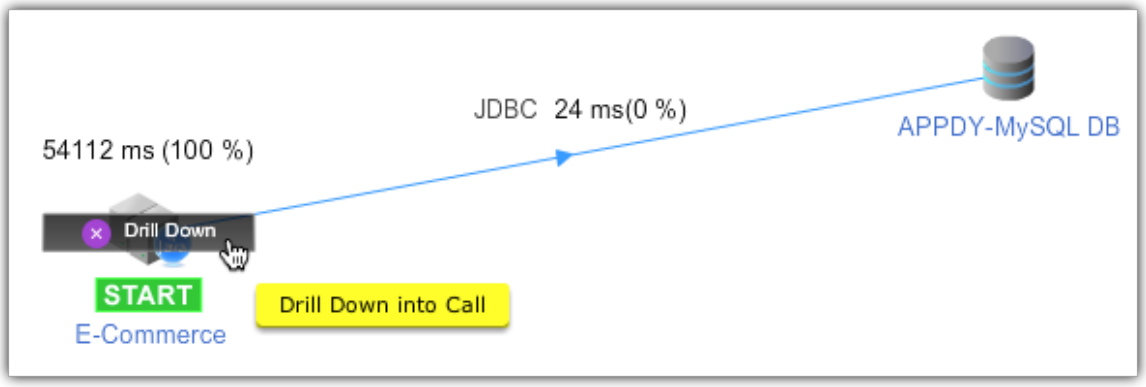


4. Use the Export to PDF button to save and email the information to your colleagues.

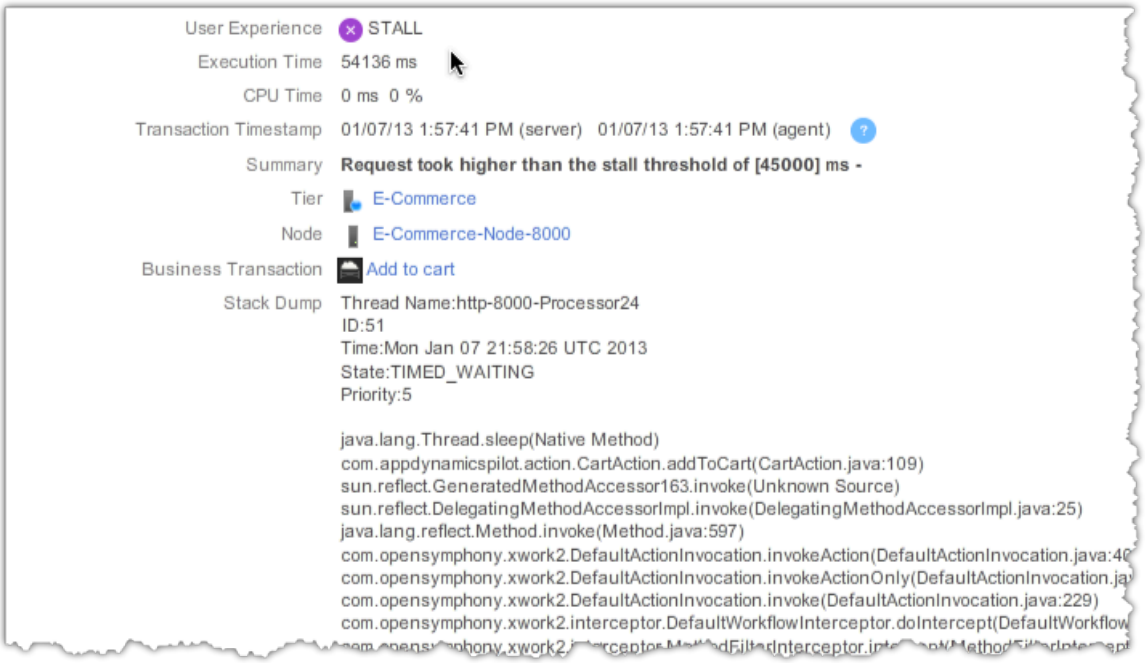
Drill down to a stalled Business Transaction

1. In the Events window click a Slow Requests - Stalled row.

2. In the Transaction Flow Map click the Drill Down icon.



3. In the Call Drill Down window click the Summary tab.

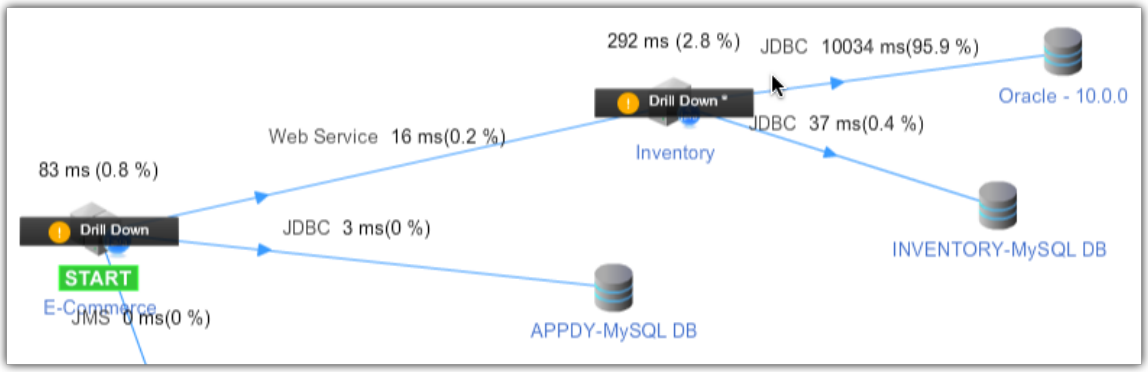


4. Use the Export to PDF button to save and email the information to your colleagues.

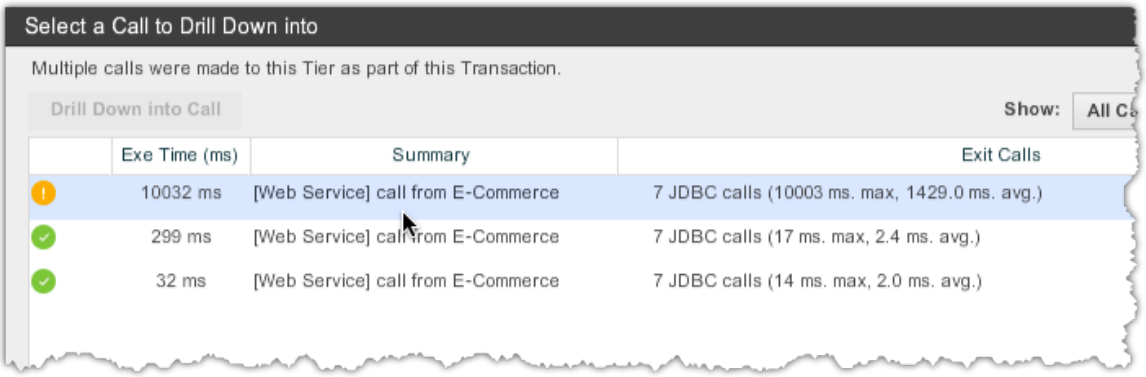
Drill down to a slow or very slow Business Transactions

1. In the Events window click a Slow Requests - Very Slow or Slow row.

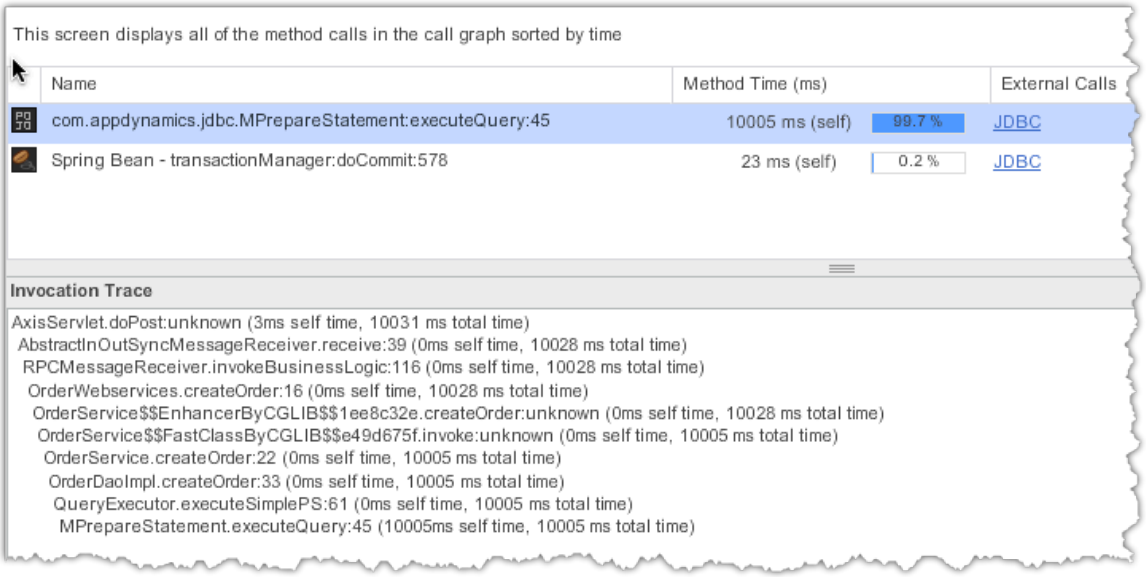
2. In the Transaction Flow Map click the Drill Down icon. If there are multiple drill down icons, select the one with the transaction that takes the most time.



3. In the Select a Call windows click the slowest call.



4. In the Call Drill Down window click the Hot Spots tab to see the slowest methods.



5. In this example, since the slow call is a database call you know you can click the SQL Calls tab to see the slowest SQL.

6. Use the Export to PDF button to save and email the information to your colleagues.

Drill down to an Application Server Exception

In the Events window click an Application Server Exception.

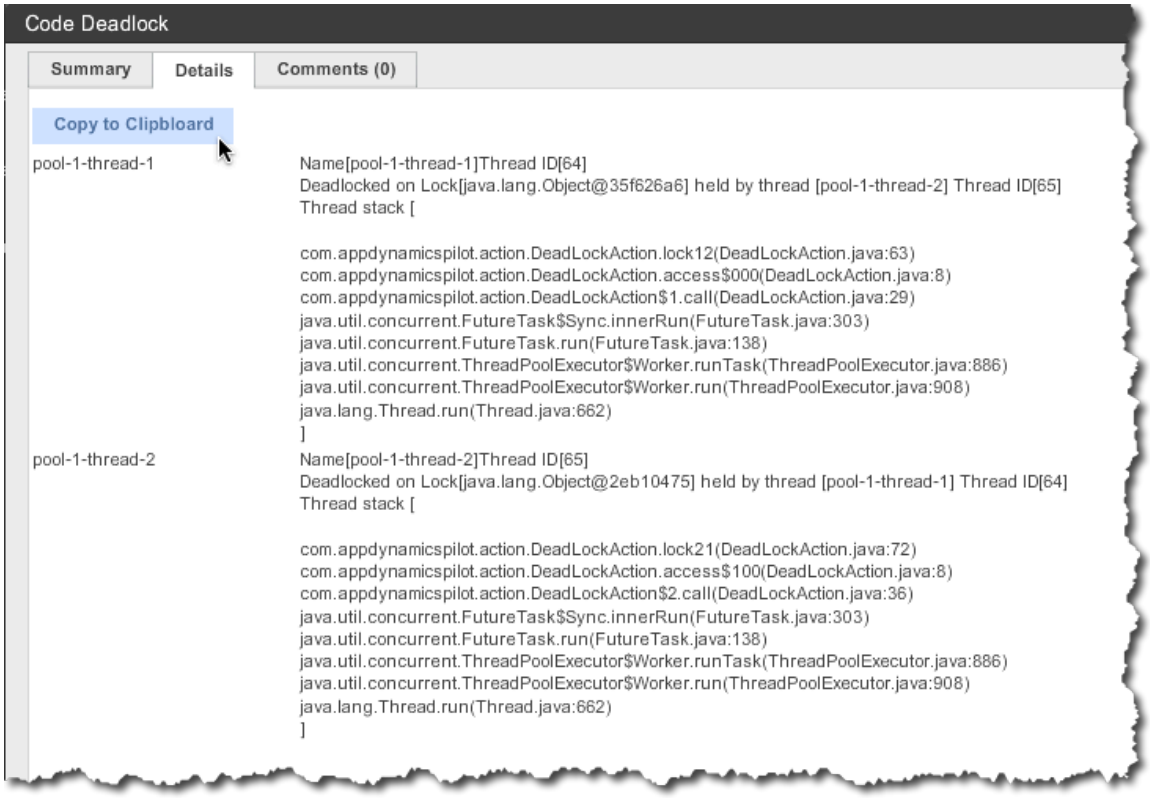
In the Application Server Exception window click the Details tab.

Use the Copy to Clipboard button to save and email the information to your colleagues.

Drill down to a Code Deadlock

1. In the Events window click a Code Deadlock.

2. In the Code Deadlock window click the Details tab.

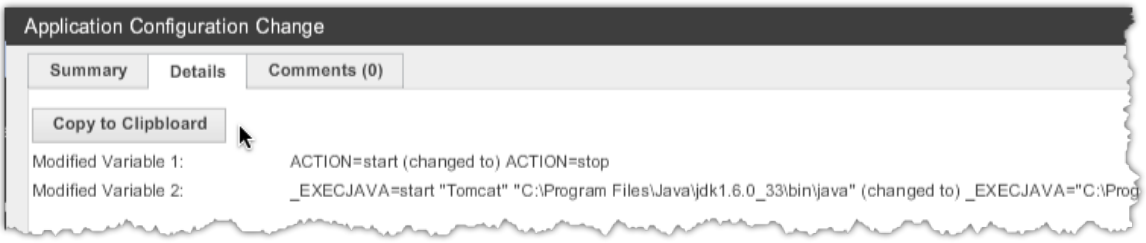


3. Use the Copy to Clipboard button to save and email the information to your colleagues.

Drill down to Application Change Events

By default AppDynamics reports events when applications are deployed, app servers restarted, and configuration parameters changed. Since these are not problems, they are indicated by a blue icon.

Click a change event to see a summary and details, for example:



Tutorial for Java - Business Transaction Health Drilldown

Business Transaction Drilldown WATCH THE VIDEO

Download MP4 version: BTHealthDrilldown.mp4

Download QuickTime version: BTHealthDrilldown.mov

Tutorial for Java - Exceptions

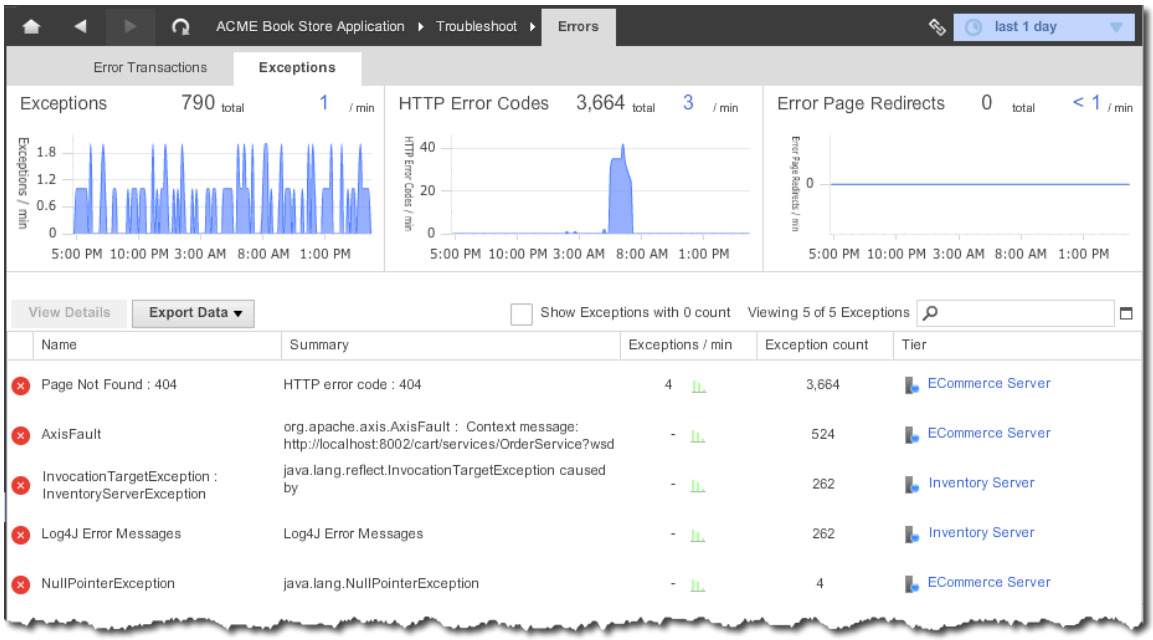
- The Exceptions
- Drill Down into the HTTP Error Code Exception
- Drill Down into the AxisFault Exception
- Drill Down into the Logger Exception
- See How Exceptions are Configured
- Learn More

The Exceptions

An exception is a code-logged message outside the context of a business transaction. Common exceptions include code exceptions or logged errors, HTTP error codes, and error page redirects.

Exceptions display in the Exceptions pane of many dashboards.

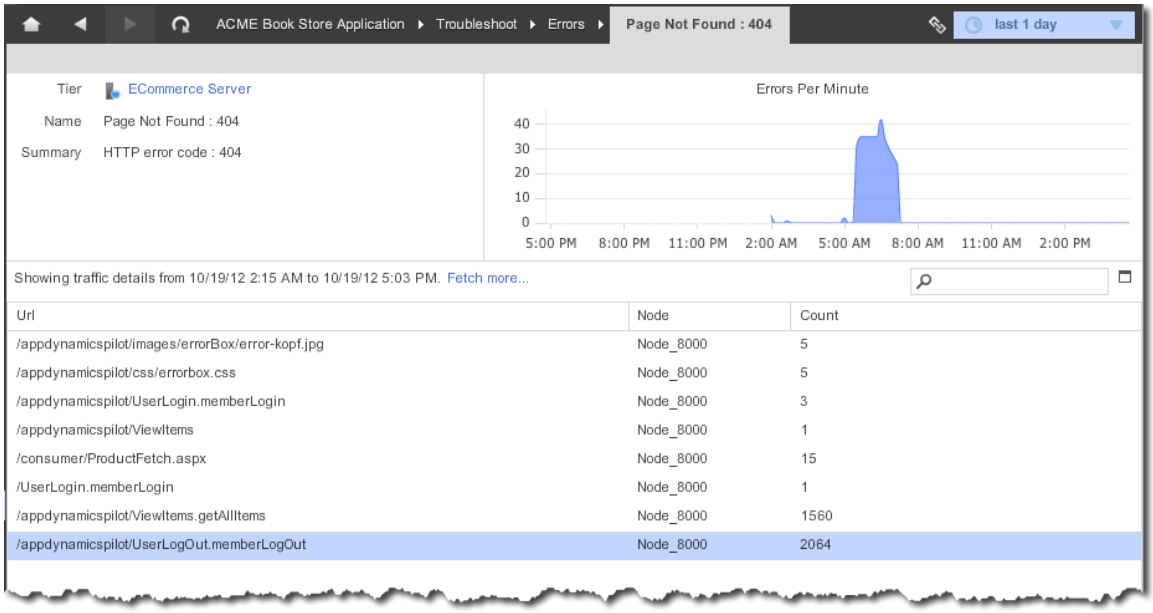
Click Exceptions to quickly see a list, ordered by frequency.



Drill Down into the HTTP Error Code Exception

Notice the spike in the HTTP Error Codes graph, and that the "Page Not Found: 404 error" is the most frequent.

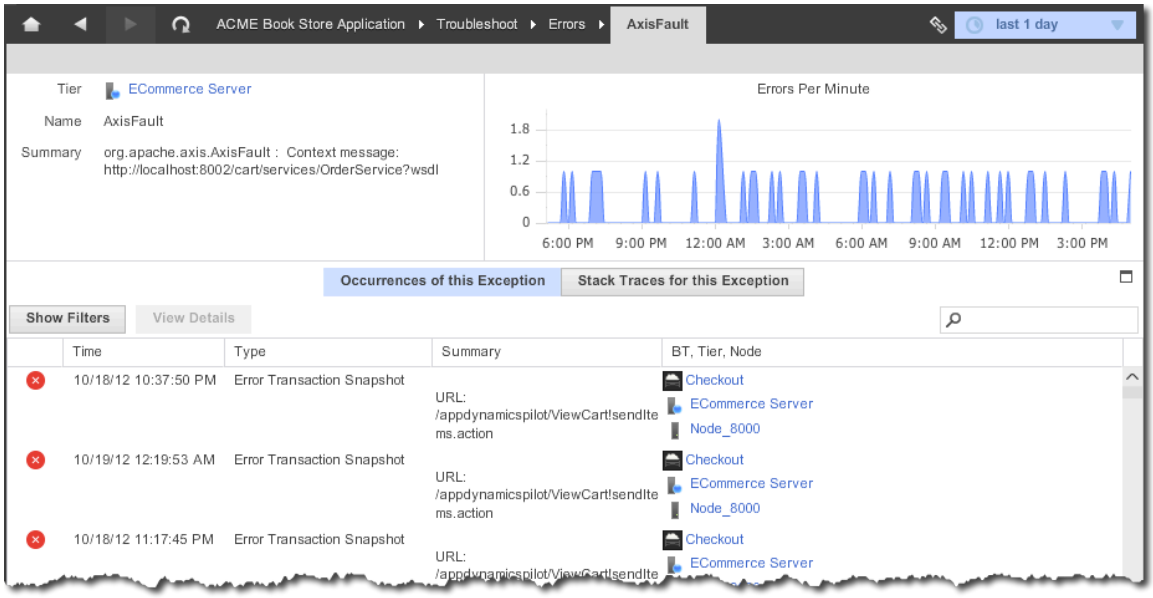
To find out more about the 404 error, click the row.



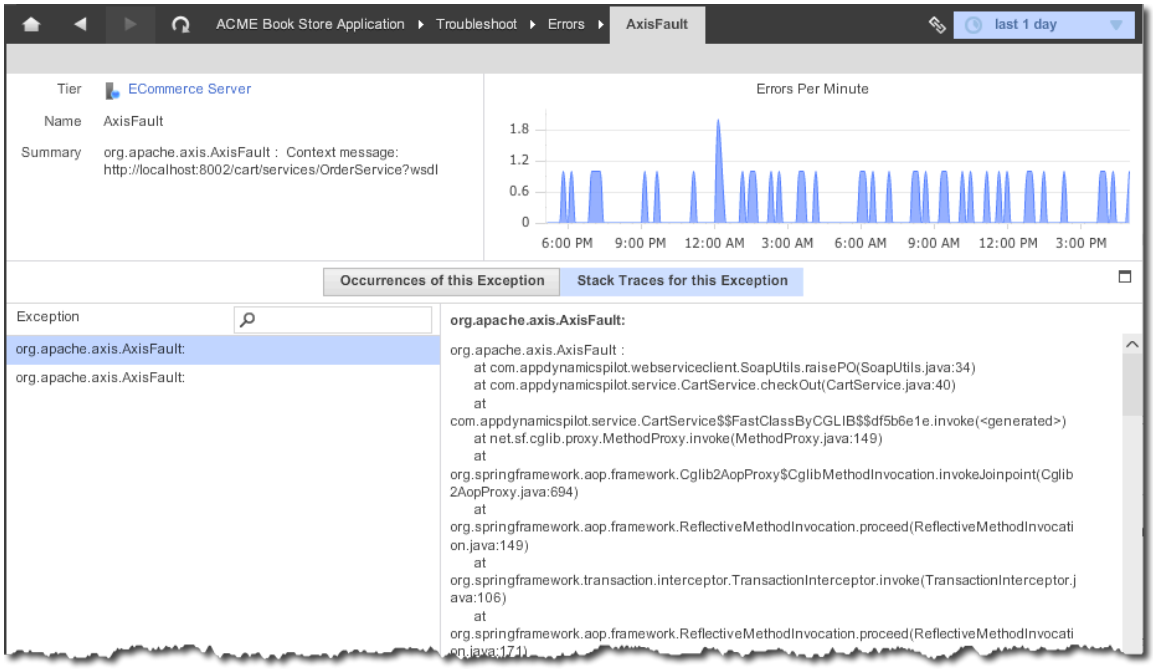
The list of URLs shows pages that have 404 errors. The memberLogOut and getAllItems URLs have the most 404 errors. You can provide this information to the web team to determine why those pages have so many 404 errors.

Drill Down into the AxisFault Exception

In the Exceptions tab, click the AxisFault row. A list of error snapshots shows the affected URL, tier, and node.



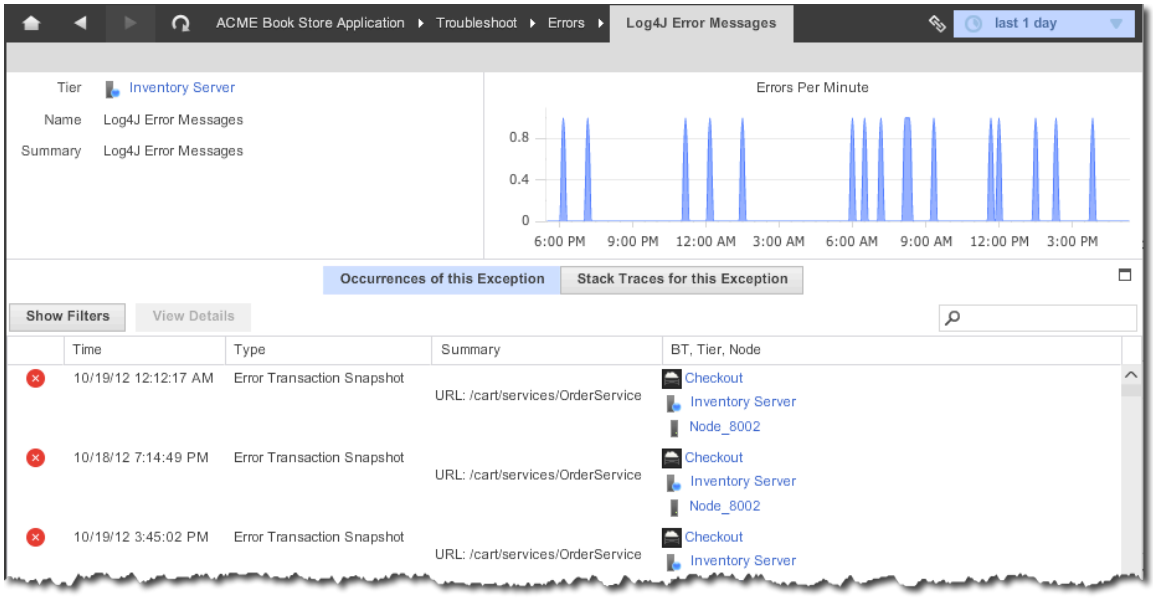
Click a row and then click the Stack Traces for This Exception tab to drill down further. Then click on one of the exceptions to see the stack trace.



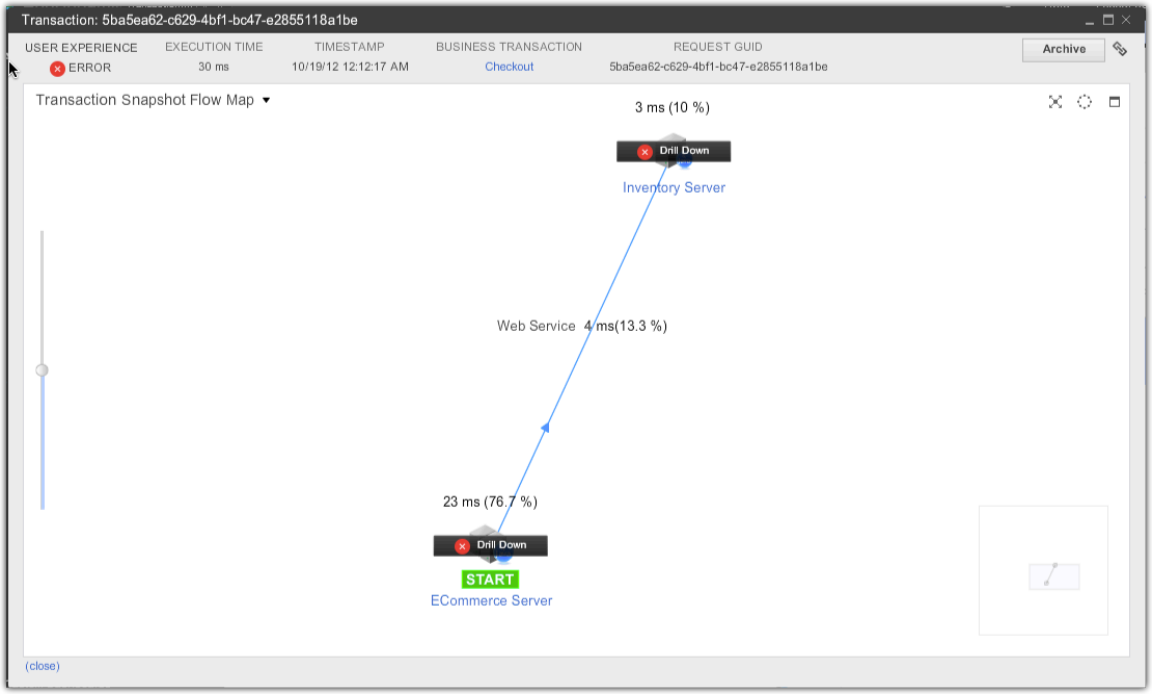
Share the stack trace with the development team to solve the problem.

Drill Down into the Logger Exception

In the Exceptions tab, click the Log4J Error Messages row. A list of error transaction snapshots shows the affected URL, business transaction, tier, and node. You can see a graph of the errors-per-minute data.

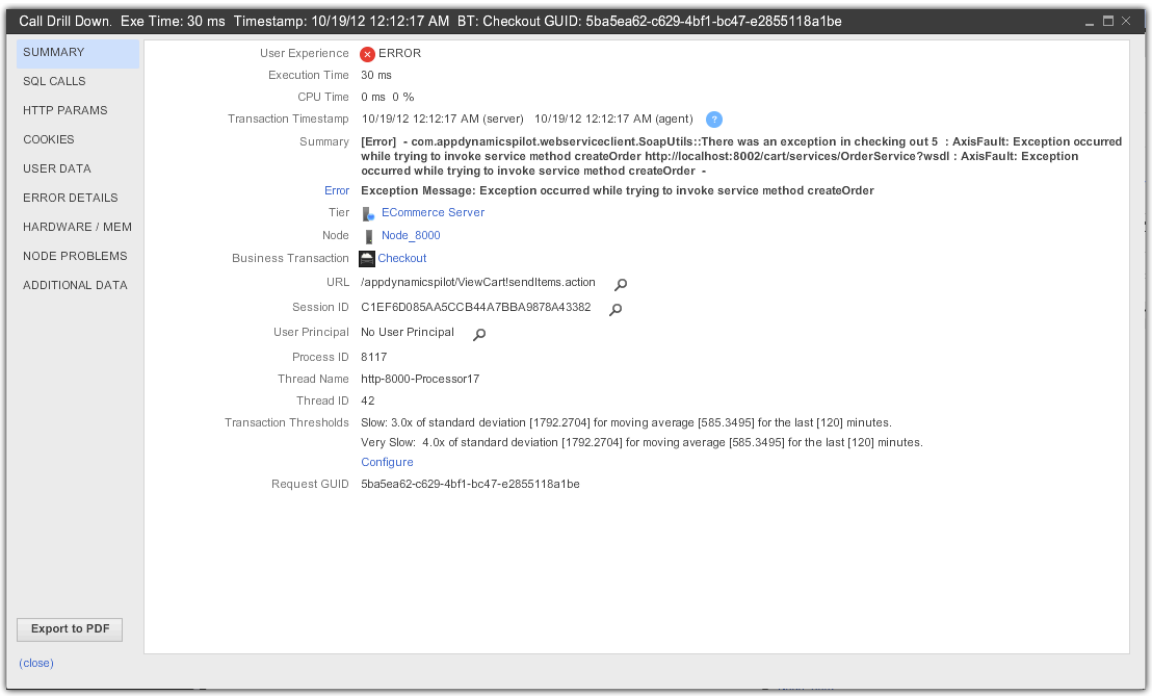


Click on a row to see the flow map for the error transaction snapshot.



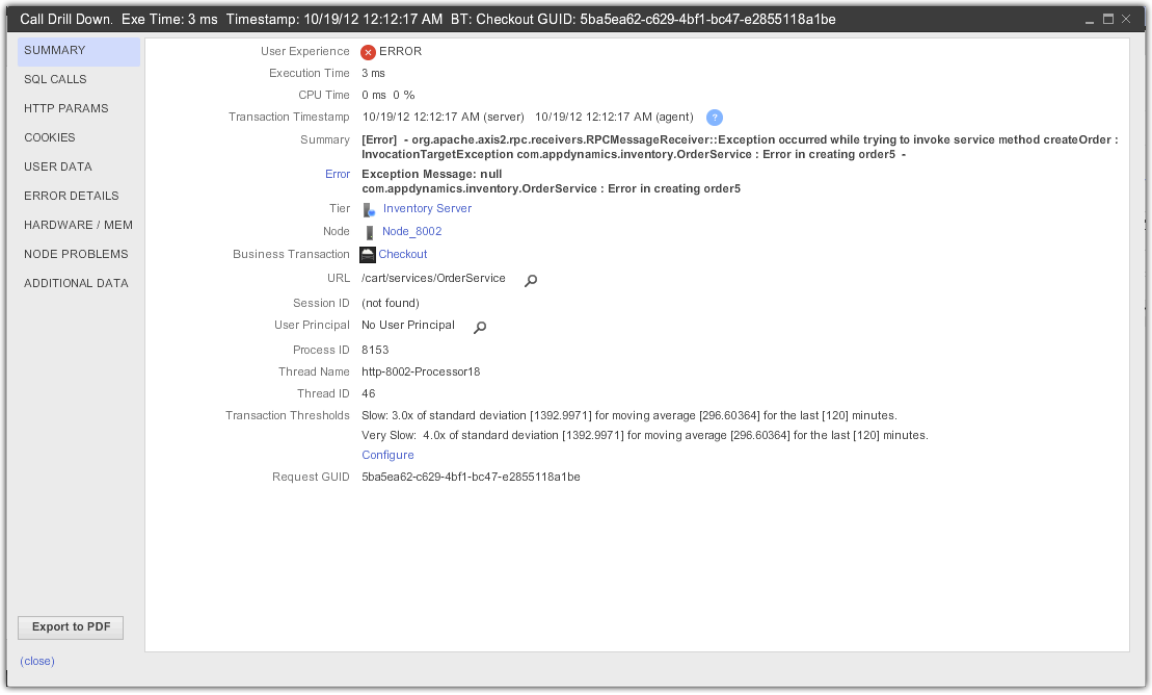
The icons for both tiers have a Drill Down button. Click the Drill Down button on Ecommerce tier; it also says "Start", indicating that the transaction started on this tier.

The Call Drill Down shows the summary of the error message.



You can use the Export to PDF button at the lower left to send this information to your colleagues.

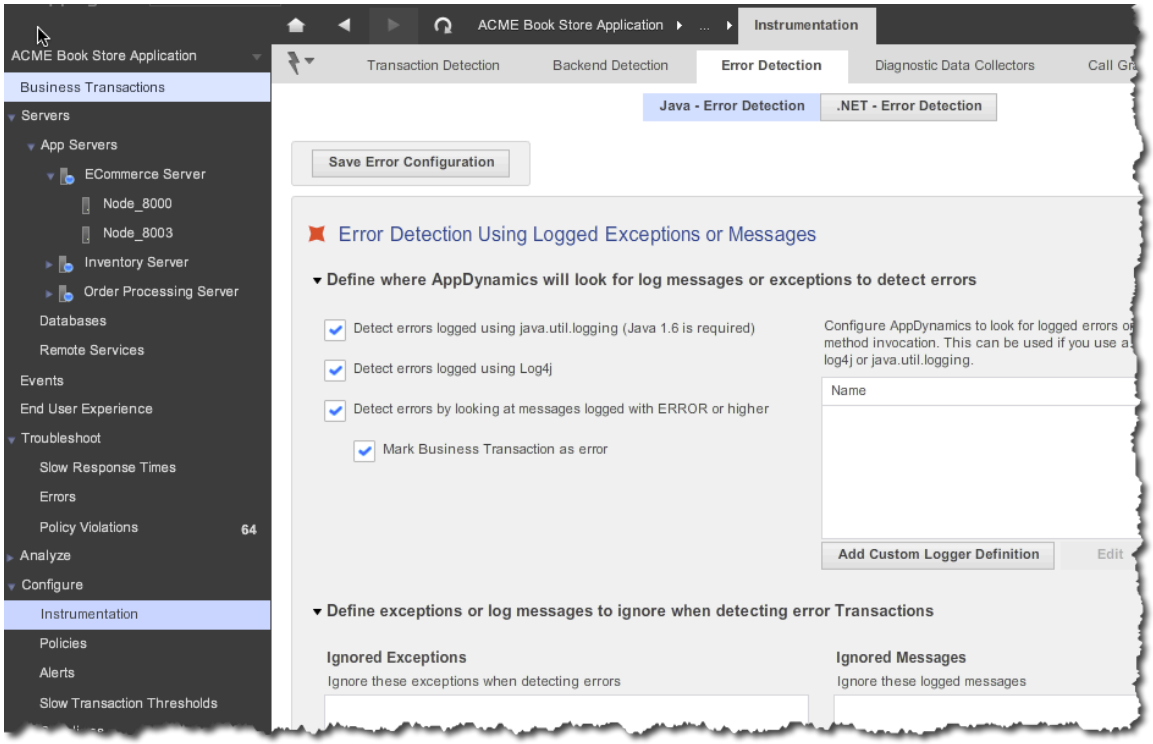
Go back to the flow map and click the Inventory tier **Drill Down** button. You see the Call Drill Down of the Inventory tier error message.



Compare the two error messages.

See How Exceptions are Configured

AppDynamics provides application-level default configurations for detecting exceptions. In the left navigation pane click **Configure -> Instrumentation -> Error Detection**.

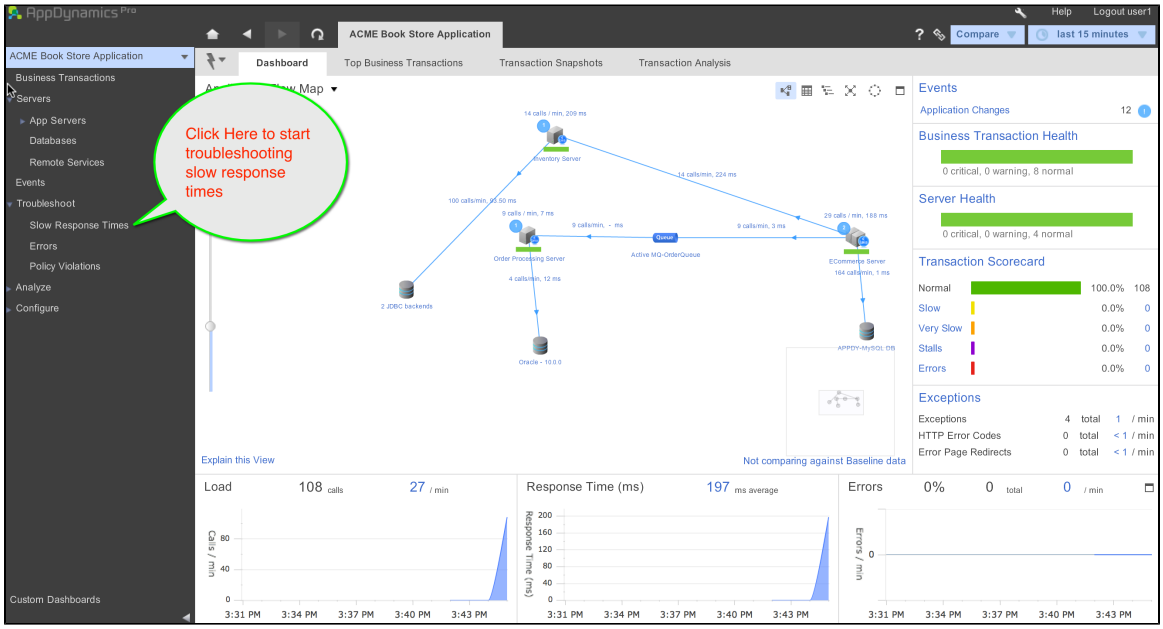


Learn More

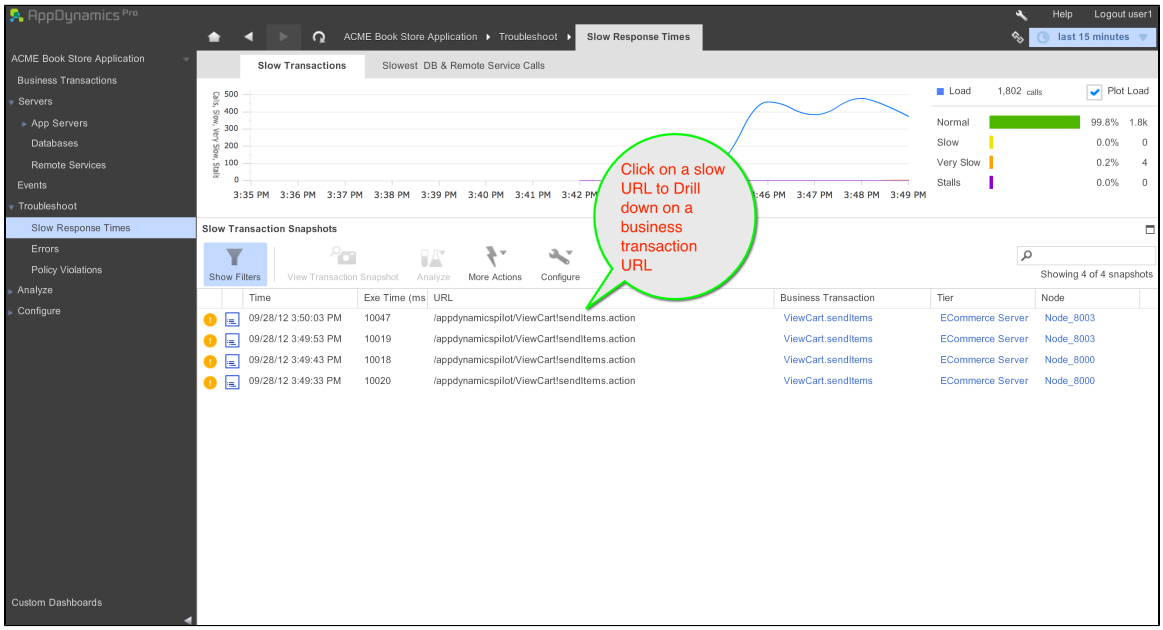
- Troubleshoot Errors
- Configure Error Detection

Tutorial for Java - Slow Transactions

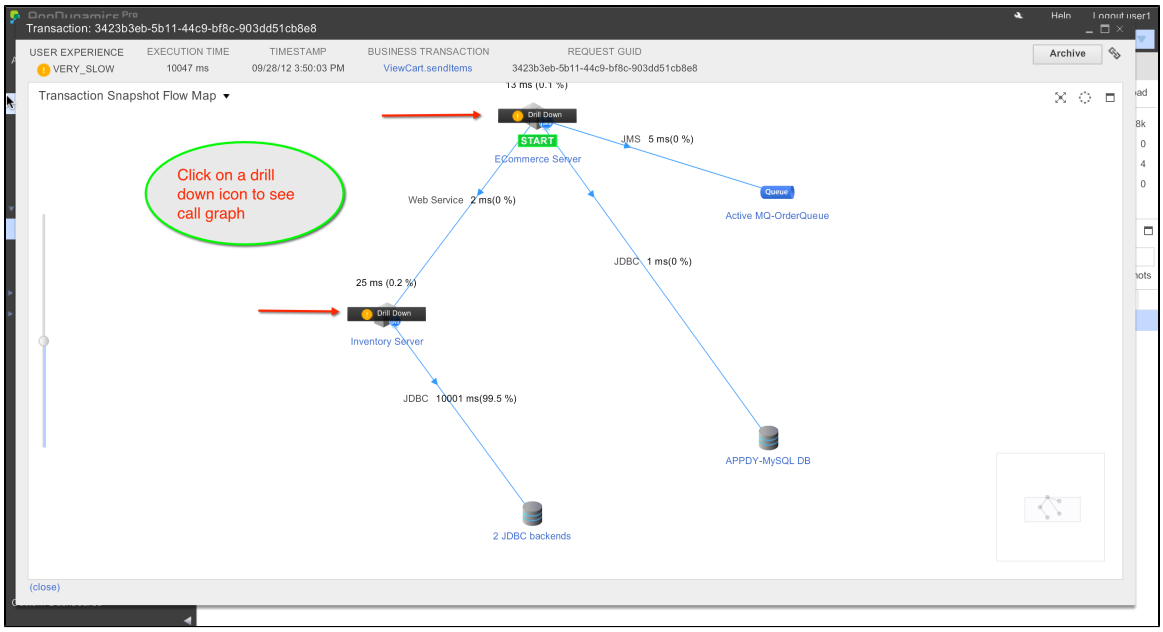
To begin troubleshooting navigate using the menu on the right:



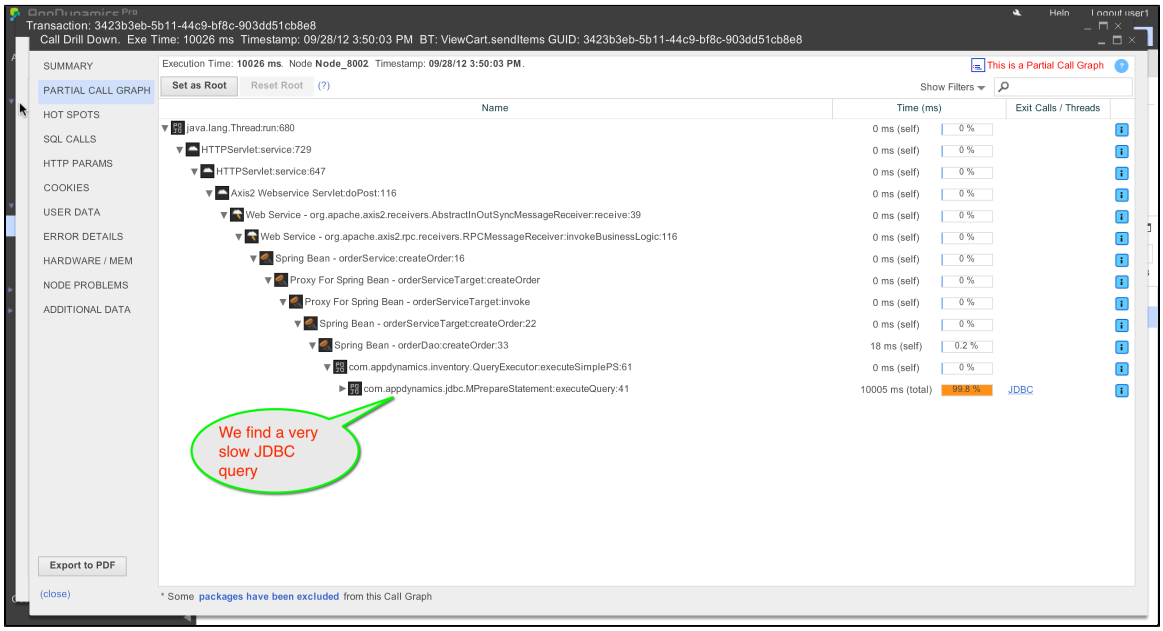
To Troubleshoot slow URLs use the Slow URL browser:



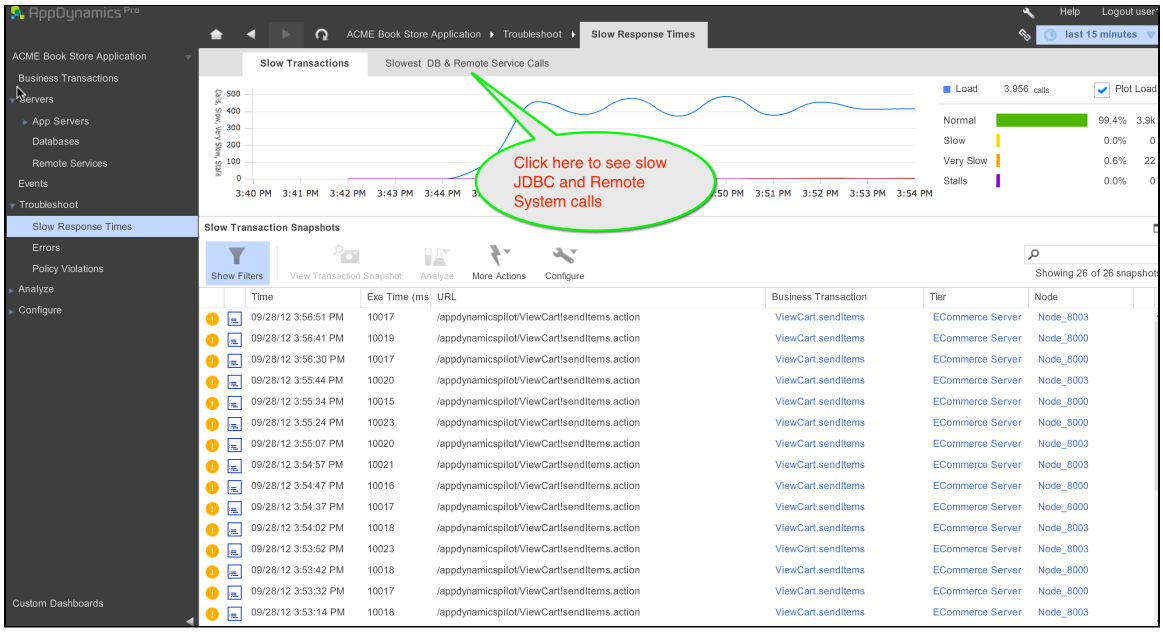
Once you select the URL you will see a visualization of the transaction. You can drill into a callgraph by clicking on the drill-down icon.



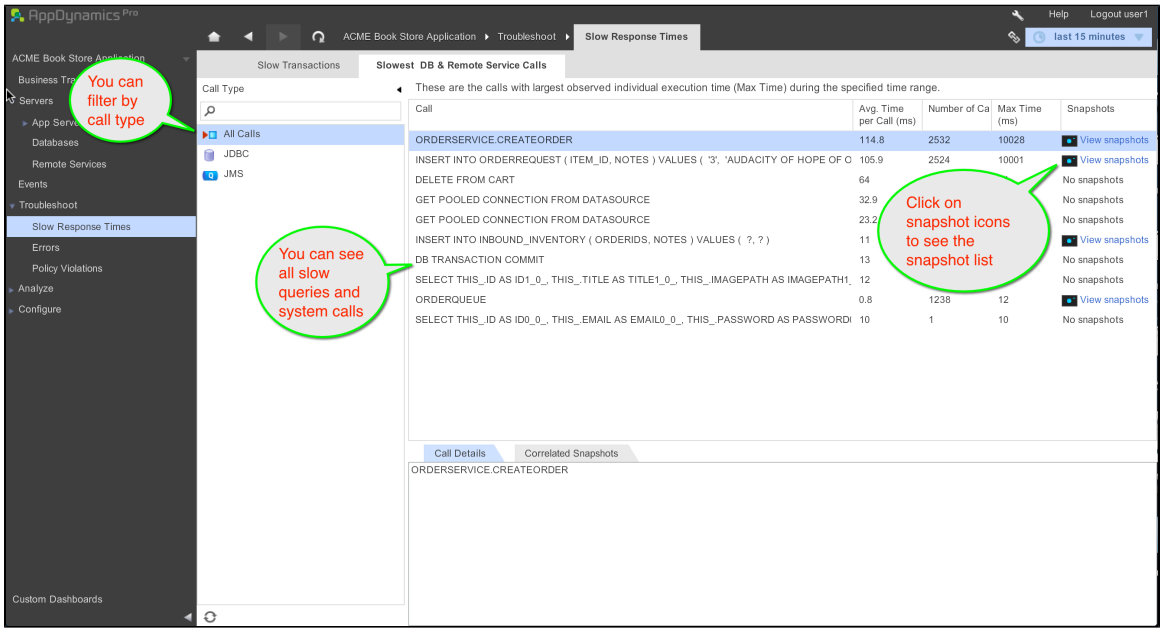
Once you are in the call graph you can look for methods that have a significant response time. For example, the executeQuery method is responsible for 99% of response time:



From the Troubleshoot menu you can navigate to the list of slow DB and Remote Calls.



You can drill into the transaction snapshots from the slow query and remote calls page to see the snapshot view:



Install the App Agent for Java

- Overview of the App Agent for Java Installation Process
- Planning for Agent Installation
  - Important Files
  - To Install the Java App Server Agent
    - 1. Download and unzip the App Agent for Java
    - 2. Add the agent properties as a 'javaagent' argument to your JVM
    - 3. Configure how the agent connects to the Controller
    - 4. (Only for Multi-tenant mode or SaaS Installations): Configure Agent account information
    - 5. Configure how the agent identifies the AppDynamics business application, tier, and node.
      - Automatic Naming for Application, Tier, and Node

- Additional Installation Scenarios
- 6. Verify agent configuration
- 7. Verify successful installation and reporting
  - a. Verify agent installation
  - b. Verify that the agent is reporting to the Controller
- Example Configuration: App Agent for Java Deployment on a Single JVM
- Learn More

The AppDynamics App Agent for Java identifies and tracks business transactions, captures statistics and diagnostic data, and analyzes and reports data to the Controller. The App Agent for Java uses dynamic bytecode injection to instrument a JVM and it runs as a part of the JVM process.

Overview of the App Agent for Java Installation Process

Installing the App Agent for Java involves adding it as a javaagent (Java Programming Language Agent) on your JVM and setting up connection and identifying parameters for it to report data to the Controller.

Install the App Agent for Java as the same user or administrator of the JVM. Otherwise the agent may not have the correct write permissions for the system. The agent directories must have write permission so that AppDynamics can update the logs and other agent files.

Planning for Agent Installation

Before installing the App Agent for Java, be prepared with the following information.

	Planning Item	Description
✓	Where is the startup script for the JVM? If using a Java service wrapper, you need to know where is the wrapper configuration.	This is where you can add startup arguments in the script file and system properties, if needed.
✓	What host and port is the Controller running on?	For SaaS customers, AppDynamics provides this information to you. For on-premise Controllers, this information is configured during Controller installation. See (Install the Controller on Linux or Install the Controller on Windows).
✓	To what AppDynamics business application does this JVM belong?	Usually, all JVMs in your distributed application infrastructure belong to the same AppDynamics business application. You assign a name to the business application. For details see Logical Model.
✓	To what AppDynamics tier does this JVM belong?	You assign a name to the tier. For details see Logical Model.

Important Files

In addition to the JVM startup script file, two other files are important during installation:

- The -javaagent argument uses the fully-qualified path of the javaagent.jar file. No separate classpath arguments need to be added.
- The <agent_home>/conf/controller-info.xml file is where you add the configuration mentioned in the planning list.

To Install the Java App Server Agent

1. Download and unzip the App Agent for Java

- Download the App Agent for Java ZIP file from AppDynamics Download Center.
- Extract the ZIP file to the destination directory as the same user or administrator of the JVM.

2. Add the agent properties as a 'javaagent' argument to your JVM

This step adds the agent to the startup script of your application server. Use the server-specific instructions below to add this argument for different Application Server JVMs:

3. Configure how the agent connects to the Controller

- Configure properties for the Controller host name and its port number.
- You can configure these two properties using either the controller-info.xml file or the JVM startup script:

Configure using controller-info.xml	Configure using System Properties	Required	Default
<controller-host>	-Dappdynamics.controller.hostName	Yes	None
<controller-port>	-Dappdynamics.controller.port	Yes	**For On-premise Controller installations:** By default, port 8090 is used for HTTP and 8181 is used for HTTPS communication. **For SaaS Controller service:** By default, port 80 is used for HTTP and 443 is used for HTTPS communication.

Optional settings for Agent-Controller communication

- To configure the Java Agent to use SSL, see App Agent for Java Configuration Properties
- To configure the Java Agent to use proxy settings see App Agent for Java Configuration Properties

4. (Only for Multi-tenant mode or SaaS Installations): Configure Agent account information

- This step is required only when the AppDynamics Controller is configured in Controller Tenant Mode or when you Use a SaaS Controller.
  Skip this step if you are using single-tenant mode, which is the default in an on-premise installation.
- Specify the properties for Account Name and Account Key. This information is provided in the Welcome email from the AppDynamics Support Team.

Configure using controller-info.xml	Configure using System Properties	Required	Default
<account-name>	-Dappdynamics.agent.accountName	Required only if your Controller is configured for multi-tenant mode or your controller is hosted.	None.
<account-access-key>	-Dappdynamics.agent.accountAccessKey	Required only if your Controller is configured for multi-tenant mode or your controller is hosted.	None.

5. Configure how the agent identifies the AppDynamics business application, tier, and node.

To better understand agents and how they relate to business applications, tiers, and nodes see Logical Model and Name Business Applications, Tiers, and Nodes.

You can configure these properties using either the controller-info.xml file or JVM startup script options. Use these guidelines when configuring agents:

- Configure items that are common for all the nodes in the controller-info.xml file.
- Configure information that is unique to a node in the startup script.

Configure using controller-info.xml	Configure using System Properties	Required	Default
<application-name>	-Dappdynamics.agent.applicationName	Yes, unless you use automatic naming	None
<tier-name>	-Dappdynamics.agent.tierName	Yes, unless you use automatic naming	None
<node-name>	-Dappdynamics.agent.nodeName	Yes, unless you use automatic naming	None

Automatic Naming for Application, Tier, and Node

Starting with release 3.7.0, the App Agent for Java javaagent command has a new uniqueID argument that AppDynamics uses to automatically name the node and its tier. For example, using this command argument AppDynamics will name the node "my-app-jvm1" and the tier "my-app-jvm1":

```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

When uniqueID is used, AND when an application name is not provided either through the system property or in the controller-info.xml, AppDynamics creates a new business application called "MyApp".

The new naming mechanism is used by the new Self-Service process. See Install Agents for 5 or fewer JVMs or CLRs (Self-Service Installations).

Additional Installation Scenarios

Refer to the links below for typical installation scenarios, especially for cases where there are multiple JVMs on the same machine:

- Configure App Agent for Java on Multiple JVMs on the Same Machine that Serves the Same Tier
- Configure App Agent for Java on Multiple JVMs on the Same Machine that Serve Different Tiers
- Configure App Agent for Java to Use Existing System Properties
- App Agent for Java on z-OS or Mainframe Environments Configuration
- Configure App Agent for Java for Batch Processes

6. Verify agent configuration

- Ensure that you have added -javaagent argument in your JVM startup script. This is not a -D system property but a different standard argument for all JVMs v1.5 and higher.
- Ensure that you have added all mandatory items either in the Agent controller-info.xml file or in the JVM startup script file.
- The user running the JVM process/application server process is the user accessing the Java Agent installation.

7. Verify successful installation and reporting

a. Verify agent installation

After a successful install, your agent logs, located at <agent_home>/logs, should contain following message:

```
Started AppDynamics Java Agent Successfully
```

If the agent log file is not present, the App Agent for Java may not be accessing the javaagent command properties. The application server log file where STDOUT is logged will have the fallback log messages, for further troubleshooting.

b. Verify that the agent is reporting to the Controller

Use the AppDynamics UI, to verify that the Java Agent is able to connect to the Controller:

- Point your browser to: http://<controller-host>:<controller-port>/controller
- Provide the admin credentials to log into the AppDynamics UI.
- Select the application. In the left navigation pane, click **Servers -> App Servers -> <tier> -> <node>**. Click the Agents tab and App Server Agent subtab. An agent successfully reporting to the Controller will be listed and the Reporting property shows an "up" arrow symbol. For more details see Verify App Agent - Controller Communication.
- When deploying multiple agents for the same tier, see if you get the exact number of nodes reporting in the same tier.

Example Configuration: App Agent for Java Deployment on a Single JVM

The following example shows a sample deployment of the App Agent for Java for the ACME Bookstore.

- Add the javaagent argument to the start-up script of the JVM:

```
java -javaagent:/home/appdynamics/AppServerAgent/javaagent.jar
```

- Define the five mandatory items for agent configuration in the Agent controller-info.xml file:

```
<controller-info>
    <controller-host>192.168.1.20</controller-host>
    <controller-port>8090</controller-port>
    <application-name>ACMEOnline</application-name>
    <tier-name>InventoryTier</tier-name>
    <node-name>Inventory1</node-name>

</controller-info>
```

Learn More

- App Agent for Java Configuration Properties
- Uninstall the App Agent for Java
- Logical Model

Multi-Agent Deployment for Java

- Deployment Procedure
  - To Deploy Java App Agents
  - To Deploy Java Machine Agents
- Sample Deployment Solutions
- Learn More

This topic describes the high-level procedures for deploying multiple AppDynamics app agents and machine agents on Java platforms.

Deployment Procedure

To Deploy Java App Agents

1. Download the latest agent ZIP file from http://download.appdynamics.com/.

2. Update deployment artifacts to use the downloaded agent.

3. Unzip the downloaded app agent file on the destination machine in the desired app agent directories.

4. Modify the app-agent-config.xml file with any custom settings for the node, tier or app.

5. Do one of the following for each application server:

- Set the application name, tier name, node name, controller host and controller port properties in the <Agent_Installation_Directory>/conf/controller-info.xml file.
  OR
- Set these properties as system startup properties in the application server startup script using the -D option.
  See App Agent for Java Configuration Properties for more information about these properties.

6. Restart the application servers to make the changes take effect.

See Install the App Agent for Java for detailed instructions on installing the app agent.

To Deploy Java Machine Agents

1. Download the latest AppDynamics Machine Agent ZIP file from http://download.appdynamics.com/.

2. Unzip the downloaded machine agent file on the destination machine in the desired machine agent directories.

3. Modify the <Machine_Agent_Installation_Directory>/conf/controller-info.xml files to set the application name, tier name, node name, controller host and controller port properties. Note that there are no -D settings allowed for machine agents, unlike app agents.

4. Configure the startup script for the machine to start the machine agent every time the machine reboots. For example, you could add the machine startup command to .bashrc.

To handle large values for metrics, run the machine agent using a 64-bit JDK.

Sample Deployment Solutions

You can download some sample solutions that our customers have created to perform multi-agent AppDynamics rollouts.

Use these samples for ideas on how to automate AppDynamics agent deployment for your environment. All of these samples deploy the agents independently of the application deployment.

- ChefExample1 and ChefExample2 use Opscode Chef recipes to automate deployment on Java platforms. See http://www.opscode.com/chef/ for information about Chef.
- JavaExample1 uses a script, configuration file and package repository.

Click below to download the samples.

- ChefExample1.tar
- ChefExample2.tar
- JavaExample1.tar

Learn More

- https://github.com/edmunds/cookbook-appdynamics

Java Server-Specific Installation Settings

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.


```
-javaagent:<agent_home>/javaagent.jar=uniqueID=<my-app-jvm1>
```

See Name Business Applications, Tiers, and Nodes.

Apache Cassandra Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to the cassandra (Linux) or cassandra.bat (Windows) file.

To add the javaagent command in a Windows environment

1. Open the apache-cassandra-x.x.x\bin\cassandra.bat file.

2. Add the AppDynamics javaagent to the JAVA_OPTS variable. Make sure to include the drive in the full path to the App Server agent directory.

```
-javaagent:<agent_home>\javaagent.jar
```

For example:

```
set JAVA_OPTS=-ea
    -javaagent:C:\appdynamics\agent\javaagent.jar
    -javaagent:"%CASSANDRA_HOME%\lib\jamm-0.2.5.jar
    . . .
    . . .
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

3. Restart the Cassandra server. The Cassandra server must be restarted for the changes to take effect.

To add the javaagent command in a Linux environment

1. Open the apache-cassandra-x.x.x/bin/cassandra.in.sh file.

2. Add the javaagent argument at the top of the file:

```
JVM_OPTS=-javaagent:<agent_home>/javaagent.jar
```

For example:


```
JVM_OPTS=-javaagent:/home/software/appdynamics/agent/javaagent.jar
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

3. Restart the Cassandra server for the changes to take effect.

Apache Tomcat Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your Tomcat catalina.sh or catalina.bat file.

If you are using Tomcat as a Windows service, see Tomcat as a Windows Service Configuration.

To add the javaagent command in a Windows environment

1. Open the **catalina.bat** file, located at <apache_version_tomcat_install_dir>\bin.

2. Add following javaagent argument to the beginning of your application server start script.

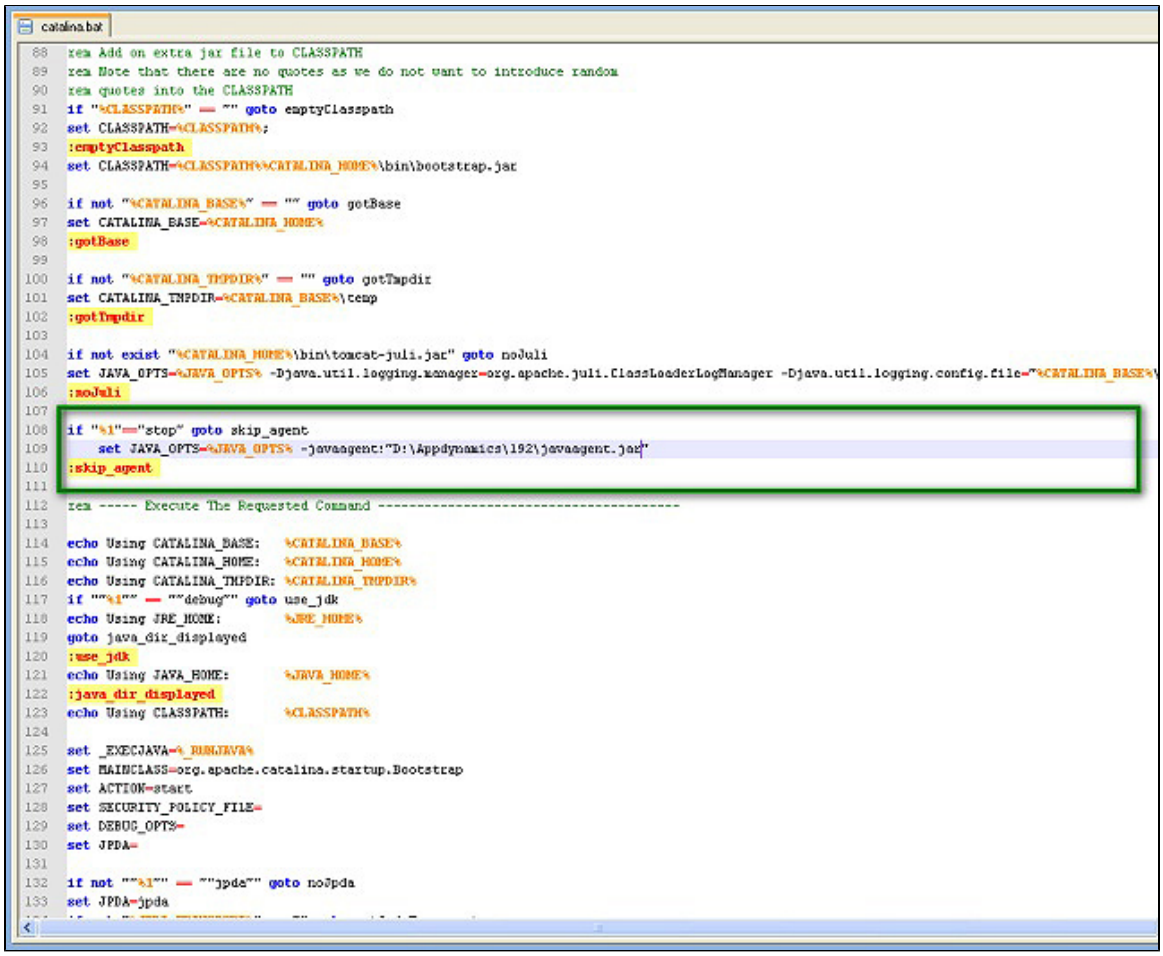
```
if "%1"=="stop" goto skip_agent
set JAVA_OPTS=%JAVA_OPTS% -javaagent:"Drive:<agent_home>\javaagent.jar"
:skip_agent
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

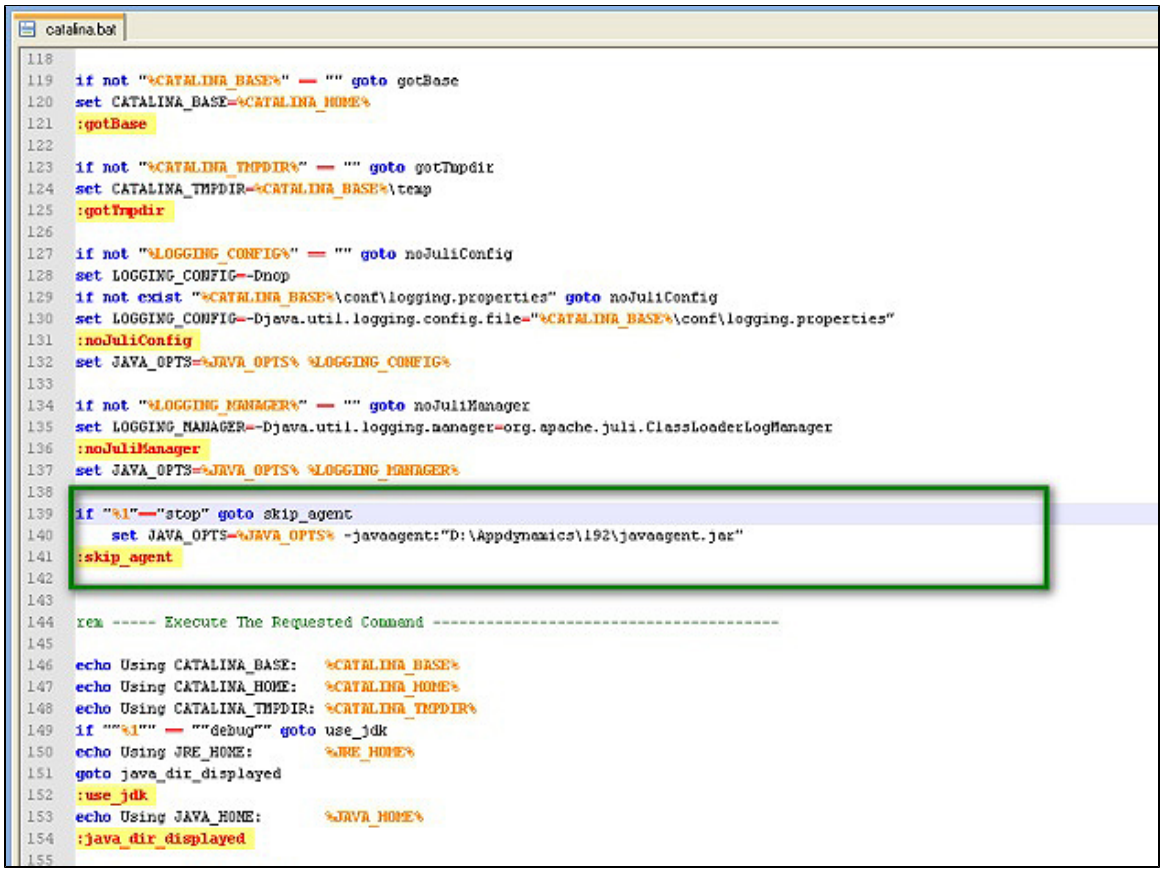
```
-javaagent:"Drive:<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path to the App Server Agent installation directory, including the drive. For details see the screen captures.

2a. Sample Tomcat 5.x catalina.bat file



2b. Sample Tomcat 6.x catalina.bat file



3. Restart the application server. The application server must be restarted for the changes to take effect.

To add the javaagent command in a Linux environment

1. Open the catalina.sh file located at <apache_version_tomcat_install_dir>/bin).

2. Add the following commands at the beginning of your application server start script.

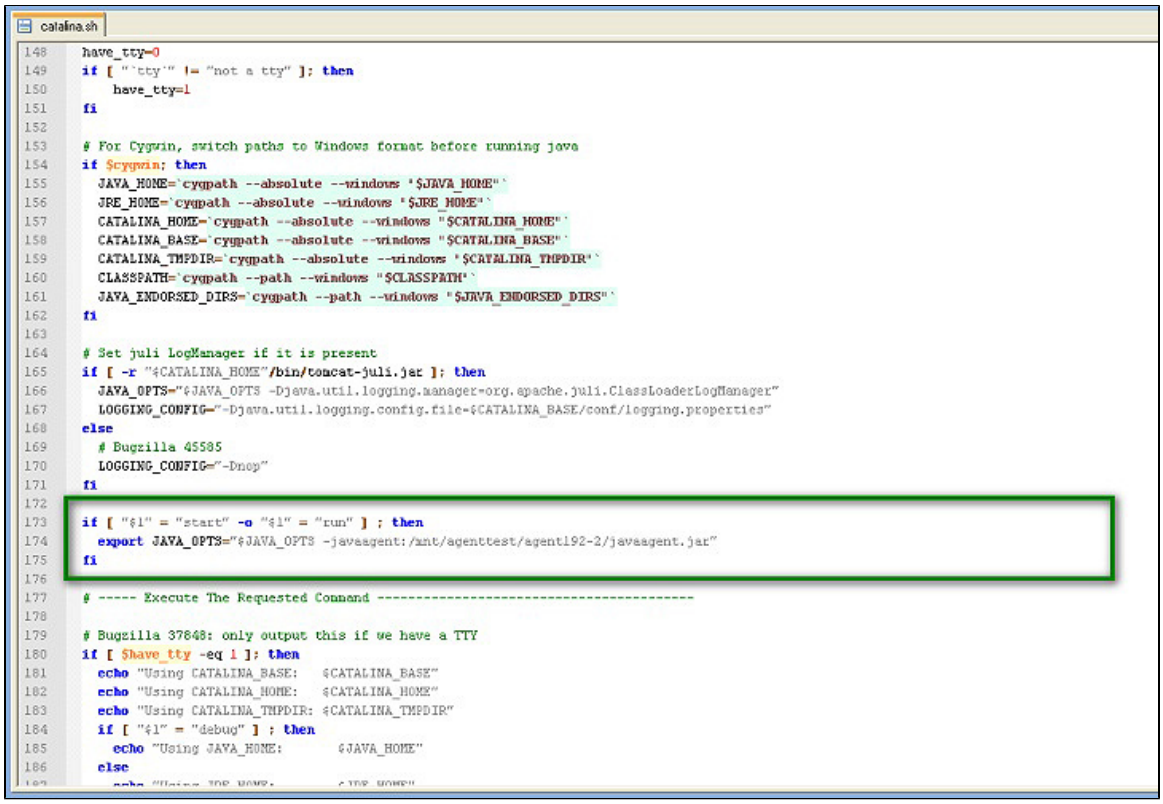
```
if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:agent_install_dir/javaagent.jar"
fi
```

The javaagent argument references the full path to the App Server Agent installation directory.

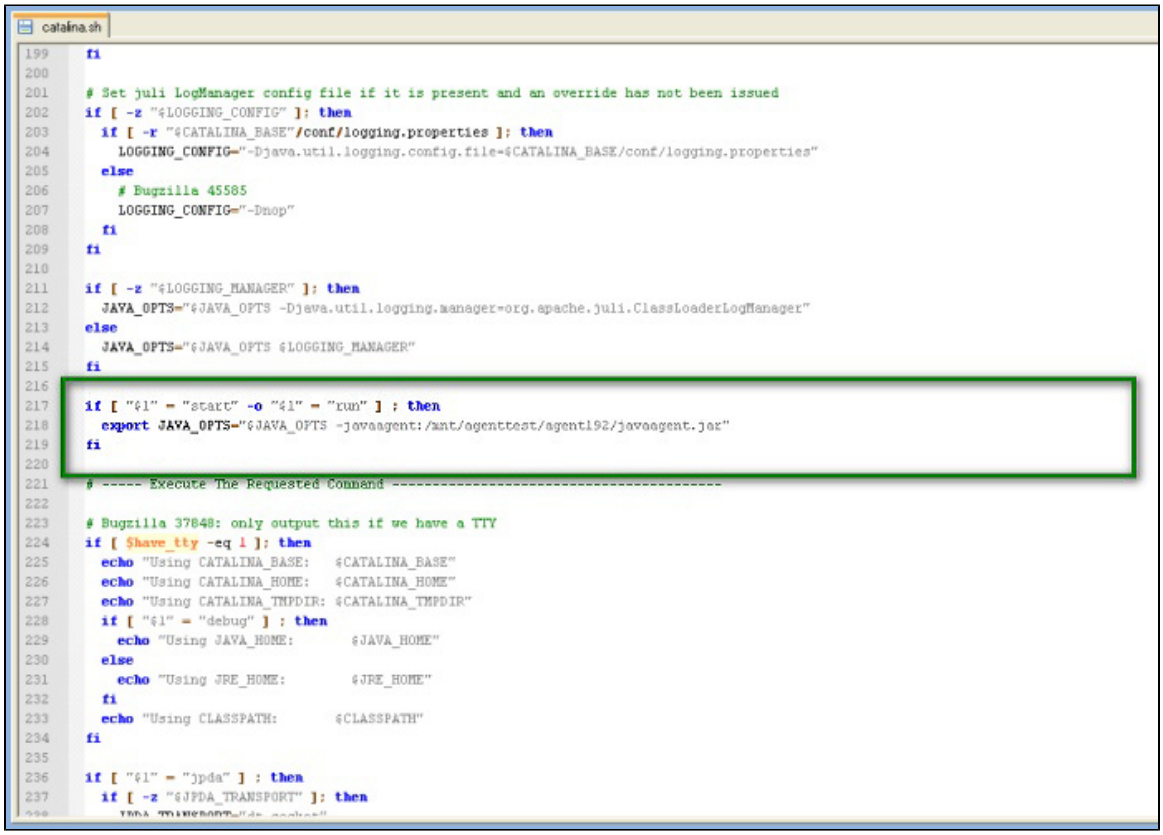
If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

For details see the screen captures.

2a. Sample Tomcat 5.x catalina.sh file



2b. Sample Tomcat 6.x catalina.sh file



3. Restart the application server. The application server must be restarted for the changes to take effect.

Tomcat as a Windows Service Configuration

- To install the javaagent as a Tomcat Windows service

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your Tomcat properties.

If you are not running Tomcat as a Windows service, see Apache Tomcat Startup Settings.

To install the javaagent as a Tomcat Windows service

These instructions apply to Apache Tomcat 6.x or later versions.

1. Ensure that you are using administrator privileges.

2. Click **Programs -> Apache Tomcat**.

3. Run **Configure Tomcat**.

4. Click the **Java** tab.

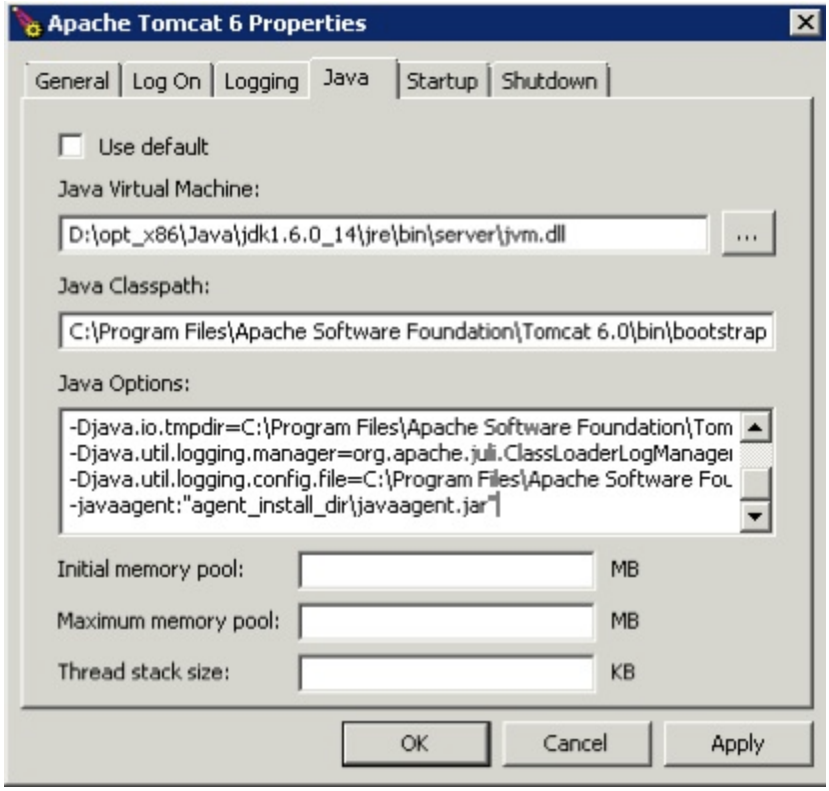
5. In the **Java Options** add:

```
-javaagent:"<agent_home\javaagent.jar"
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

For details see the following screenshot.



6. Restart the Tomcat service. The application server must be restarted for the changes to take effect.

Glassfish Startup Settings

- To add the javaagent command in a GlassFish environment
- To verify the Agent configuration
- About AppServer Management Extensions

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option.

To add the javaagent command in a GlassFish environment

1. If you are using **GlassFish v3.x**, first configure the OSGi containers. For details see OSGi Infrastructure Configuration.

2. Log into the **GlassFish domain** where you want to install the App Server Agent.

3. In the left navigation tree **Common Tasks** section, click **Application Server**. The Application Server Settings dialog opens.

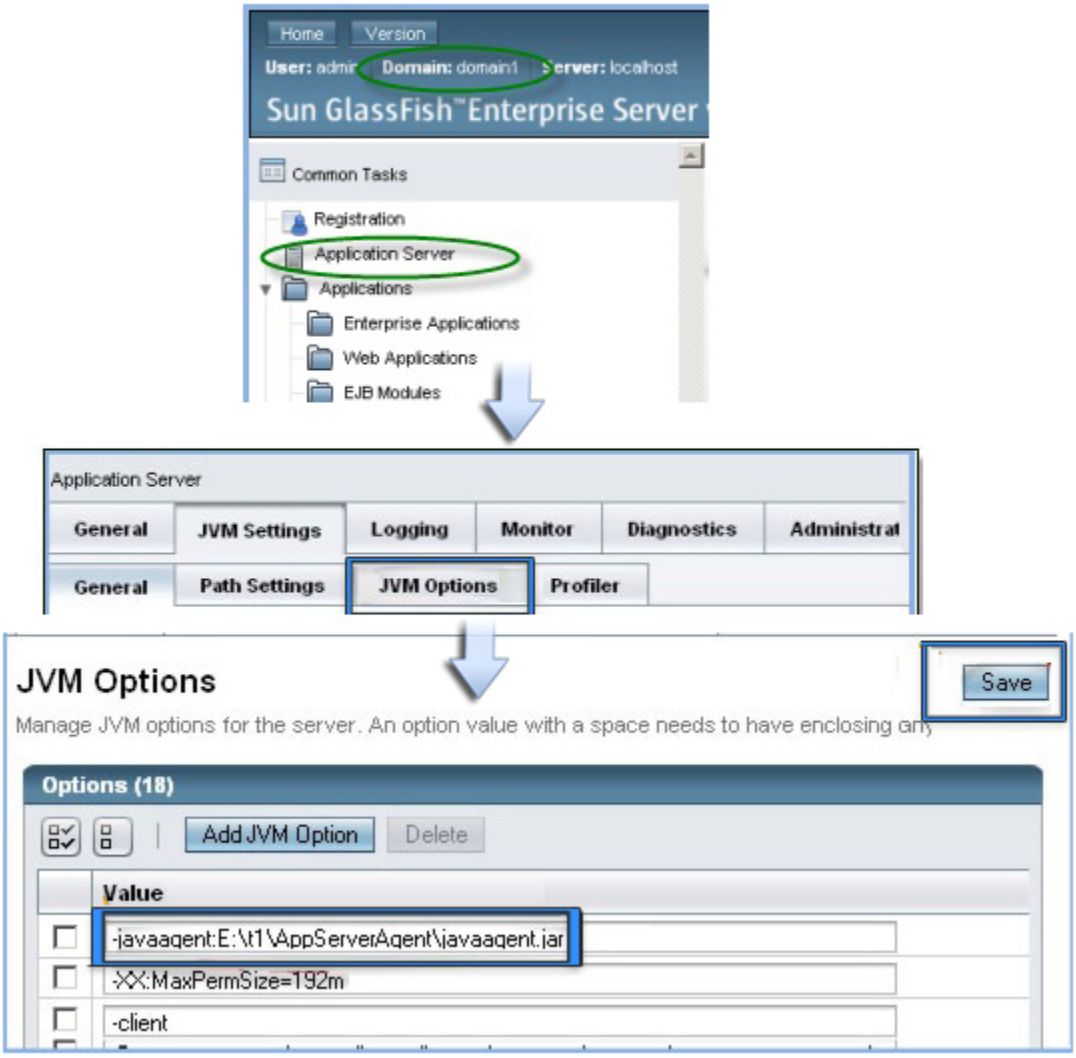
4. In the **JVM Settings** tab, click **JVM Options**.

5. Click **Add JVM Option** and add an entry for the javaagent argument. The javaagent argument contains the full path, including the drive, of the App Server Agent installation directory.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

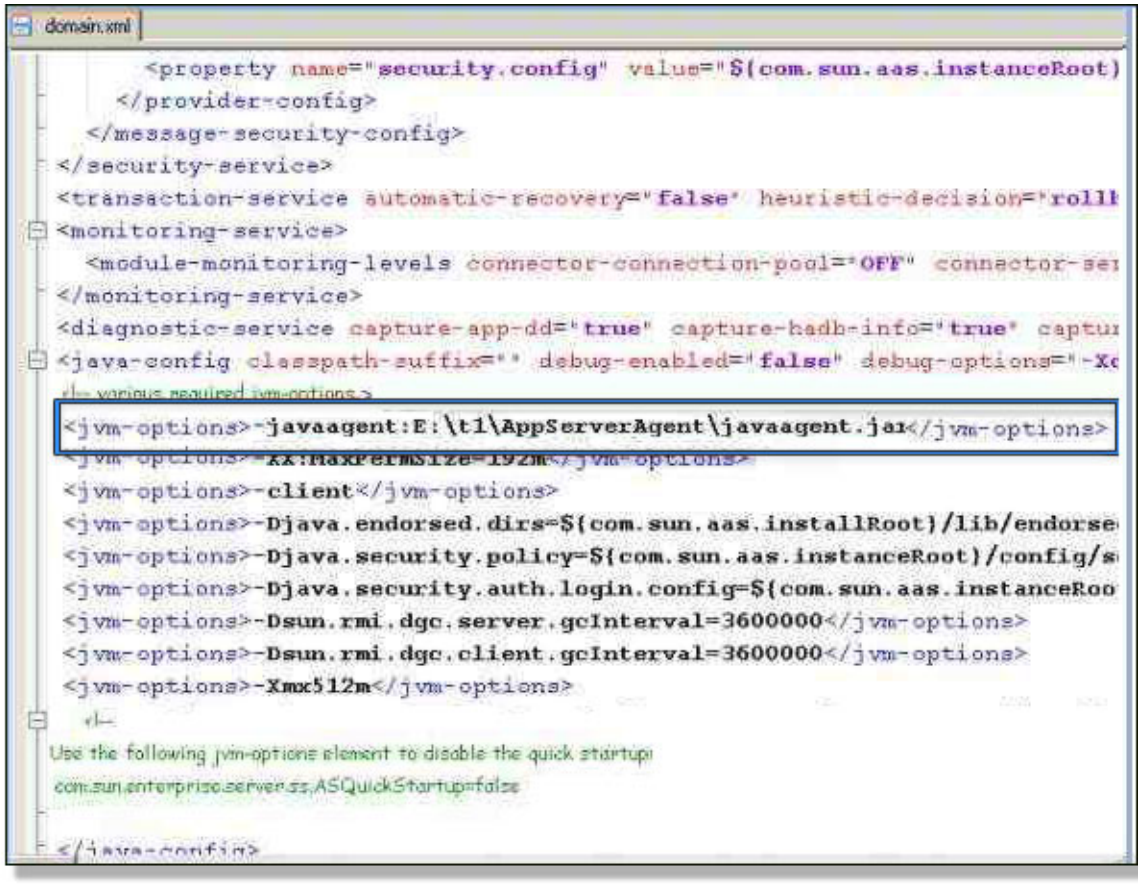

```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```



6. Restart the application server. The application server must be restarted for the changes to take effect.

To verify the Agent configuration

To verify this configuration, look at the domain.xml file located at <glassfish_install_dir>\domains\<domain_name>. The domain.xml file should have an entry as shown in the following screenshot.



About AppServer Management Extensions

AppDynamics does not support Glassfish AMX. AMX MBeans do not appear in the MBean Browser.

IBM WebSphere Startup Settings

- To add the javaagent command in a WebSphere environment
- To verify the Agent configuration
- Security permissions requirement

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option.

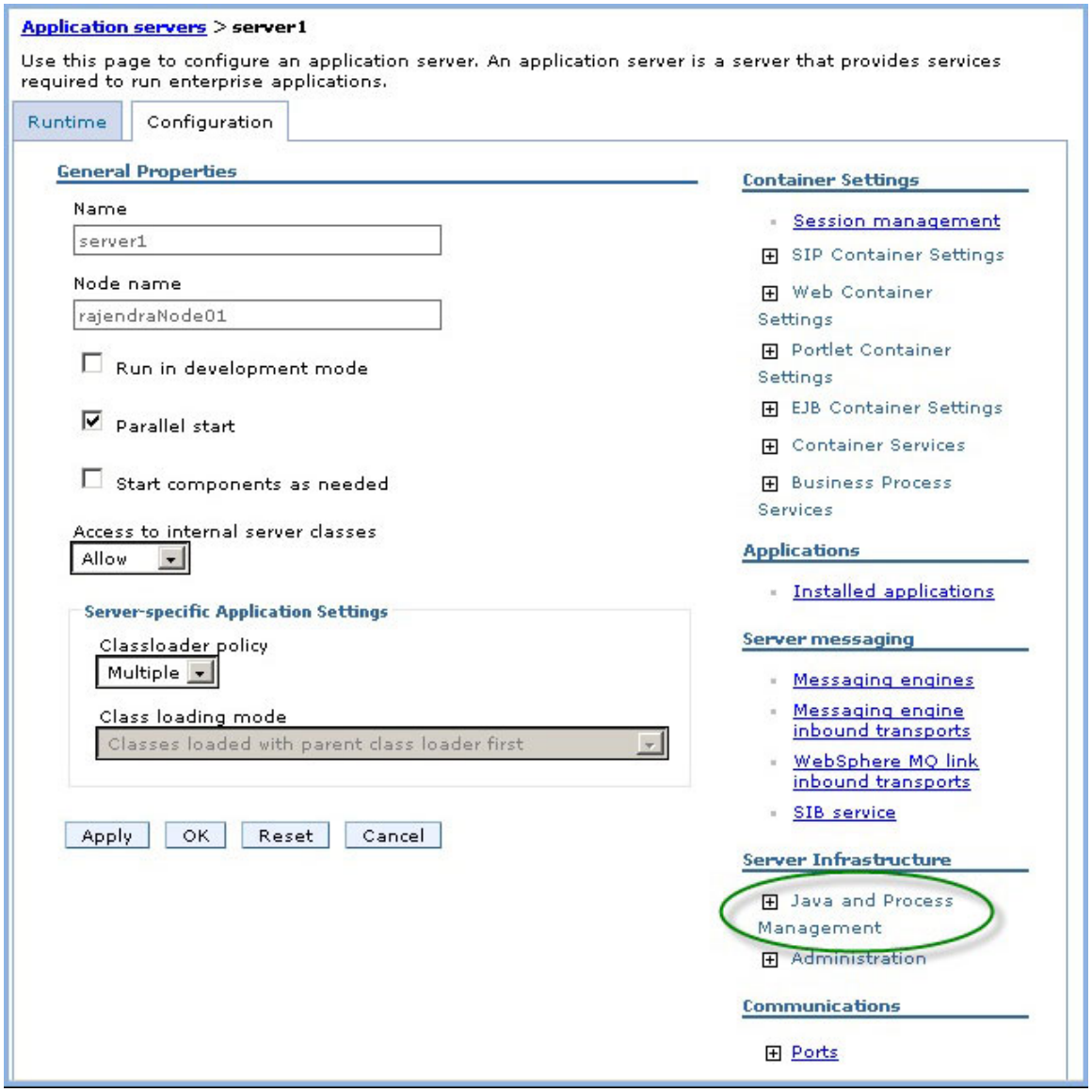
To add the javaagent command in a WebSphere environment

1. Log in to the **Administrator** console of the WebSphere node where you want to install the App Server Agent.

2. In the left navigation tree, click **Servers -> Application servers**.

3. Click the name of your server in the list of servers.

4. In the "**Configuration**" tab, click **Java and Process Management**.



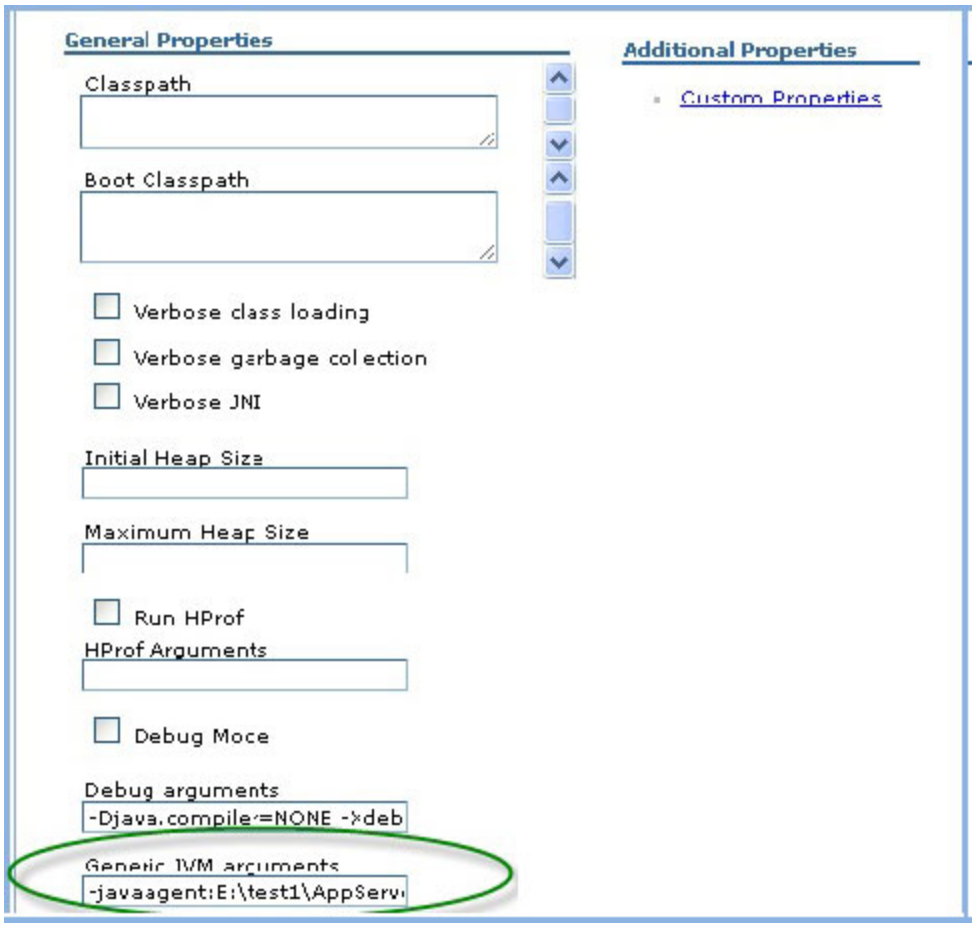
5. Enter the javaagent option with the full path to the AppDynamics javaagent.jar file in the **Generic JVM arguments** field.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>
```

6. Click OK.



To verify the Agent configuration

Verify the configuration settings by checking the server.xml file of the WebSphere node where you installed the App Server Agent. The server.xml file should have this entry:

```
<jvmEntries ...
genericJvmArguments='-javaagent:E:\test1\AppServerAgent\javaagent.jar'
cisableJIT="false"/>
```

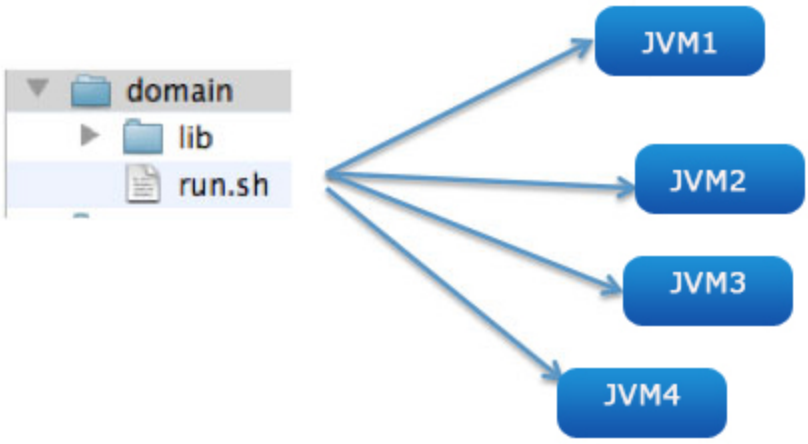
Security permissions requirement

Full permissions are required for the agent to function correctly with WebSphere. Grant all permissions on both the server level and the profile level.

App Agent for Java on z-OS or Mainframe Environments Configuration

- To configure the Agent to automatically generate node prefixes
- To remove the nodes that are not alive

In some environments JVMs have transient identity, such as when a single script spawns multiple JVMs.



For example, an environment may consist of WebSphere on IBM Mainframes, using a dynamic workload management feature that spawns new JVMs for an existing application server (called a servant). These JVMs are exact clones of an existing JVM, but each of them has a different process ID. Based on load, any number of additional JVMs may be created.

The App Server Agent can monitor each of these dynamically generated JVMs using the appdynamics.agent.auto.node.prefix option. You specify a node name prefix in your JVM startup script.

```
-Dappdynamics.agent.auto.node.prefix=<node name prefix>
```

To configure the Agent to automatically generate node prefixes

1. Add the application and tier name to "controller-info.xml" file for each JVM.

These JVMs belong to the **same** tier.

2. Add the javaagent argument and system properties (-D options) to the startup script of your JVM processes or to your Java Runtime Environment (JRE):

```
java -javaagent:<Agent-Installation-Directory>/javaagent.jar
-Dappdynamics.agent.reuse.nodeName=true
-Dappdynamics.agent.auto.node.prefix=$nodePrefix
```

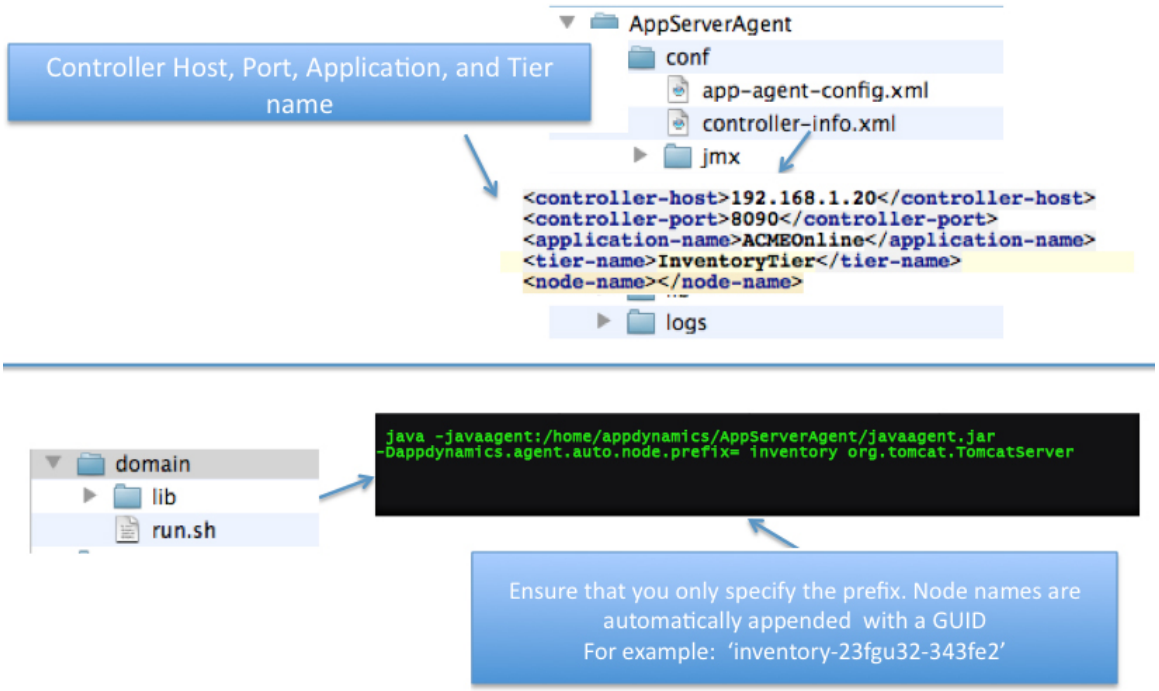
By default, the node names will be based on a serial number maintained by the Controller with a prefix of the tier name. If you need to change the prefix, specify the property "-Dappdynamics.agent.auto.node.prefix=$nodePrefix".

For example, if no prefix is specified for node on tier ECommerceTier, the node name will be ECommerceTier-1, ECommerceTier-2, etc. If a prefix is provided as in the example, the node names will be customPrefix-1, customPrefix-2, etc.

IMPORTANT:

- Ensure that you have not specified the node name anywhere (controller-info.xml file or as a system property in your JVMs start-up script).
- Ensure that all these system properties are separated by a white space character.
- These JVMs may or may not be short-lived.

The following illustration shows the sample configuration for ACME Bookstore. This configuration will create unique node names for every instance of the virtual machine starting up in the ACME Bookstore environment.

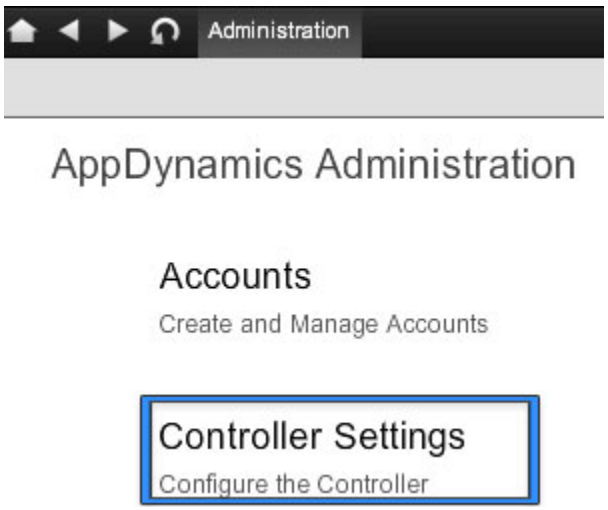


To remove the nodes that are not alive

In a typical environment, the nodes are recycled and new nodes get generated. AppDynamics strongly recommends that you remove the dead nodes both from the AppDynamics user interface (UI) as well as from the system.

1. Log in to the Controller administration console.

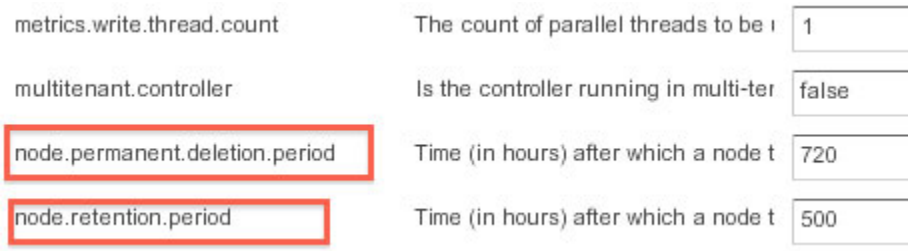
2. Go to "Controller Settings" section to see the advanced properties for the Controller.



3. Set the retention and deletion properties, based on the requirements for your environment. AppDynamics recommends that you set the permanent deletion period at least an hour more than the retention period.

- **node.permanent.deletion.period:** Time (in hours) after which a node that has lost contact with the Controller is deleted permanently from the system.
- **node.retention.period:** Time (in hours) after which a node that has lost contact with the Controller is deleted. In this case, the

AppDynamics UI will not display the node, however the system will continue to retain it.



JBoss Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment for JBoss 5.x
- To add the javaagent command in a Linux environment for JBoss AS 6.x
- To add the javaagent command in a Linux environment for JBoss AS 7.x
- To add the javaagent command in a Windows environment for JBoss AS 7.x

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your JBoss server run.sh or run.bat file.

To add the javaagent command in a Windows environment

1. Open the server run.bat file, located at <jboss_version_install_directory>\bin.

2. Add the following javaagent argument at the beginning of your app server start script.

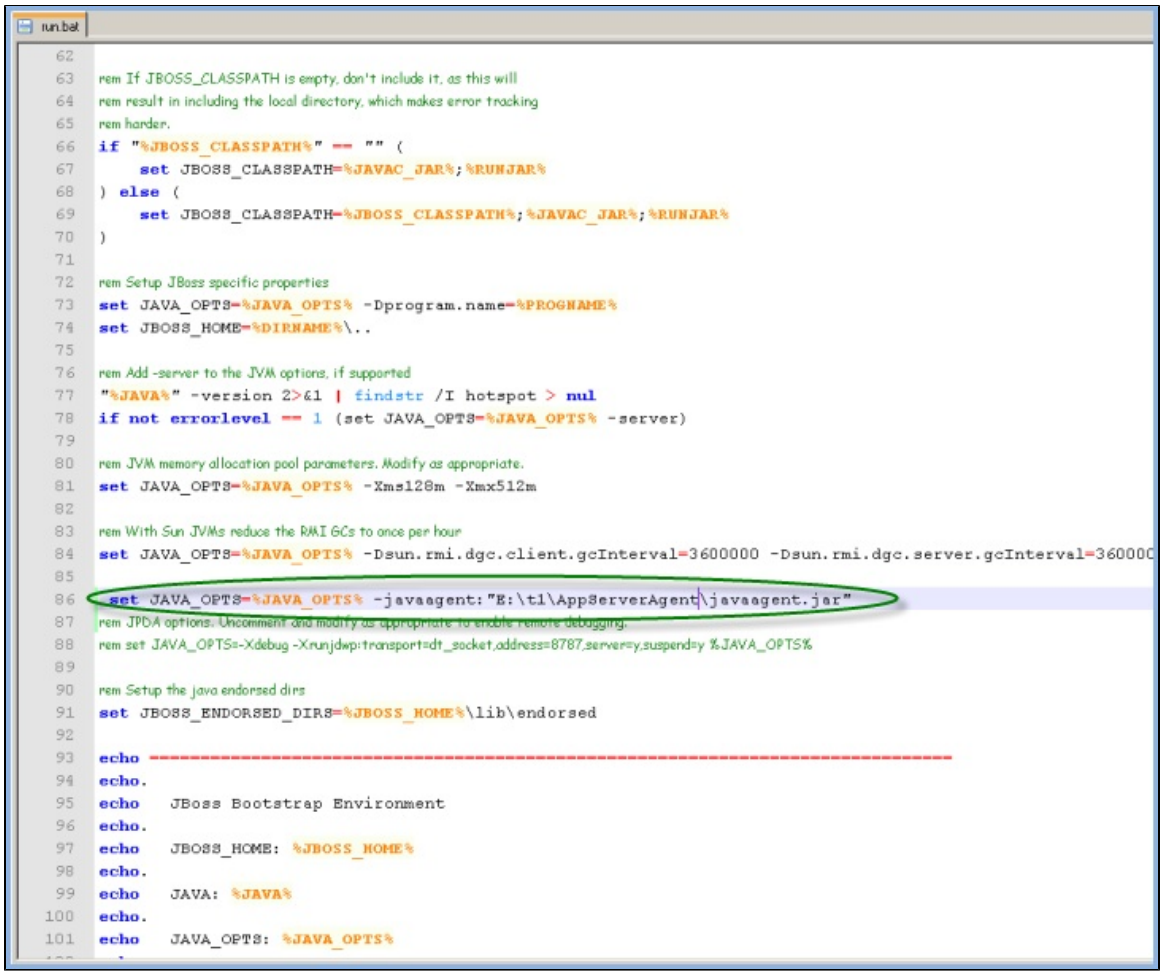
```
set JAVA_OPTS=%JAVA_OPTS% -javaagent:"<drive>:\<agent_home>\javaagent.jar"
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path of the App Server Agent installation directory, including the drive. For details see the screen captures.

2a. Sample JBoss 4.x run.bat file



2b. Sample JBoss 5.x run.bat file

```
run.bat
97
98  rem If JBOSS_CLASSPATH empty, don't include it, as this will
99  rem result in including the local directory in the classpath, which makes
100 rem error tracking harder.
101 if "x%JBOSS_CLASSPATH%" == "x" (
102   set "RUN_CLASSPATH=%RUNJAR%"
103 ) else (
104   set "RUN_CLASSPATH=%JBOSS_CLASSPATH%;%RUNJAR%"
105 )
106
107 set JBOSS_CLASSPATH=%RUN_CLASSPATH%
108
109 rem Setup JBoss specific properties
110 rem JVM memory allocation pool parameters. Modify as appropriate.
111 set JAVA_OPTS=%JAVA_OPTS% -Xms128m -Xmx512m -XX:MaxPermSize=256m
112
113 rem Warn when resolving remote XML dtd/schemas
114 set JAVA_OPTS=%JAVA_OPTS% -Dorg.jboss.resolver.warning=true
115
116 rem With Sun JVMs reduce the RMI GCs to once per hour
117 set JAVA_OPTS=%JAVA_OPTS% -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3
118
119 set JAVA_OPTS=%JAVA_OPTS% -javaagent:"E:\t1\AppServerAgent\javaagent.jar"
120 rem JPDA options. Uncomment and modify as appropriate to enable remote debugging.
121 rem set JAVA_OPTS=%JAVA_OPTS% -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y
122
123 rem Setup the java endorsed dirs
124 set JBOSS_ENDORSED_DIRS=%JBOSS_HOME%\lib\endorsed
125
126 echo ===============================================================================
127 echo.
128 echo   JBoss Bootstrap Environment
129 echo.
130 echo   JBOSS_HOME: %JBOSS_HOME%
131 echo.
132 echo   JAVA: %JAVA%
133 echo.
134 echo   JAVA_OPTS: %JAVA_OPTS%
135 echo.
136 echo   CLASSPATH: %JBOSS_CLASSPATH%
```

3. Restart the application server. The application server must be restarted for the changes to take effect.

To add the javaagent command in a Linux environment for JBoss 5.x

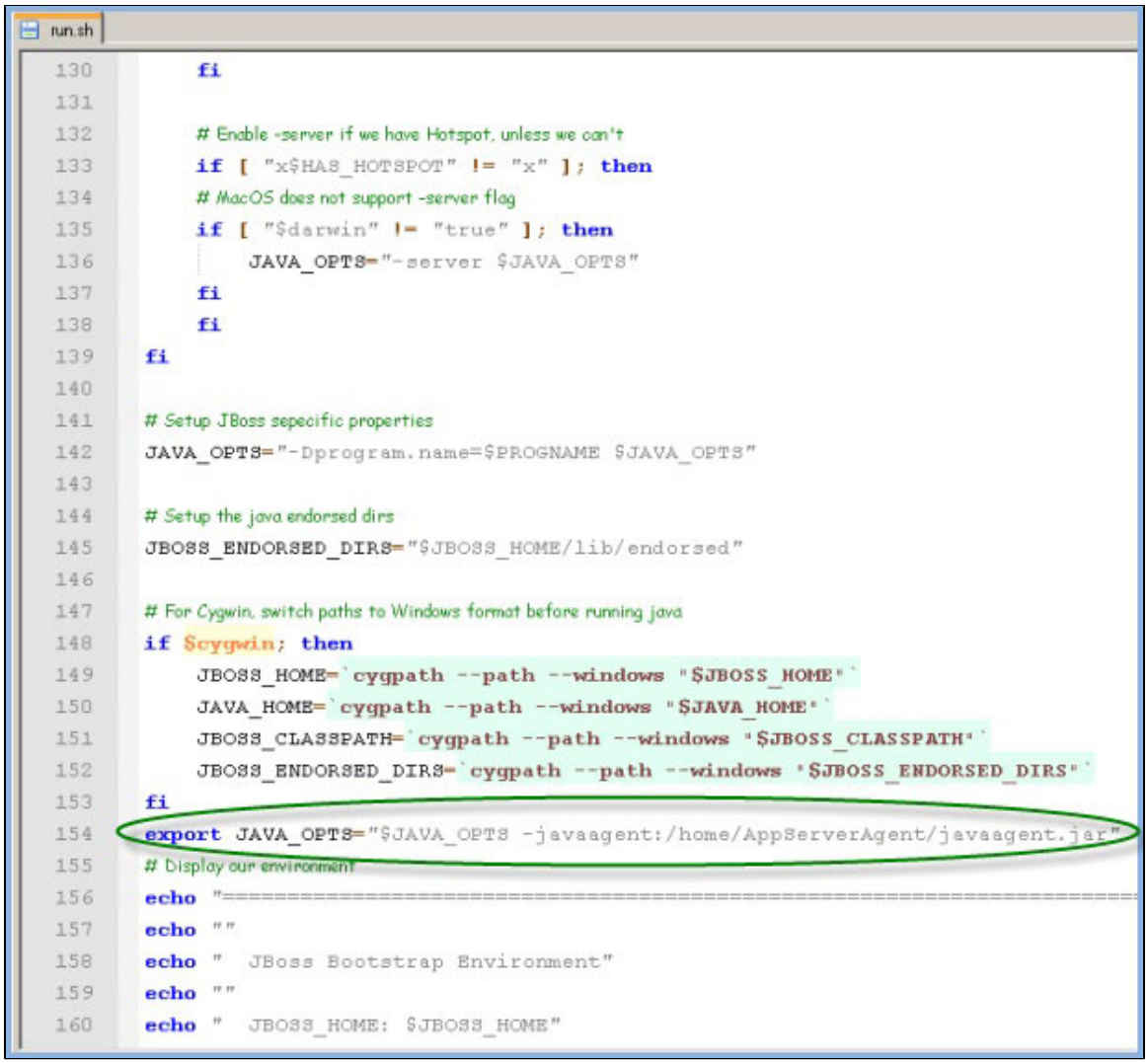
1. Open the server run.sh file, located at <jboss_version_install_dir>/bin.

2. Add the following javaagent argument to the server start script.

```
export JAVA_OPTS="$JAVA_OPTS -javaagent:/<agent_home>/javaagent.jar"
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

2a. Sample JBoss 5.x run.sh file



3. Restart the application server. The application server must be restarted for the changes to take effect.

To add the javaagent command in a Linux environment for JBoss AS 6.x

1. Open the server run.sh file, located at <jboss_version_install_dir>/bin.

2. Add the following Java environment variables to the server start script.

```
JAVA_OPTS="$JAVA_OPTS
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
JAVA_ARGS="$JAVA_OPTS
-Dorg.jboss.logging.Logger.pluginClass=org.jboss.logging.logmanager.LoggerPluginIr
```

3. Add the following javaagent argument to the server start script.

```
export JAVA_OPTS="$JAVA_OPTS -javaagent:/agent_install_dir/javaagent.jar"
```

4. Restart the application server. The application server must be restarted for the changes to take effect.

To add the javaagent command in a Linux environment for JBoss AS 7.x

1. Open the standalone.conf file.

2. Search for the following line in standalone.conf.

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman"
```

Add the com.singularity and org.jboss.logmanager packages to that line as follows:

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.singularity,org.jboss.logmanager"
```

For JBoss 7.1.1, add the additional packages as shown here:

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.appdynamics,com.appdynamics.,com
```

3. Add the following to the end of the standalone.conf file in the JAVA_OPTION section.

```
-Djava.util.logging.manager=org.jboss.logmanager.LogManager
-Xbootclasspath/p:<JBOSS-DIR>/modules/org/jboss/logmanager/main/jboss-logmanager-
```

Note: The path for the necessary JAR files may differ for different versions. Provide the correct path of these JAR files for your version. If any of the packages are not available with the JBoss ZIP, download the missing package and add it to the path.

4. In the standalone.sh file, add the following javaagent argument.

```
export JAVA_OPTS="$JAVA_OPTS -javaagent:/agent_install_dir/javaagent.jar"
```

above the following section of standalone.sh

```
...
while true;do
if [ "x$LAUNCH_JBOSS_IN_BACKGROUND" = "X" ]; then
   # Execute the JVM in the foreground
      eval \"$JAVA\" -D\"[Standalone]\"$JAVA_OPTS \
     \"-Dorg.jboss.boot.log.file=$JBOSS_LOG_DIR/boot.log\" \
     \"-Dlogging.configuration=file:$JBOSS_CONFIG_DIR/logging.properties\" \
      -jar \"$JBOSS_HOME/jboss-modules.jar\" \
```

The revised section of your startup script file should look similar to the following image:


```
173  echo ""
174
175  export JAVA_OPTS="$JAVA_OPTS -javaagent:/agent_install_dir/javaagent.jar"
176
177  while true; do
178     if [ "x$LAUNCH_JBOSS_IN_BACKGROUND" = "x" ]; then
179        # Execute the JVM in the foreground
180        eval \"$JAVA\" -D\"[Standalone]\" $JAVA_OPTS \
181           \"-Dorg.jboss.boot.log.file=$JBOSS_LOG_DIR/boot.log\" \
182           \"-Dlogging.configuration=file:$JBOSS_CONFIG_DIR/logging.properties\" \
183           -jar \"$JBOSS_HOME/jboss-modules.jar\" \
184           -mp \"${JBOSS_MODULEPATH}\" \
185           -jaxpmodule "javax.xml.jaxp-provider" \
186           org.jboss.as.standalone \
187           -Djboss.home.dir=\"$JBOSS_HOME\" \
188           "$@"
189        JBOSS_STATUS=$?
190     else
```

5. End User Monitoring (EUM) is not supported on JBoss 7 and you must disable it using the eum-disable-filter-injection agent configuration property. This property is available in AppDynamics App Agent for Java, version 3.5.2 and newer.
a. In the Controller UI left navigation pane, select **Servers -> App Servers<tier> -> <node>**.
b. Click the **Agents** tab and then the **App Server Agent** subtab.
c. Click **Configure**.
d. In the App Server Agent Configuration, select the node.
e. Click Add (the plus symbol) and enter the eum-disable-filter-injection property. Its type is Boolean and its value is "true".
f. Click **Save**.

6. Restart the application server. The application server must be restarted for the changes to take effect.

To add the javaagent command in a Windows environment for JBoss AS 7.x

1. Open the bin\standalone.conf file.

2. Search for the line JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman" and the com.singularity ad org.jboss.logmanager packages to that line as follows:

```
JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman,com.singularity,org.jboss.logmanager"
```

3. Open the bin\standalone.conf.bat file.

4. Search for the following line:
[code}
set "JAVA_OPTS=%JAVA_OPTS% -Djboss.modules.system.pkgs=org.jboss.byteman

```

```

Add the package com.singularity to that line as follows:

```
set "JAVA_OPTS=%JAVA_OPTS%
-Djboss.modules.system.pkgs=org.jboss.byteman,com.singularity"
```

5. Save the file.
6. Open the standalone.bat file.
7. Add the following javaagent argument to the standalone.bat file.


```
:RESTART
"%JAVA%" -javaagent:<AGENT-DIR>javaagent.jar %JAVA_OPTS% ^
"-Dorg.jboss.boot.log.file=%JBOSS_HOME%\standalone\log\boot.log" ^
"-Dlogging.configuration=file:%JBOSS_HOME%/standalone/configuration/logging.proper
^
-jar "%JBOSS_HOME%\jboss-modules.jar" ^
```

8. Save the file.
9. End User Monitoring (EUM) is not supported on JBoss 7 and you must disable it using the eum-disable-filter-injection agent configuration property. This property is available in AppDynamics App Agent for Java, version 3.5.2 and newer.
a. In the Controller UI left navigation pane, select **Servers -> App Servers<tier> -> <node>**.
b. Click the **Agents** tab and then the **App Server Agent** subtab.
c. Select the node.
d. Click Add (the plus symbol) and enter the eum-disable-filter-injection property. Its type is Boolean and its value is "true".
e. Click **Save**.

10. Restart the application server. The application server must be restarted for the changes to take effect.

Jetty Startup Settings

- To add the javaagent command in a Jetty environment

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your jetty.sh file.

To add the javaagent command in a Jetty environment

1. Open the jetty.sh start script file.

2. Add the following javaagent argument to the beginning of the script.

```
java -javaagent:/<agent_home>/javaagent.jar
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

3. Save the script file.

4. Restart the application server for the changes to take effect.

Oracle WebLogic Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your startWebLogic.sh or startWebLogic.cmd file.

To add the javaagent command in a Windows environment

1. Open the startWebLogic.cmd file, located at <weblogic_version_install_dir>\user_projects\domains\<domain_name>\bin.

2. Add following javaagent argument to the beginning of your application server start script.

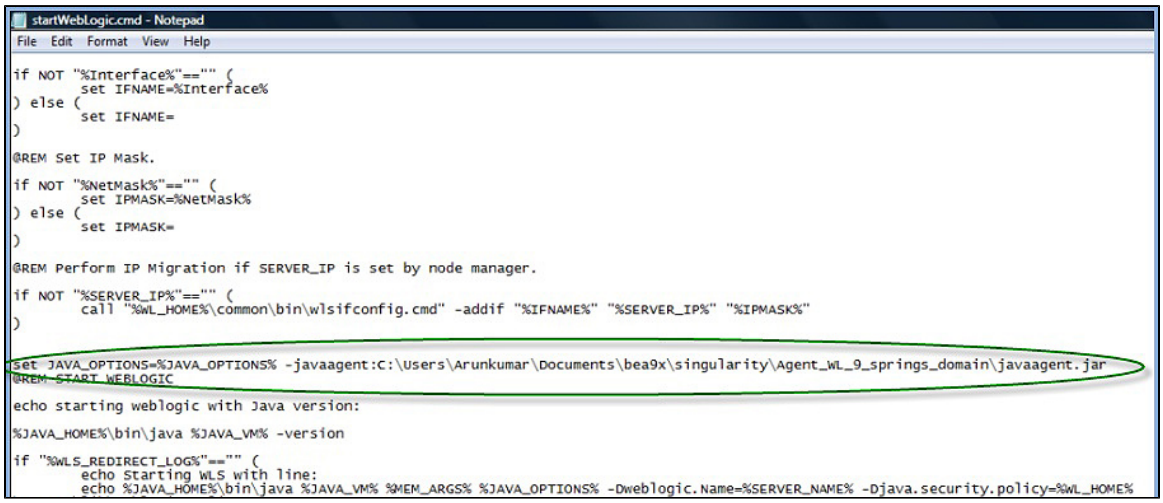

```
set JAVA_OPTIONS=% JAVA_OPTIONS%
-javaagent:"<drive>:\<agent_home>\javaagent.jar"
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path of the App Server Agent installation directory, including the drive.

2a. Sample WebLogic v9.x startWebLogic.cmd file



2b. Sample WebLogic v10.x startWebLogic.cmd file

startWebLogic.cmd

```
165 ) else (
166     set IPMASK=
167 )
168
169 @REM Perform IP Migration if SERVER_IP is set by node manager.
170
171 if NOT "%SERVER_IP%"=="" (
172     call "%WL_HOME%\common\bin\wlsifconfig.cmd" -addif "%IFNAME%" "%SERVER_IP%" "%IPMASK%"
173 )
174
175 set JAVA_OPTIONS=%JAVA_OPTIONS% -javaagent:"E\t1\AppServerAgent\javaagent.jar"
176 @REM START WEBLOGIC
177
178 echo starting weblogic with Java version:
179
180 %JAVA_HOME%\bin\java %JAVA_VM% -version
181
182 if "%WLS_REDIRECT_LOG%"=="" (
183     echo Starting WLS with line:
184     echo %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_
185     %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_NAME%
186 ) else (
187     echo Redirecting output from WLS window to %WLS_REDIRECT_LOG%
188     %JAVA_HOME%\bin\java %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% -Dweblogic.Name=%SERVER_NAME%
189 )
```

3. Restart the application server. The application server must be restarted for the changes to take effect.

To add the javaagent command in a Linux environment

1. Open the startWebLogic.sh file, located at <weblogic_<version#>_install_dir>/user_projects/domains/<domain_name>/bin.

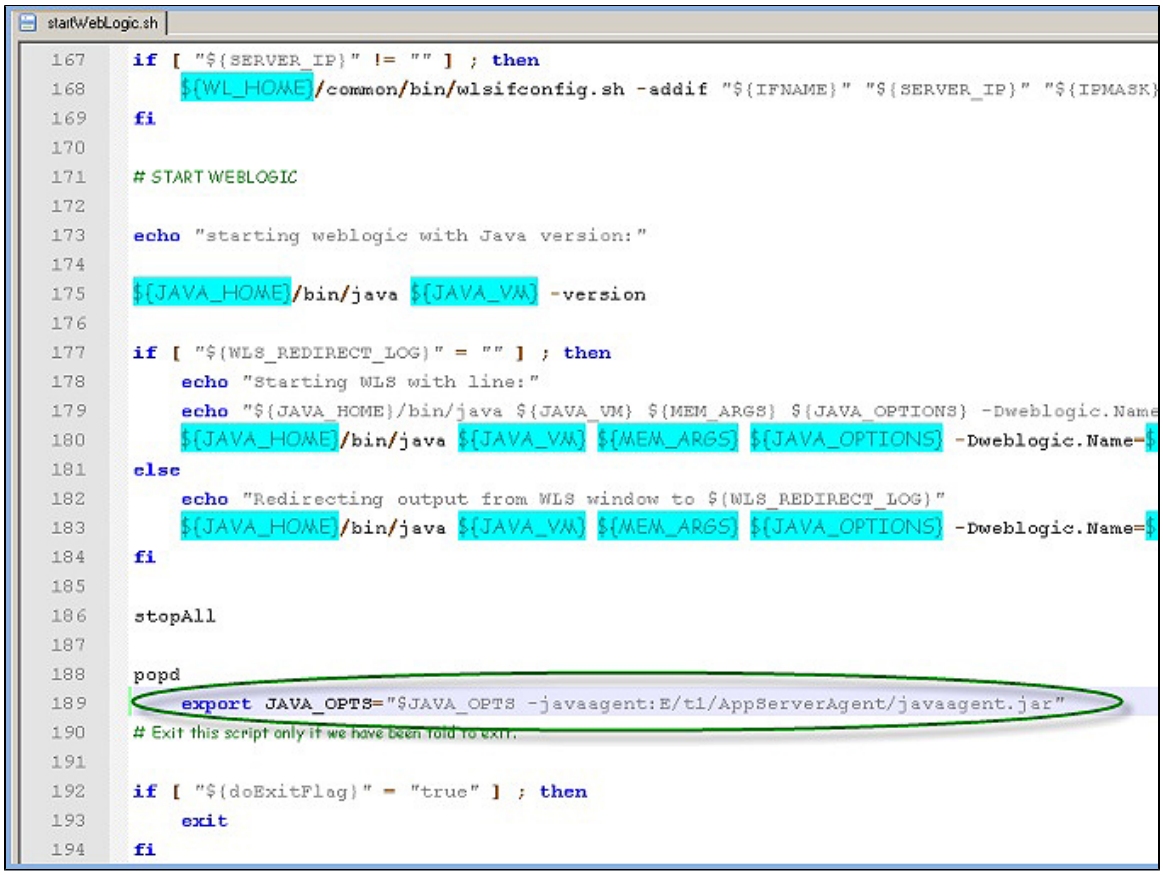
2. Add the following lines of code to the beginning of your application server start script.

```
export JAVA_OPTIONS="$JAVA_OPTIONS -javaagent:/agent_home/javaagent.jar"
```

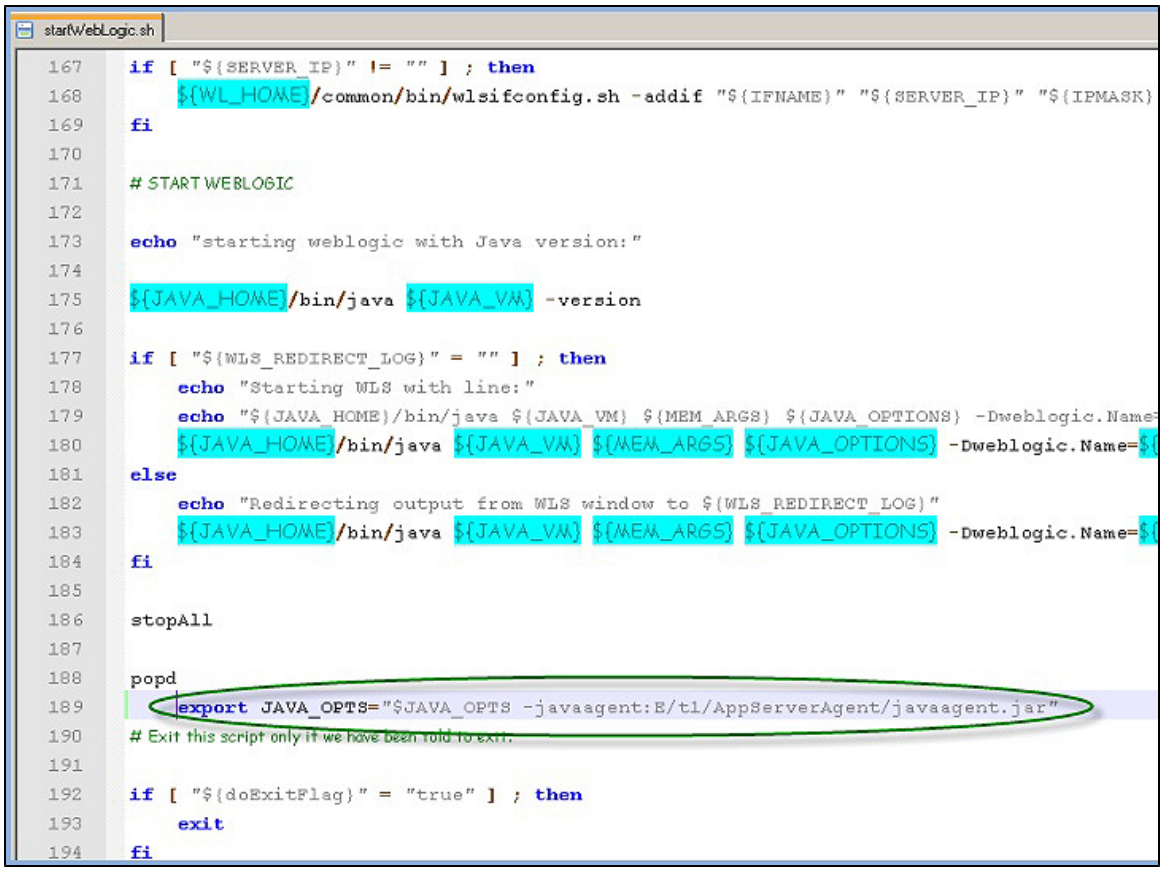
If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

The javaagent argument references the full path of the App Server Agent installation directory. For details see the screen captures.

2a. Sample WebLogic v9.x startWebLogic.sh file



2b. Sample WebLogic v10.x startWebLogic.sh file



3. Restart the application server. The application server must be restarted for the changes to take effect.

OSGi Infrastructure Configuration

- Configuring OSGi Containers
  - To configure Equinox
  - To configure Apache Sling
  - To configure Felix for GlassFish
    - For GlassFish 3.1.2
  - To configure Felix for Jira or Confluence
    - Jira 5.x and newer
  - To configure other OSGi-based containers

Configuring OSGi Containers

The GlassFish application server versions 3.x and later uses OSGi architecture. By default, OSGi containers follow a specific model for bootstrap class delegation. Classes that are not specified in the container's CLASSPATH are not delegated to the bootstrap classloader; therefore you must configure the OSGi containers for the App Server Agent classes.

For more information see GlassFish OSGi Configuration per Domain.

To ensure that the OSGi container identifies the agent, specify the following package prefix:

```
org.osgi.framework.bootdelegation=com.singularity.*
```

This prefix follows the regular boot delegation model so that the App Server Agent classes are visible.

If you already have existing boot delegations, add "com.singularity.*" to the existing path separated by a comma. For example:

org.osgi.framework.bootdelegation=com.sun.btrace.**, com.singularity.**

To configure Equinox

1. Open the config.ini file located at <glassfish-install>/glassfish/osgi/equinox/configuration.

2. Add following package prefix to the config.ini file:

```
org.osgi.framework.bootdelegation=com.singularity.*
```

For more information see Getting Started with Equinox.

To configure Apache Sling

1. Open the sling.properties file. The location of the sling.properties varies depending on the Java platform.
In the Sun/Oracle implementation, the sling.properties file is located at <java.home>/lib.

2. Add following package prefix to the sling.properties file.

```
org.osgi.framework.bootdelegation=com.singularity.*
```

To configure Felix for GlassFish

1. Open the config.properties file, located at <glassfish-install>/glassfish/osgi/felix/conf.

2. Add following package prefix to the config.properties file.

```
org.osgi.framework.bootdelegation=com.singularity.*
```

For GlassFish 3.1.2

Add:

```
com.singularity.*
```

to the boot delegation list in the <GlassFish_Home_Directory>\glassfish\config\osgi.properties file. For example:

```
org.osgi.framework.bootdelgation=${eclipselink.bootdelegation}, com.sun.btrace,
com.singularity.*
```

To configure Felix for Jira or Confluence

1. From <atlassian-install>/bin (Stand-alone) or <Tomcat-home>/bin (EAR-WAR installation), open:

- For Linux: setenv.sh
- For Windows: setenv.bat

2. Update the Java options:

- For Linux: JAVA_OPTS=
- For Windows: set JAVA_OPTS=%JAVA_OPTS%


```
-javaagent:<path>/javaagent.jar
-Datlassian.org.osgi.framework.bootdelegation=<other_services>,com.singularity,
com.singularity.*,<other_services>
```

Jira 5.x and newer

```
-Datlassian.org.osgi.framework.bootdelegation=META-INF.services,com.yourkit,com.yo
```

To configure other OSGi-based containers

For other OSGI-based runtime containers, add the following package prefix to the appropriate OSGi configuration.

```
file.org.osgi.framework.bootdelegation=com.singularity.*
```

Resin Startup Settings

- To Configure Resin 1.x - 3.x
  - To add the javaagent command in a Windows environment
  - To add the javaagent command in a Linux environment
- To Configure Resin 4.x

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to the resin.sh or resin.bat file.

To Configure Resin 1.x - 3.x

To add the javaagent command in a Windows environment

1. Open the resin.bat file, located at <Resin_installation_directory>/bin.

2. Add following javaagent argument to the beginning of your application server start script.

```
exec JAVA_EXE -javaagent:"<drive>:\<agent_home>\javaagent.jar"
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path of the App Server Agent installation directory, including the drive.

3. Restart the application server for changes to take effect.

To add the javaagent command in a Linux environment

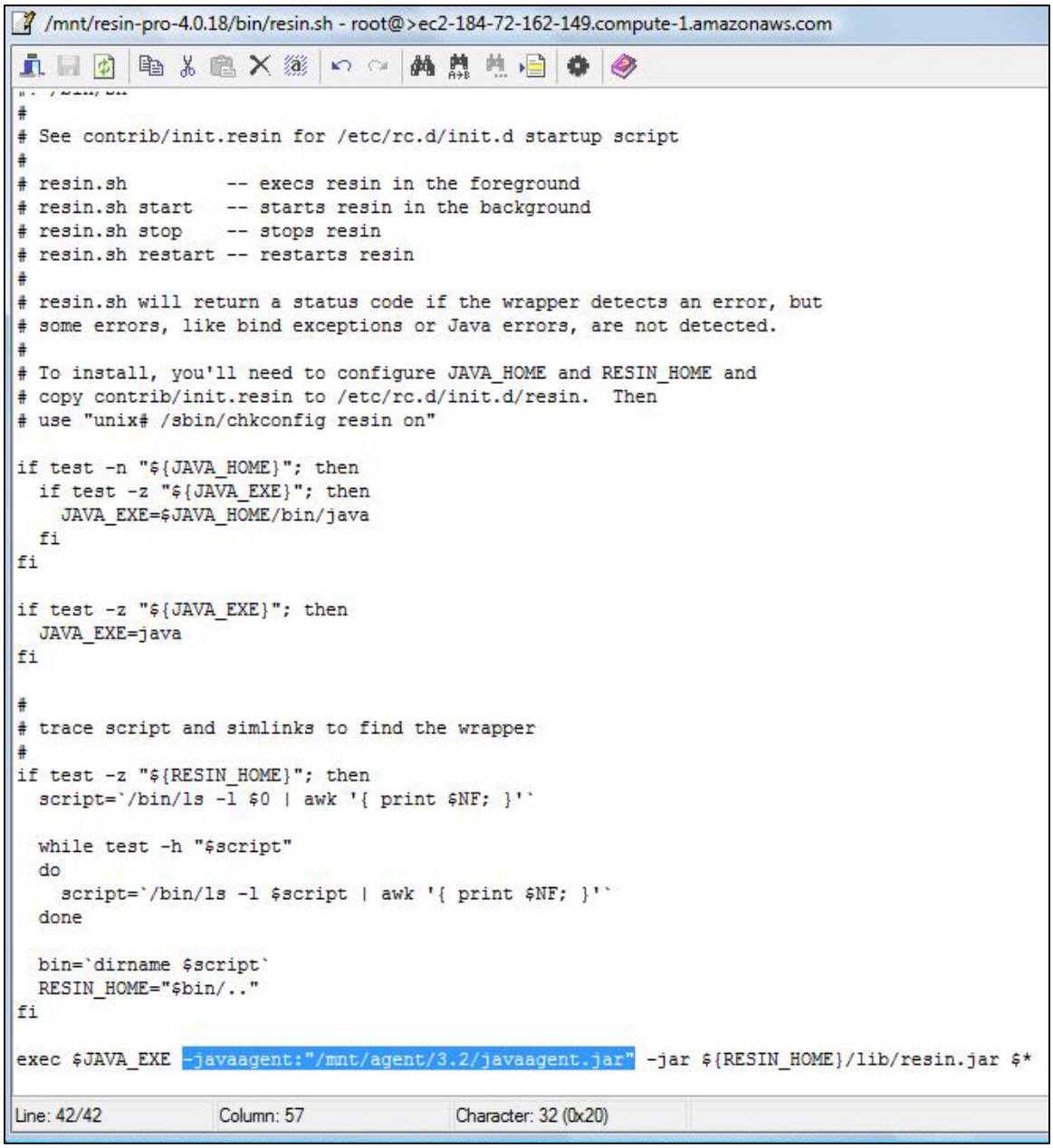
1. Open the resin.sh file, located at <Resin_Installation_Directory>/bin.

2. Add the following javaagent argument to the beginning of your application server start script.


```
exec $JAVA_EXE -javaagent:"<agent_home>/javaagent.jar"
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

The javaagent argument references the full path of the App Server Agent installation directory. See the following screenshot.



3. Restart the application server. The application server must be restarted for the changes to take effect.

To Configure Resin 4.x

1. To install the App Server Agent into Resin 4.X or later, edit the ./conf/resin.xml file and add:


```
<jvm-arg>-Xmx512m</jvm-arg>
<jvm-arg>-javaagent:<$appagent_location>/javaagent.jar</jvm-arg>
```

2. Restart the application server. The application server must be restarted for the changes to take effect.

Solr Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to your Solr server.

To add the javaagent command in a Windows environment

1. Open the Windows command line utility.

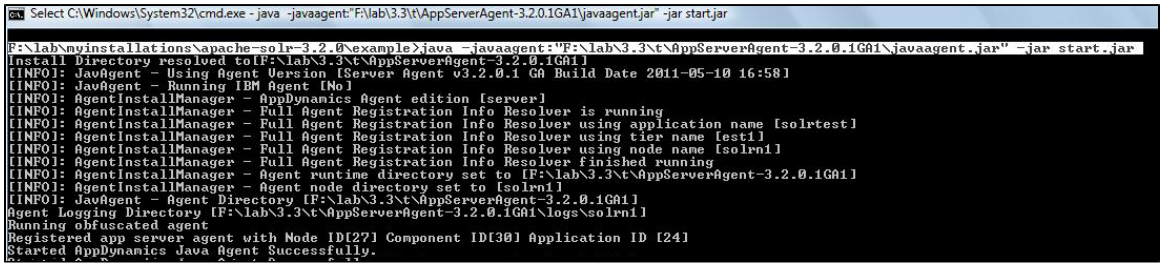
2. Execute the following commands to add the javaagent argument to the Solr server:

```
>cd $Solr_Installation_Directory
>java -javaagent:"<drive>:\<agent_home>\javaagent.jar" -jar start.jar
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path to the App Server Agent installation directory, including the drive. For details see the screenshots.



To add the javaagent command in a Linux environment

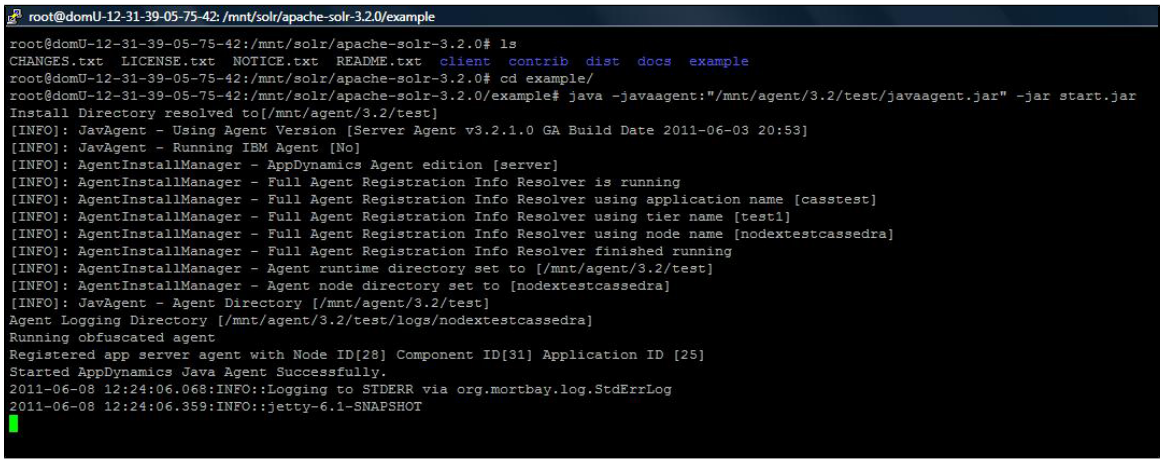
1. Open the terminal.

2. Execute the following commands to add the javaagent argument to the Solr server:

```
>cd $Solr_Installation_Directory
>java -javaagent:"<agent_home>/javaagent.jar" -jar start.jar
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

The javaagent argument references the full path to the App Server Agent installation directory. For details see the screenshot.



Standalone JVM Startup Settings

- To add the javaagent command in a Windows environment
- To add the javaagent command in a Linux environment

AppDynamics works just as well with JVMs that are not application servers or containers.

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option, a standard Java option and can be used with any JVM. Add this option to your standalone JVM.

To add the javaagent command in a Windows environment

1. Open the command line utility for Windows.

2. Add javaagent argument to the standalone JVM:

```
>java -javaagent:"Drive:\agent_home\javaagent.jar"
<fully_qualified_class_name_with_main_method>
```

For example:

```
>java -javaagent:"C:\AppDynamics\agentDir\javaagent.jar"  com.main.HelloWorld
```

If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

```
-javaagent:"<drive>:\<agent_home>\javaagent.jar=uniqueID=<my-app-jvm1>"
```

The javaagent argument references the full path to the App Server Agent installation directory, including the drive.

To add the javaagent command in a Linux environment

1. Open the terminal.

2. Add the javaagent argument to the standalone JVM:


```
>java -javaagent:"/agent_install_dir/javaagent.jar"
<fully_qualified_class_name_with_main_method>
```

For example:

```
>java -javaagent:"/mnt/AppDynamics/agentDir/javaagent.jar" com.main.HelloWorld
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

The javaagent argument references the full path to the App Server Agent installation directory.

Tanuki Service Wrapper Configuration

- To configure the Tanuki Service Wrapper

The AppDynamics Java App Server Agent bootstraps using the javaagent command line option. Add this option to the Tanuki Service wrapper.conf file.

To configure the Tanuki Service Wrapper

1. Open the wrapper.conf file.

2. Use the wrapper.java.additional.<n> property to add the javaagent option.

```
wrapper.java.additional.6=-javaagent:/C:/agent/javaagent.jar
```

 If you are a Self-Service Trial user, add the App Agent for Java javaagent argument to your JVM start script where <my-app-jvm1> is the name you use for the application running on that JVM.

For more information see:

- Tanuki Service Wrapper Properties
- Example Configuration
- More Help On Tanuki Service Wrapper

Tibco BusinessWorks Configuration

There are typically two scripts associated with the Tibco BusinessWorks Services engine.

- my_application.sh
- my_application.tra

The JVM that runs the services start with a command line tool called bwengine(.exe).

Add the following to the .tra file:

```
Memory
java.extended.properties -javaagent:/opt/appagent/javaagent.jar
```


See also: https://ssl.tibcommunity.com/thread/14856

Upgrade the App Agent for Java

This topic provides instructions for upgrading the App Agent for Java.

Upgrade Sequence

If you are upgrading both the Controller and agents, first upgrade the Controller and then upgrade the Agents.

Upgrade Note: If you upgrade the Java App Server Agent to 3.3.3 and your environment has a JDBC URL greater than 500 characters, you must upgrade your Controller to version 3.3.3.

To upgrade the App Agent for Java

1. Shut down the application server where the App Agent for Java and the optional machine agent is installed.

2. Create a backup copy of the current agent installation directory and move the backup directory to a new location.

3. Download the latest release from AppDynamics Download Center.

4. Extract the agent binaries. Where you extract them depends on the version you are upgrading from.

a. **From 3.1.x or earlier**: extract the Agent binaries to a new directory and rename old directory. Then rename the new directory the same name as the original one. Copy the controller-info.xml from the old Agent directory to the new Agent directory. At the end, you should have the new files using the same directory path as the previous one.

b. **From 3.2.x or later**: extract in the same directory. AppDynamics takes care of potential naming conflicts.

5. If you previously made changes to the <App_Server_Agent_Installation_Directory>/conf/app-agent-config.xml file, copy those changes to the new file.

Using the same directory path avoids the task of manually changing the agent-related configurations in your JVM startup script.

6. Restart the application server.

Uninstall the App Agent for Java

- To uninstall the Java Agent
- Learn More

This topic describes how to uninstall the App Agent for Java.

When a JVM is no longer instrumented by the App Agent for Java, it no longer uses a license.

To uninstall the Java Agent

1. Stop the application server on which the App Agent for Java is configured.

2. Remove the -javaagent argument in startup script of the JVM.

3. Remove any system properties configured for the App Agent for Java from the startup script of your JVM.

4. Restart the application server.

Learn More

- Manage App Agents
- Install the App Agent for Java
- App Agent for Java Configuration Properties

Administer App Agents for Java

App Agent for Java Configuration Properties

- Where to Configure App Agent Properties
  - Creating and Registering Tiers
- Example Java App Agent controller-info.xml File
- Example Startup-up Using System Properties
- Java App Server Agent Properties
  - Agent-Controller Communication Properties
    - Controller Host Property
    - Controller Port Property
  - Agent Identification Properties
    - Application Name Property
    - Tier Name Property
    - Node Name Property
  - Multi-Tenant Mode Properties
    - Account Name Property
    - Account Access Key Property
  - Proxy Properties for the Controller
    - Proxy Host Property
    - Proxy Port Property
  - Other Properties
    - Controller SSL Enabled Property
    - Enable Orchestration Property
    - Agent Runtime Directory Property
    - Redirect Logfiles Property
    - Force Agent Registration Property
    - Reuse Node Name Property
    - Auto Node Name Prefix Property
    - Chron/Batch JVM Property
    - Unique Host ID Property
- Learn More

Where to Configure App Agent Properties

You can configure the App Server Agent properties:

- in the controller-info.xml file in the <Agent_Installation_Directory>/conf directory
- in the system properties (-D options) in the JVM startup script

The system properties override the settings in the controller-info.xml file.

For shared binaries among multiple JVM instances, AppDynamics recommends using a combination of the xml file and the start-up properties to configure the app agent. Configure all the properties common to all the JVMs in the controller-info.xml file. Configure the properties unique to a JVM using the system properties in the start-up script.

For example:

- For multiple JVMs belonging to the same application serving different tiers, configure the application name in the controller-info.xml file and the tier name and node name using the system properties.
- For multiple JVMs belonging to the same application and the same tier, configure the application name and the tier name in the controller-info.xml file and the node name using the system properties.

After you configure agent properties, confirm that the javaagent argument has been added to the JVM startup script. For more information, see Java Server-Specific Installation Settings.

For some properties, you can use system properties already defined in the start-up script as the App Server Agent property values. For more information, see Configure App Agent for Java to Use Existing System Properties.

Creating and Registering Tiers

You can create a tier in the Controller prior to setting up any agents. Alternatively, an agent can register its tier with the Controller the first time, and only the first time, that it connects with the Controller. If a tier with the name used to connect already exists, the agent is associated with the existing tier.

Example Java App Agent controller-info.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<controller-info>

 <controller-host>192.168.1.20</controller-host>

 <controller-port>8090</controller-port>

 <controller-ssl-enabled>false</controller-ssl-enabled>

 <application-name>ACMEOnline</application-name>

 <tier-name>InventoryTier</tier-name>

 <node-name>Inventory1</node-name>

 <agent-runtime-dir></agent-runtime-dir>

 <enable-orchestration>false</enable-orchestration>

 <account-name></account-name>
 <account-access-key></account-access-key>

 <force-agent-registration>false</force-agent-registration>

</controller-info>
```

Example Startup-up Using System Properties

The following command uses the system properties to start the agent that monitors the ACME Online sample application's Inventory tier.

```
java -javaagent:/home/appdynamics/AppServerAgent/
-DappDynamics.controller.hostName=192.168.1.20 -DappDynamics.controller.port=8090
-DappDynamics.agent.applicationName=ACMEOnline -DappDynamics.agent.tierName=Inventory
-DappDynamics.agent.nodeName=Inventory1 SampleApplication
```

Java App Server Agent Properties

This section describes the Java App Agent configuration properties, including their controller-info-xml elements and their system property options.

Agent-Controller Communication Properties

Controller Host Property

Description: This is the host name or the IP address of the AppDynamics Controller. Example values are 192.168.1.22 or myhost or myhost.abc.com. This is the same host that you use to access the AppDynamics browser-based user interface. For an on-premise Controller, use the value for Application Server Host Name that was configured when the Controller was installed. If you are using the AppDynamics SaaS Controller service, see the Welcome email from AppDynamics.

Element in controller-info.xml: <controller-host>

System Property: -Dappdynamics.controller.hostName

Type: String

Default: None

Required: Yes, if the Enable Orchestration property is false.

If Enable Orchestration is true, and if the app agent is deployed in a compute cloud instance created by an AppDynamics workflow, do not set the Controller host unless you want to override the auto-detected value. See Enable Orchestration Property.

Controller Port Property

Description: This is the HTTP(S) port of the AppDynamics Controller. This is the same port that you use to access the AppDynamics browser-based user interface.

If the Controller SSL Enabled property is set to true, specify the HTTPS port of the Controller; otherwise specify the HTTP port. See Controller SSL Enabled Property.

Element in controller-info.xml: <controller-port>

System Property: -Dappdynamics.controller.port

Type: Positive Integer

Default: For On-premise installations, port 8090 for HTTP and port 8181 for HTTPS are the defaults.
For the SaaS Controller Service, port 80 for HTTP and port 443 for HTTPS are the defaults.

Required: Yes, if the Enable Orchestration property is false.

If Enable Orchestration is true, and if the app agent is deployed in a compute cloud instance created by an AppDynamics workflow, do not set the Controller port unless you want to override auto-detected value. See Enable Orchestration Property.

Agent Identification Properties

Application Name Property

Description: This is the name of the logical business application that this JVM node belongs to. Note that this is not the deployment name(ear/war/jar) on the application server.
If a business application of the configured name does not exist, it is created automatically.

Element in controller-info.xml: <application-name>

System Property: -Dappdynamics.agent.applicationName

Type: String

Default: None

Required: Yes

Tier Name Property

Description: This is the name of the logical tier that this JVM node belongs to. Note that this is not the deployment name (ear/war/jar) on the application server.

If the JVM / AppServer start-up script already has a system property that references the tier, such as -Dserver.tier, you could use ${server.tier} as the tier name. For more information, see Configure App Agent for Java to Use Existing System Properties.

See Name Business Applications, Tiers, and Nodes.

Element in controller-info.xml: <tier-name>

System Property: -Dappdynamics.agent.tierName

Type: String

Default: None

Required: Yes

Node Name Property

Description: This is the name of JVM node.

Where JVMs are dynamically created, use the system property to set the node name.

If your JVM / AppServer start-up script already has a system property that can be used as a node name, such as -Dserver.name, you could use ${server.name} as the node name. You could also use expressions such as ${server.name}_${host.name}.MyNode to define the node name. See Configure App Agent for Java to Use Existing System Properties for more information.

In general, the node name must be unique within the business application and physical host. If you want to use the same node name for multiple nodes on the same physical machine, create multiple virtual hosts using the Unique Host ID property. See Unique Host ID Property.

See Name Business Applications, Tiers, and Nodes.

Element in controller-info.xml: <node-name>

System Property: -Dappdynamics.agent.nodeName

Type: String

Default: None

Required: Yes

Multi-Tenant Mode Properties

Description: If the AppDynamics Controller is running in multi-tenant mode or if you are using the AppDynamics SaaS Controller, specify the account name and account access key for this agent to authenticate with the Controller. If you are using the AppDynamics SaaS Controller, the account name is provided in the Welcome email sent by AppDynamics.

If the Controller is running in single-tenant mode (the default) there is no need to configure these values.

Account Name Property

Description: This is the account name used to authenticate with the Controller.

Element in controller-info.xml: <account-name>

System Properties: -Dappdynamics.agent.accountName

Type: String

Default: None

Required: Yes for AppDynamics SaaS Controller and other multi-tenant users; no for single-tenant users.

Account Access Key Property

Description: This is the account access key used to authenticate with the Controller.

Element in controller-info.xml: <account-access-key>

System Properties: -Dappdynamics.agent.accountAccessKey

Type: String

Default: None

Required: Yes for AppDynamics SaaS Controller and other multi-tenant users; no for single-tenant users.

Proxy Properties for the Controller

These properties route data to the Controller through a proxy.

Proxy Host Property

Description: This is the proxy host name or IP address.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.http.proxyHost

Type: String

Default: None

Required: No

Proxy Port Property

Description: This is the proxy HTTP(S) port.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.http.proxyPort

Type: Positive Integer

Default: None

Required: No

Other Properties

Controller SSL Enabled Property

Description: When set to true, this property specifies that the agent should use SSL (HTTPS) to connect to the Controller. If SSL Enabled is true, set the Controller Port property to the HTTPs port of the Controller. See Controller Port Property.

Element in controller-info.xml: <controller-ssl-enabled>

System Property: -Dappdynamics.controller.ssl.enabled

Type: Boolean

Default: False

Required: No

Enable Orchestration Property

Description: When set to true, enables auto-detection of the controller host and port when the app server is a compute cloud instance created by an AppDynamics orchestration workflow. See Controller Host Property and Controller Port Property.

In a cloud compute environment, auto-detection is necessary for the Create Machine tasks in the workflow to run correctly.

If the host machine on which this agent resides is not created through AppDynamics workflow orchestration, this property should be set to false.

Element in controller-info.xml: <enable-orchestration>

System Property: Not applicable

Type: Boolean

Default: False

Required: No

Agent Runtime Directory Property

Description: This property sets the runtime directory for all runtime files (logs, transaction configuration) for nodes that use this agent installation. If this property is specified, all agent logs are written to <Agent-Runtime-Directory>/logs/node-name and transaction configuration is written to the <Agent-Runtime-Directory>/conf/node-name directory.

Element in controller-info.xml: <agent-runtime-dir>

System Property: -Dappdynamics.agent.runtime.dir

Type: String

Default: <Agent_Installation_Directory>/nodes

Required: No

Redirect Logfiles Property

Description: This property sets the destination directory to which to redirect log files for a node.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.agent.logs.dir

Type: String

Default: <Agent_Installation_Directory>/logs/<Node_Name>

Required: No

Force Agent Registration Property

Description: Set to true only under the following conditions:

- The Agent has been moved to a new application and/or tier from the UI
  and
- You want to override that move by specifying a new application name and/or tier name in the agent configuration.

Element in controller-info.xml: <force-agent-registration>

System Property: Not applicable

Type: Boolean

Default: False

Required: No

Reuse Node Name Property

Description: Set this property if you want the Controller to generate unique node names automatically using a prefix.

You can specify the prefix in the Auto Node Prefix Property. If you do not provide a prefix but set the reuse.nodeName property to true, the Controller uses the tier name as a prefix.

This property is useful for dynamic multi-tier clustered applications with many JVMs that have short life spans. It allows AppDynamics to reuse node names and to capture historical data for these short-lived nodes after they become historical or are deleted.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.agent.reuse.nodeName

Type: Boolean

Default: None

Required: No

Auto Node Name Prefix Property

Description: Set this property if you want the Controller to generate node names automatically using a prefix that you provide.

The Controller generates node names based on the prefix concatenated with a number, which is incremented sequentially. For example, if you assign a value of "mynode" to this property, the Controller generates node names "mynode-1", "mynode-2" and so on.

If one of the nodes is deleted and the Reuse Node Name Property is true, the Controller will re-use the deleted node name.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.agent.auto.node.prefix=<your_prefix>

Type: String

Default: Serial number maintained by the Controller appended to the tier name

Required: No

Chron/Batch JVM Property

Description: Set this property to true if the JVM is a batch/chron process or if you are instrumenting the main() method.

Element in controller-info.xml: Not applicable

System Property: -Dappdynamics.cron.vm

Type: Boolean

Default: False

Required: No

Unique Host ID Property

Description: This property logically partitions a single physical host or virtual machine.

You can use the unique host id when you want to use the same node name for multiple nodes on the same physical machine.

Set the value to a string that is unique across the entire managed infrastructure. The string may not contain any spaces.

If this property is set on the app agent, it must be set on the machine agent as well.

System Property: -Dappdynamics.agent.uniqueHostId

Type: String

Default: None

Required: No

Learn More

- Name Business Applications, Tiers, and Nodes
- Configure App Agent for Java for JVMs that are Dynamically Identified
- Configure App Agent for Java to Use Existing System Properties
- Java Agent on z-OS or Mainframe Environments Configuration

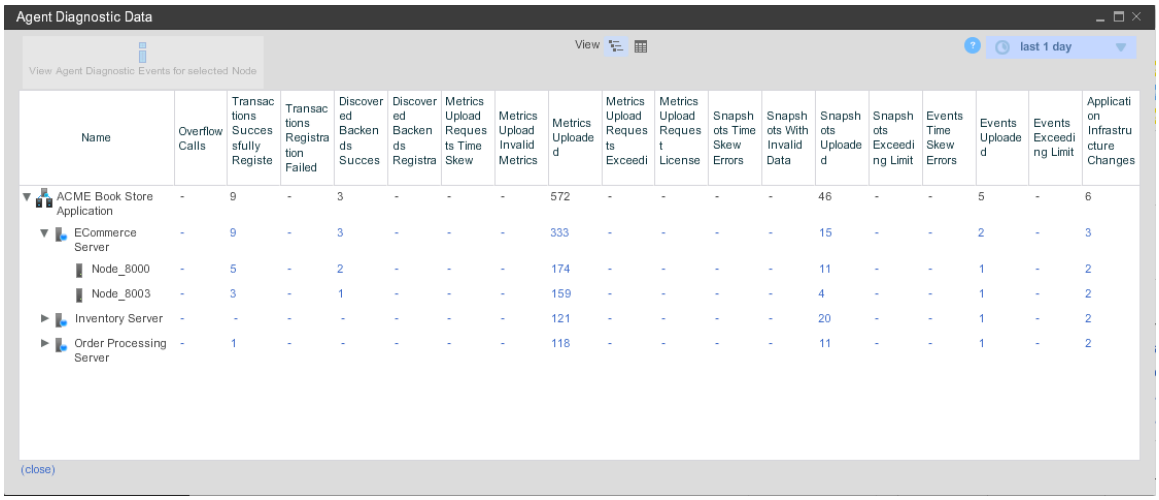
App Agent for Java Diagnostic Data

- To view agent diagnostic data
- To view agent diagnostic stats
- Learn More

To view agent diagnostic data

1. From an Application Dashboard, click **Actions -> View Agent Diagnostics**.

The Agent Diagnostic Data window opens and displays various metrics related to the application, tiers, and nodes.



Name	Overflow Calls	Transactions Successfully Registe	Transactions Registration Failed	Discovered Backends Succes	Discovered Backends Registra	Metrics Upload Requests Time Skew	Metrics Upload Invalid Metrics	Metrics Uploaded	Metrics Upload Requests Exceedi	Metrics Upload Request License	Snapshots Time Skew Errors	Snapshots With Invalid Data	Snapshots Uploaded	Snapshots Exceeding Limit	Events Time Skew Errors	Events Uploaded	Events Exceeding Limit	Application Infrastructure Changes
ACME Book Store Application	-	9	-	3	-	-	-	572	-	-	-	-	46	-	-	5	-	6
ECommerce Server	-	9	-	3	-	-	-	333	-	-	-	-	15	-	-	2	-	3
Node_8000	-	5	-	2	-	-	-	174	-	-	-	-	11	-	-	1	-	2
Node_8003	-	3	-	1	-	-	-	159	-	-	-	-	4	-	-	1	-	2
Inventory Server	-	-	-	-	-	-	-	121	-	-	-	-	20	-	-	1	-	2
Order Processing Server	-	1	-	-	-	-	-	118	-	-	-	-	11	-	-	1	-	2

2. Select a node and click **View Agent Diagnostic Events for selected Node**.

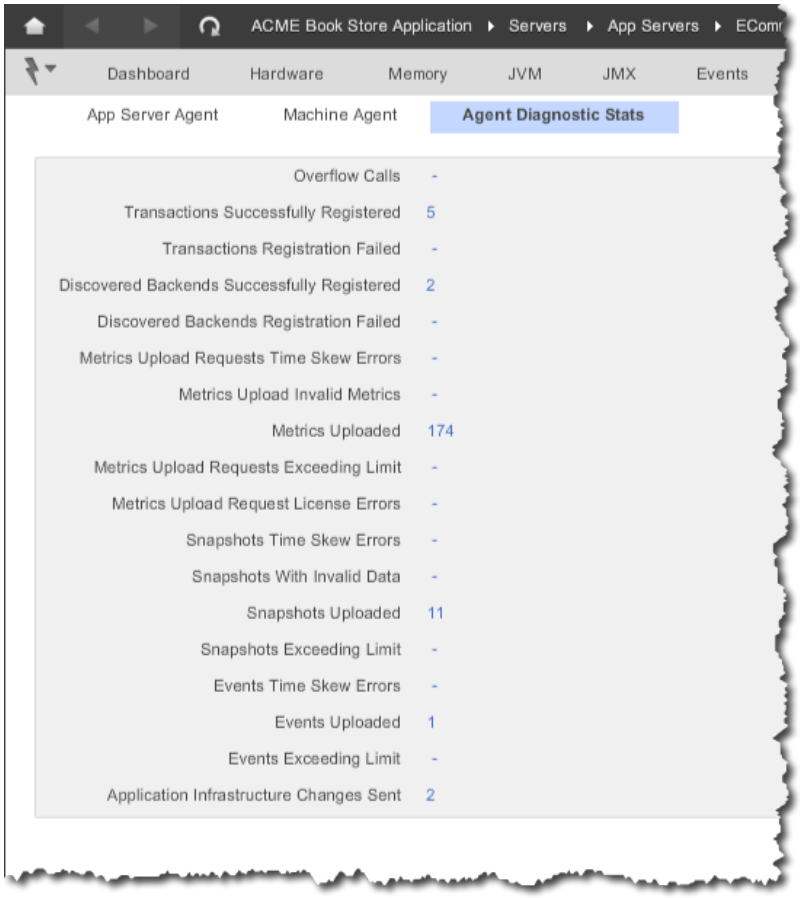
The Agent Diagnostics Event window opens.

To view agent diagnostic stats

1. From the left navigation pane, click *Servers -> App Servers -> <Tier> -> <Node>.

2. In the Node Dashboard, click the Agents tab.

3. Click the Agent Diagnostic Stats subtab.



Learn More

- Troubleshoot Node Problems

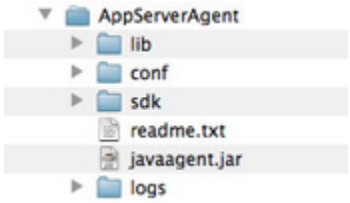
App Agent for Java Directory Structure

- App Agent for Java Directory Structure
- The conf Directory

App Agent for Java Directory Structure

The AppServerAgent.zip folder contains files for installing the App Agent for Java.

The directory structure for the agent is illustrated in the following screen shot.

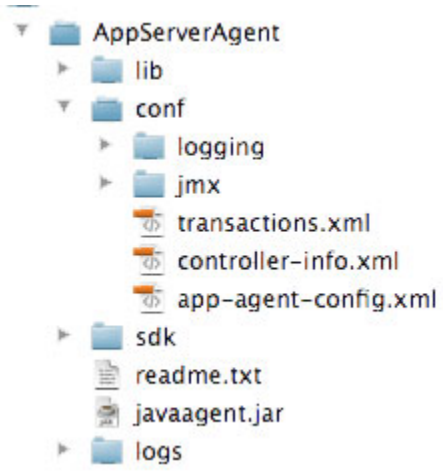


- **lib:** The lib folder has the core agent libraries and the third-party libraries.
- **conf:** The conf folder is the configuration directory that has local configuration backup.
- **sdk:** This folder contains the Javadocs and APIs to extend AppDynamics' monitoring capabilities.
- **javaagent.jar:** This JAR file has the argument to bootstrap the App Agent for Java. To enable the agent, the --javaagent

argument for JVM startup is required. See Install the App Agent for Java.
- **logs:** All logs written by the agent in this directory.

The conf Directory

The files located in this directory are commonly used for agent configuration and deployment.



- **transactions.xml:** This XML file has configurations for business transaction identified by the agent.
- **controller-info.xml:** This XML file has controller connectivity and identification.

The following agent settings are configured using this XML file:

- Agent communication settings (specified using <controller-host> and <controller-port> properties).
- Agent identification settings (specified using <application-name>, <tier-name>, and <node-name> properties).
- **app-agent-config.xml:** This XML file contains agent configuration. Any changes to the agent's local settings are specified in this file, and these changes override the global configuration. This file is typically used for short-term properties settings or for debugging agent issues.
- **jmx:** The jmx folder contains files for configuring the JMX and Websphere PMI metrics.
- **logging/log4j.xml:** This file contains flags to control logging levels for the agent. It is highly recommended not to change the default logging levels.

To specify a different log directory, use the following system property:

```
-Dappdynamics.agent.logs.dir
```

App Agent for Java Performance Tuning

- Business Transaction Thresholds
- Snapshot Collection Thresholds
  - Disable Scheduled Snapshots
  - Suggested Scheduling Settings
  - Suggested Diagnostic Session Settings
- Tuning Call Graph Settings
  - Suggested SQL Query Time and Parameters
- Memory Monitoring
- Learn More

This topic discusses how to get the best performance from App Agents for Java.

Business Transaction Thresholds

AppDynamics determines whether transactions are slow, very slow, or stalled based on the thresholds for Business Transactions. AppDynamics recommends using standard deviation based dynamic thresholds. See Configure Thresholds.

Snapshot Collection Thresholds

Snapshot collection thresholds determine when snapshots are collected for a Business Transaction. Too many snapshots can affect performance and therefore snapshot collection thresholds should be considered carefully in production or load testing scenarios. See Configure Thresholds.

Disable Scheduled Snapshots

For more information see Transaction Snapshots.

Suggested Scheduling Settings

- 10 minutes for small deployments < 10 BTs
- 20 minutes for medium deployments < 10 - 50 BTs
- 30 minutes for > 50 - 100 BTs
- 60 minutes > 100 BTs

Suggested Diagnostic Session Settings

- Settings for Slow requests (%value): 20 - 30
- Settings for Error requests (%value): 10 - 20
- Click **Apply to all Business Transactions**.

Tuning Call Graph Settings

To configure call graph settings, click **Configure -> Instrumentation** and the **Call Graph Settings** tab. See Configure Call Graphs.

Suggested SQL Query Time and Parameters

- Minimum SQL Query Time : 100 ms (Default is 10)
- Enable **Filter SQL Parameter Values**.

Memory Monitoring

Memory monitoring features such as leak detection, object instance tracking, and custom memory should be enabled only for a specific node or nodes when debugging a memory problem. Automatic leak detection is on-demand, and therefore, the leak detection will execute only for the specified duration.

When you observe periods of growth in the heap utilization (%), you should enable on-demand memory leak capture. See Troubleshoot Java Memory Leaks.

Learn More

- Configure Thresholds
- Configure Call Graphs
- Troubleshoot Java Memory Leaks

Configure App Agent for Java for Batch Processes

- To configure the App Agent for Java
- To use the script name as the node name
- Learn More

You can configure the App Agent for Java for those JVMs that run as cron or batch jobs where the JVM runs only for the duration of the job. AppDynamics monitors the main method of the Java program.

To configure the App Agent for Java

1. Add the application and tier name to the controller-info.xml file.

2. Add the appdynamics.cron.vm property to the AppDynamics javaagent command in the startup script of your JVM process:

```
-javaagent:<agent_install_dir>/javaagent.jar
-Dappdynamics.agent.nodeName=${NODE_NAME} -Dappdynamics.cron.vm=true
```

The agent_install_dir is the full path of the App Agent for Java installation directory.

The appdynamics.cron.vm property creates a delay between the end of the main method and the JMV exit so that the Agent has time to upload metrics to the Controller.

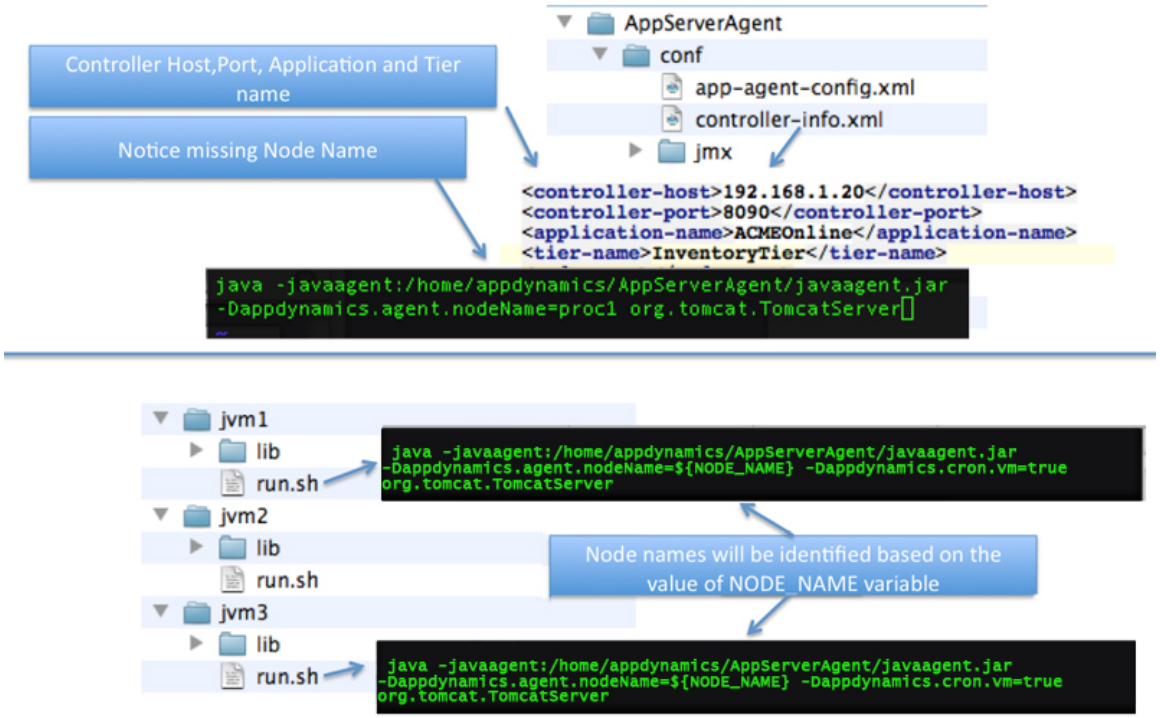
To use the script name as the node name

You can use the script name that executes the cron or batch job as the node name.

The following commands set the value of variable NODE_NAME using the combination of the script and host name. Add these commands to the startup script of the JVM.

```
# Use the name of the script (no path, no extension) as the name of the node.
NODE_NAME=sample
NODE_NAME="${NODE_NAME%%.}"
echo $NODE_NAME
# Localize the script to the host.
NODE_NAME="$NODE_NAME@$HOSTNAME"
```

The following illustration shows the sample configuration for controller-info.xml and the startup script of the JVM.



Learn More

- Configure Background Tasks (Java)

Configure App Agent for Java for JVMs that are Dynamically Identified

- To configure the node name of the App Agent for Java
- Configuration notes

This topic describes how to configure the App Agent for Java in environments where the JVMs are dynamic.

To configure the node name of the App Agent for Java

1. Add the **application** and **tier name** to the controller-info.xml file.

2. Add the javaagent argument and the following system properties (-D options) to the startup script of the JVMs:

```
java -javaagent:<agent-install-dir>/javaagent.jar -Dappdynamics.agent.nodeName=${NODE_NAME}
```

Configuration notes

The system properties are separated by a white space character.

The <agent-install-dir> references the full path of the App Agent for Java installation directory.

The token ${NODE_NAME} identifies the JVMs dynamically and names these JVMs based on the parameter value passed during the execution of the startup script for your JVM process.

The application and tier names can also be configured using the system properties. For more details see App Agent for Java

Configuration Properties.

Some application server management consoles allow you to specify startup arguments using a web interface.
For details see Java Server-Specific Installation Settings.

Configure App Agent for Java in Restricted Environments

- To write the "startup hook" agent program

Some restricted environments do not allow any changes to the JVM startup script. For these environments AppDynamics provides the appdynamics.agent.startup.hook property. This "startup hook" allows a single point of deployment for the agent. You create a Java main method that is invoked programmatically, before your startup script is executed.

To write the "startup hook" agent program

1. Implement a class with Java main method.

2. Create a JAR file for this class.

3. In the manifest of the JAR file, specify the class created in step 1.

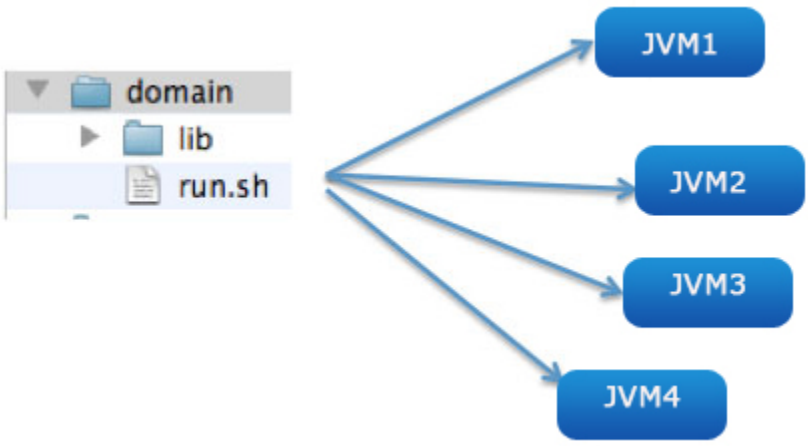
4. Add the following javaagent argument and system properties (-D options) to your startup script:

```
-javaagent:<agent_install_dir>/javaagent.jar
-Dappdynamics.agent.startup.hook=<JAR-file>
```

Configure App Agent for Java in z-OS or Mainframe Environments

- To name nodes automatically
- To remove dead nodes
- Learn More

In some environments JVMs have transient identity, such as when a single script spawns multiple JVMs.



For example, an environment may consist of WebSphere on IBM Mainframes, using a dynamic workload management feature that spawns new JVMs for an existing application server (called a servant). These JVMs are exact clones of an existing JVM, but each of them has a different process ID. Based on load, any number of additional JVMs may be created.

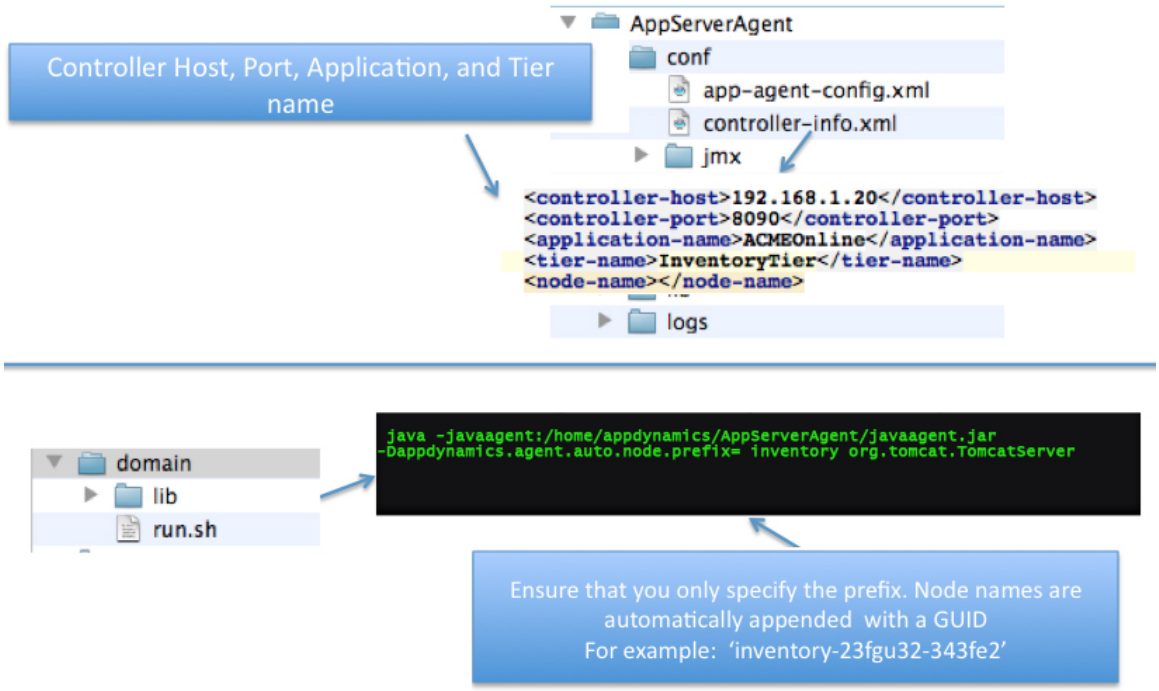
To name nodes automatically

The App Server Agent can automatically name the dynamically generated JVMs using the appdynamics.agent.auto.node.prefix property. See Reuse Node Name Property to enable re-use of node names and Auto Node Name Prefix Property to set the prefix used for automatically-named nodes. You used these properties in your startup script.

```
-Dappdynamics.agent.auto.node.prefix=<node name prefix>
-Dappdynamics.agent.agent.reuse.nodeName=true
```

If you are using these properties, ensure that you have not specified the node name anywhere (controller-info.xml file or as a system property) in your JVMs start-up script.

The following illustration shows the sample configuration for ACME Bookstore. This configuration will create unique node names for every instance of the virtual machine starting up in the ACME Bookstore environment.

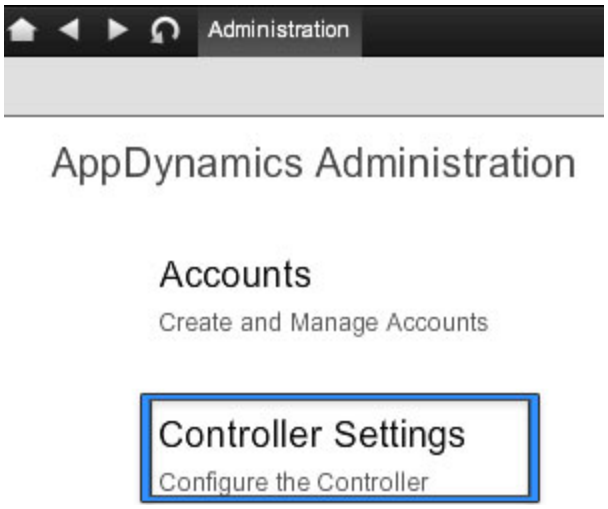


To remove dead nodes

In a typical environment, the nodes are recycled and new nodes get generated. AppDynamics strongly recommends that you remove the dead nodes both from the AppDynamics user interface (UI) as well as from the system.

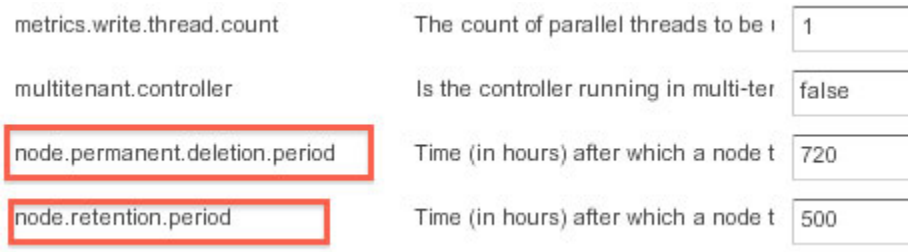
1. Log in to the Controller administration console.

2. Go to "Controller Settings" section to see the advanced properties for the Controller.



3. Set the retention and deletion properties, based on the requirements for your environment. AppDynamics recommends that you set the permanent deletion period at least an hour more than the retention period.

- **node.permanent.deletion.period:** Time (in hours) after which a node that has lost contact with the Controller is deleted permanently from the system.
- **node.retention.period:** Time (in hours) after which a node that has lost contact with the Controller is deleted. In this case, the AppDynamics UI will not display the node, however the system will continue to retain it.



Learn More

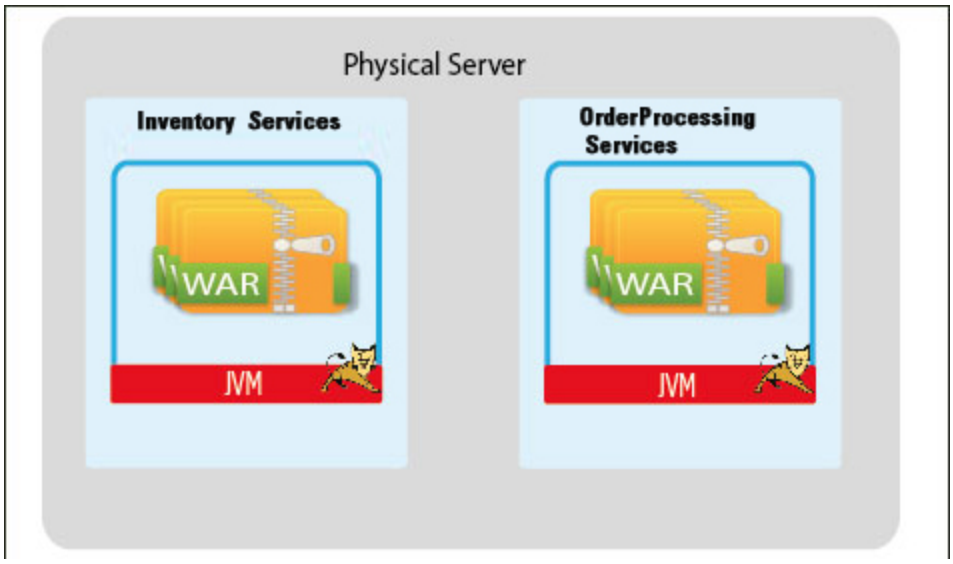
- App Agent for Java Configuration Properties

Configure App Agent for Java on Multiple JVMs on the Same Machine that Serve Different Tiers

- To configure the App Agent for Java

This section describes how you can configure the App Agent for Java for multiple JVMs that are located on a single machine and are serving two tiers.

For example, the ACME Bookstore has two JVMs on the same physical server. These two JVMs are bound to two different virtual IP (one JVM is used for Order Processing Services and the other JVM is used for Inventory Services).



For such cases, follow these rules:

- All of the common information should be configured using controller-info.xml file.
- All of the information unique to a JVM should be configured using the system properties in the JVM startup script.
- Information in the startup scripts always overrides the information in the controller-info.xml file.

To configure the App Agent for Java

1. Add **application name** to **controller-info.xml** file.

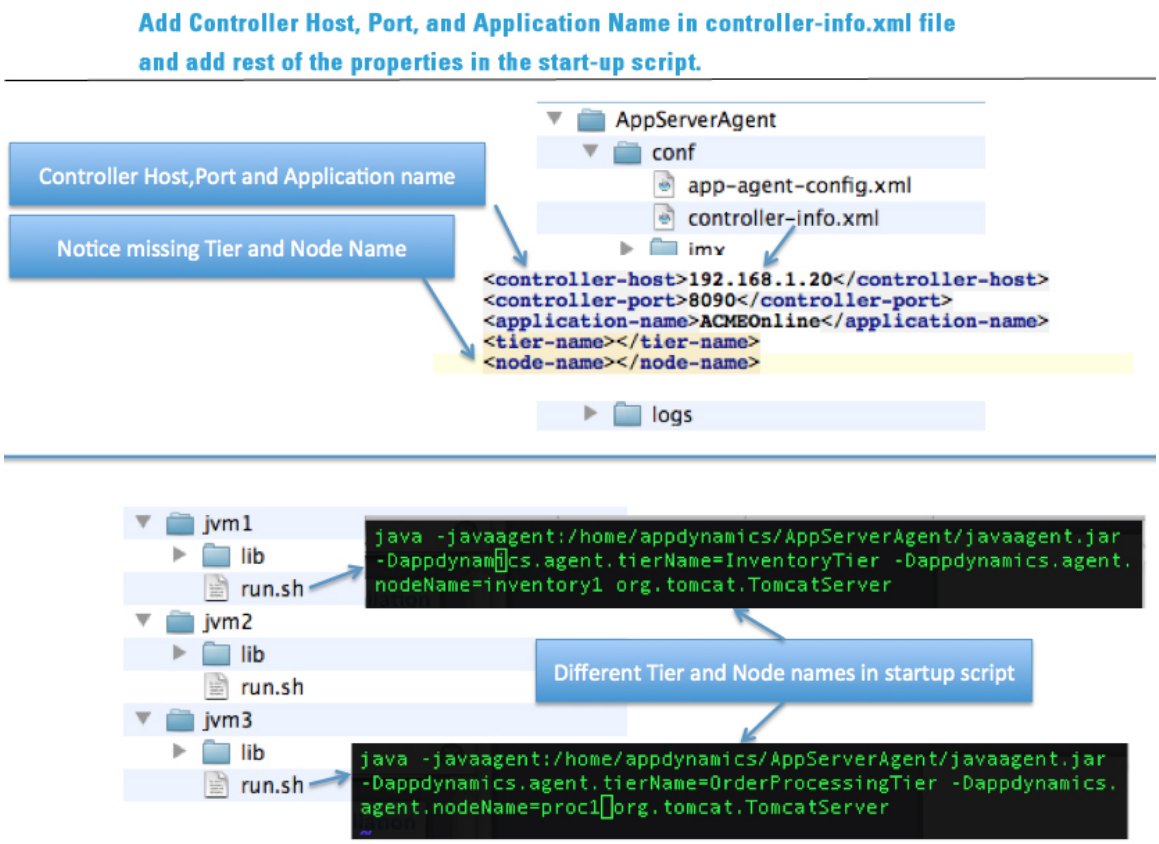
2. Add **javaagent** argument and following system properties (-D options) to the start-up script to each of your JVM:

```
java -javaagent:<Agent-Installation-Directory>/javaagent.jar
-Dappdynamics.agent.tierName=$tierName -Dappdynamics.agent.nodeName=$nodeName
```

Separate the system properties with a white space character.

All agents in shared mode need a unique node name so that they can be differentiated from one another. See Configure App Agent for Java to Use Existing System Properties.

The following illustration displays how this configuration is applied to ACME Bookstore:



Tips:

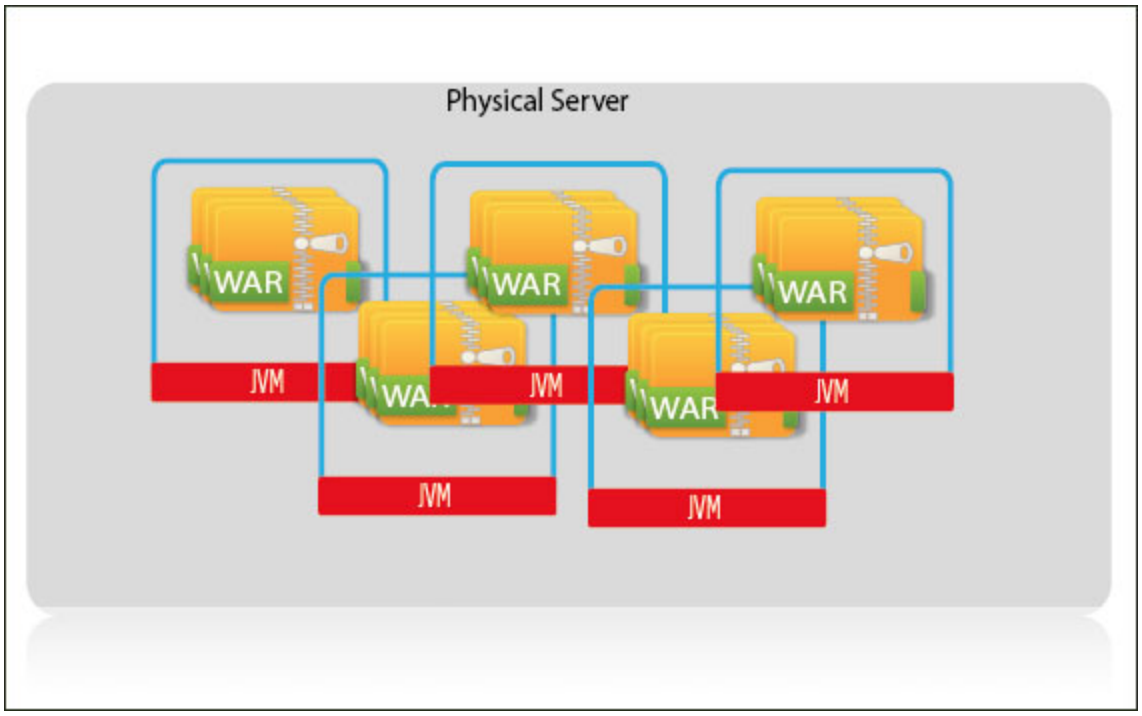
- The application and tier names can also be configured using the system properties. For more details see App Agent for Java Configuration Properties.
- Some application server management consoles allow you to specify startup arguments using a web interface. See Java Server-Specific Installation Settings.

Configure App Agent for Java on Multiple JVMs on the Same Machine that Serves the Same Tier

- To configure the App Agent for Java properties

This topic describes how to configure the App Agent for Java for multiple JVMs that are located on a single machine and are serving the same tier.

For example, ACME Bookstore has a physical server with five JVMs installed on it. All of these JVMs are used for the Inventory Services.



For such cases, follow these rules:

- All of the common information should be configured using controller-info.xml.
- All of the information unique to a JVM should be configured using the system properties in the start-up script.
- Information in the startup scripts always overrides the information in the controller-info.xml file.

To configure the App Agent for Java properties

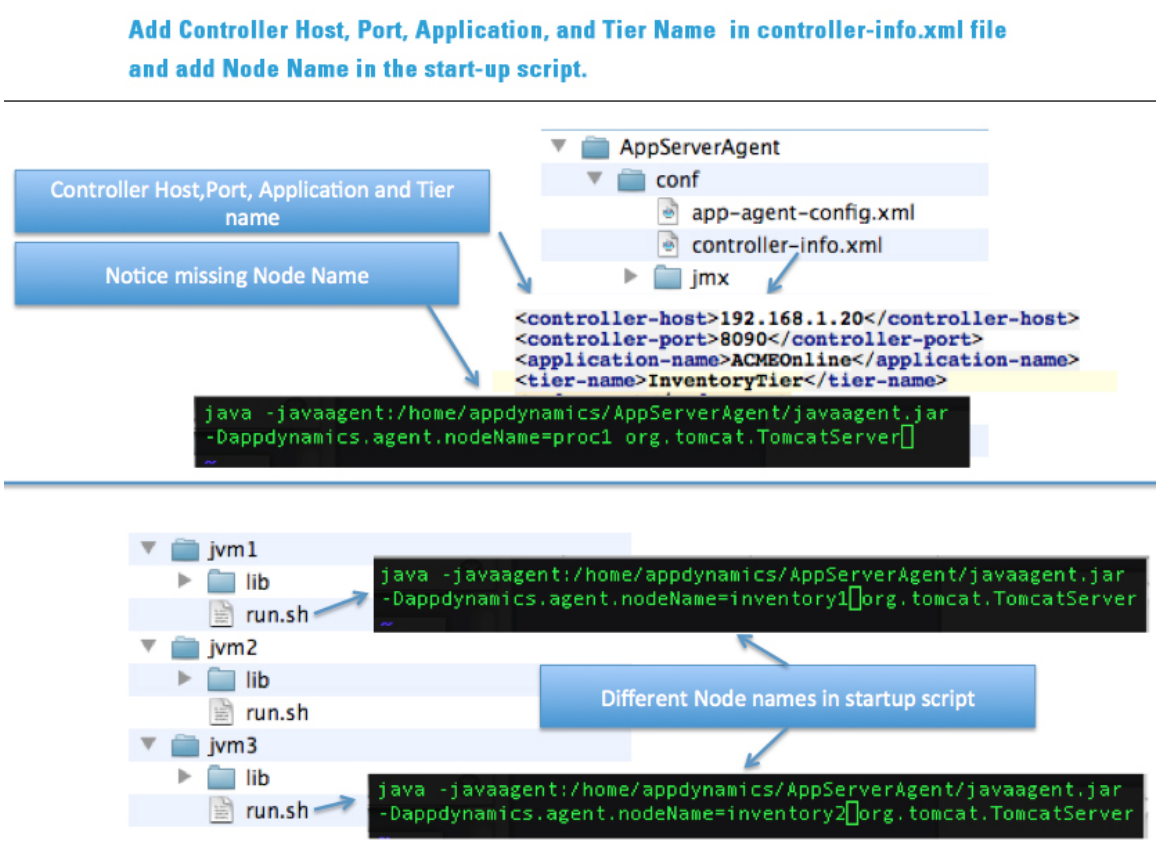
1. Provide the application and tier name in the controller-info.xml file.

2. Add the javaagent argument and system property (-D option) for the node name to the batch file or startup script of each JVM.

```
java -javaagent:<Agent-Installation-Directory>/javaagent.jar
-Dappdynamics.agent.nodeName=$nodeName
```

Separate the system properties with a white space character.

The following illustration displays how this configuration is applied to the ACME Bookstore.



The application and tier names can also be configured using the system properties. See App Agent for Java Configuration Properties.

Some application server management consoles allow you to specify start-up arguments using a web interface. See Java Server-Specific Installation Settings.

Configure App Agent for Java to Use Existing System Properties

- System Properties
  - Using System Properties
    - To identify nodes
    - To identify tiers
  - Sample Agent Configuration Using System Properties
- Learn More

This topic explains how to configure the App Agent for Java using the existing system property values.

System Properties

AppDynamics recommends that you use the existing system properties to configure the Agent when your environment consists of multiple JVMs on the same machine. Once you have these variables configured, you can complete Agent installation for all JVMs by simply adding the javaagent argument to each JVM startup script. Then add the rest of the information to the controller-info.xml file.

AppDynamics recommends that you use the system properties if the same startup script is starting all the JVMs in your environment.

You can identify the node name based on the value of -Dserver.name and the tier name based on the value of -Dcluster.name.

Also, you can combine two or more system properties to identify the node or tier name. You can use -Dhost.name and -Dserver.name to identify similarly named nodes on different machines even when they belong to the same tier.

Using System Properties

Use the following syntax to represent the value of the system property in the controller-info.xml file.

```
${system property name}
```

You can combine multiple system properties.

```
${host.name}${server.name}
```

You can combine system properties with literals. In the following example '_' and 'inventory' are literals.

```
${host.name}_${server.name}.inventory
```

To identify nodes

```
${host.name}
```

or

```
${server.name}
```

or

```
${host.name}${server.name}
```

To identify tiers

```
${cluster.name}
```

Sample Agent Configuration Using System Properties

Consider a JVM with a script file named startserver.sh. This script file has following system property:

```
-Dserver.name=$1
```

If you execute:


```
startserver.sh ecommerce01
```

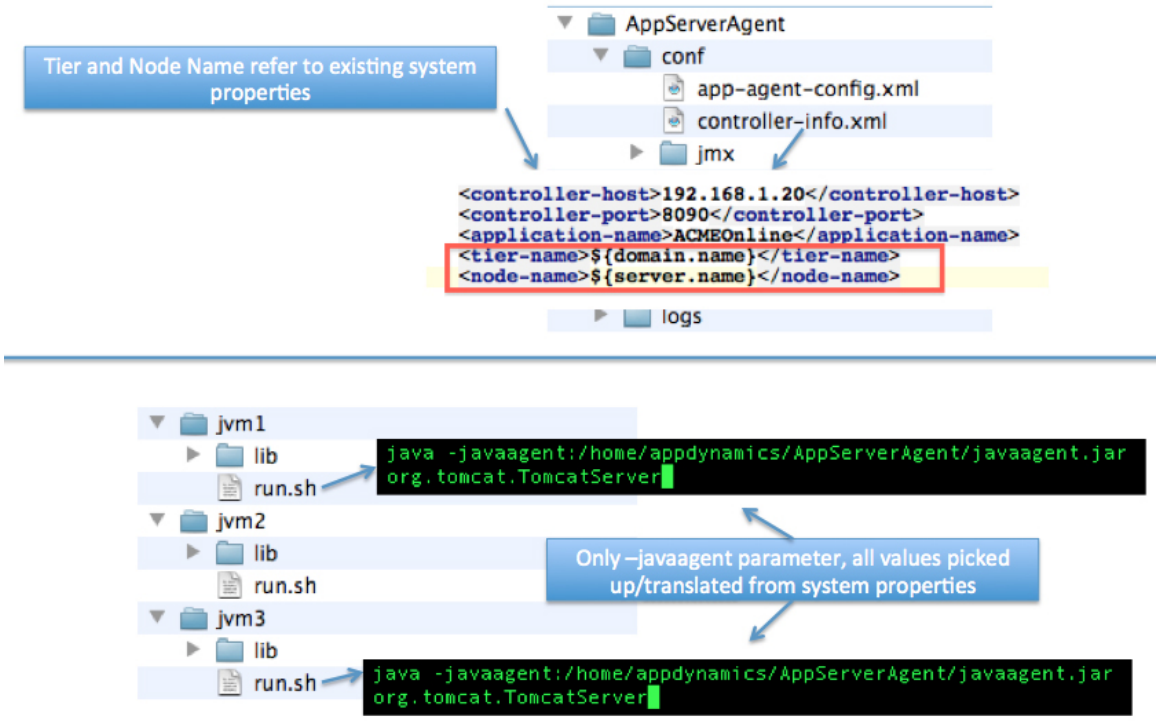
The script creates a new server named ecommerce01.

To use this system property for Agent configuration, add the appdynamics.nodeName property to startserver.sh file.

```
-Dappdynamics.nodeName=$server.name
```

When the script creates the new server named ecommerce01, it will identified by AppDynamics as both a server and as a node.

The following screenshot shows a sample configuration for the controller-info.xml file and the startup script.



Learn More

- Logical Model
- Install the App Agent for Java

IBM App Agent for Java

- Supported JVMs
- Instrumenting the IBM App Agent for Java

Under most circumstances, the IBM App Agent for Java works the same as the App Agent for Java.
This topic gathers information specific to the IBM App Agent for Java.

Supported JVMs

IBM JVM 1.5.x, 1.6.x

Instrumenting the IBM App Agent for Java

To change instrumentation for the IBM App Agent for Java, the IBM JVM must be restarted. By default the IBM App Agent for Java does not apply BCI changes without restarting the JVM. This is because in the IBM VM (J9 1.6.0) the implementation of re-transformation affects performance (changes the JIT behavior such that less optimization occurs).

The following changes require that you restart the IBM JVM.

- Automatic leak detection
- Custom memory structures
- Information points
- Aggressive snapshot policy (also called hotspot instrumentation)
- Custom POJO rules for transaction detection
- Custom exit point rules
- End user experience monitor (EUM), when you enable it and/or disable it after first enabling it

Move an App Agent for Java Node to a New Application or Tier

- Moving a Node to a New Application or Tier
  - To change the Java Agent associations from the UI
    - Optionally update the controller-info.xml file
    - Forcing node re-registration using the controller-info.xml file
  - Learn More

If your JVM machine has both an App Agent for Java and a Machine Agent, you cannot change the associations in the Machine Agent controller-info.xml file. You can only change these associations either through the UI or by modifying the App Agent for Java controller-info.xml file.

Moving a Node to a New Application or Tier

You can assign an App Agent for Java node to a new application or tier using the AppDynamics UI without restarting the JVM.

You can optionally update the controller-info.xml file and restart the JVM. You must restart the JVM when you change the associations in the controller-info.xml file.

Moving the node using the AppDynamics UI cannot be overridden by an agent configuration unless you set the force-agent-registration flag to true in the controller-info.xml file.

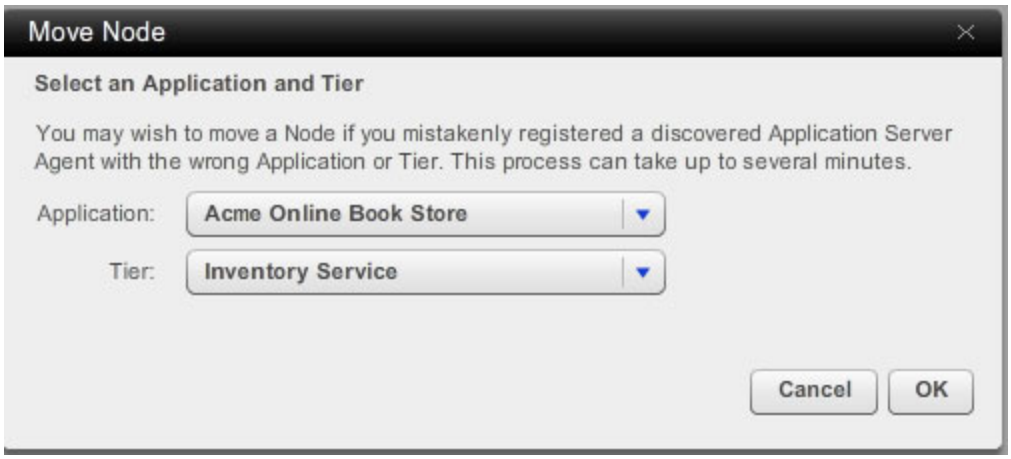
To change the Java Agent associations from the UI

1. Select the node that you want to move by clicking **Servers -> App Servers -> <Tier> -><Node>**.

2. Click **Actions -> Move Node**.

4. In the Move Node window select the new application and/or tier from the drop-down menus.

5. Click **OK** to confirm the reassignment. It may take several minutes to complete.



When you change the associations for an App Agent for Java, AppDynamics registers an Application Changes event. You can review the details of the event in the Events view.

Optionally update the controller-info.xml file

The UI does not update the controller-info.xml file. However if you restart the JVM the Controller will remember and keep the change you made from the UI.

You may want to maintain consistency with the controller-info.xml file, just for neatness' sake. Perform these two additional steps:

6. Update these configuration changes in the App Agent for Java controller-info.xml file.

7. Restart the JVM.

If it is not feasible to restart the JVM at this time, remember the change and update the file the next time you restart the JVM.

Forcing node re-registration using the controller-info.xml file

If you've moved a node in the UI and you want to move it again elsewhere using controller-info.xml, then when you restart the JVM you set the force-agent-registration property to 'true'. See App Agent for Java Configuration Properties#Force Agent Registration Property.

Learn More

- Logical Model

Troubleshoot App Agent for Java

- Troubleshooting App Agent for Java Startup
  - Locating the App Agent for Java Log Files
  - Troubleshooting Incomplete Agent Configuration
  - Troubleshooting the Controller Port
  - Troubleshooting Incorrect File Permissions
- Learn More

This topic discusses techniques for troubleshooting and interpreting the information in the App Agent for Java log files.

Troubleshooting App Agent for Java Startup

After sending a request to your web application, data should appear in the UI. If no data appears, check the following:

1. You have re-started the application server.
2. Verify that the javaagent argument has been added to the startup script of your JVM.
3. Verify that you configured the agent-controller communication properties and agent identification properties in the controller-info.xml file or as system properties in the startup script of your JVM. See App Agent for Java Configuration Properties.

4. Check the Agent logs directory located at <Agent_Installation_Directory>/logs/<Node_Name> for the agent.log file.
5. Verify that the Agent is compatible with the Controller. For details see Agent - Controller Compatibility Matrix.

Locating the App Agent for Java Log Files

Agent log files are located in the <Agent_Installation_Directory>/logs/<Node_Name> folder.

The agent.log file is the recommended file to help you with troubleshooting. This log can indicate the following:

- Incomplete information in your Agent configuration.
- Indicates if the Controller port is blocked.
- Incorrect permissions.

Error messages related to starting the App Agent for Java use this format:

```
ERROR com.singularity.JavaAgent - Could Not Start Java Agent
```

Troubleshooting Incomplete Agent Configuration

The following table lists the typical error messages for incomplete Agent configuration:

Error Message	Solution
Cannot connect to the Agent - ERROR com.singularity.XMLConfigManager - Incomplete Agent Identity data, Invalid Controller Port Value []	This indicates that the value for the controller port in controller-info.xml is missing. Add the port value, along with the host value (<your-host-name>), to fix this error.  • **For on-premise Controller installations:** Default port value is 8090 for HTTP and 8181 for HTTPS. • **For Controller SaaS service:** Default port value is 80 for HTTP and 443 for HTTPS.
Caused by: com.singularity.ee.agent.configuration.a: Could not resolve agent-controller basic configuration	This is usually caused because of incorrect configuration in the Controller-info.xml file. Ensure that the information for agent communication (Controller host and port) and agent identification (application, tier and node names) is correctly configured. Alternatively, you can also use the system properties (-D options) or environment variables to configure these settings.

For more information about agent properties see App Agent for Java Configuration Properties.

Troubleshooting the Controller Port

The following table lists the typical error message when the Controller port is blocked in your network:

Error Message	Solution
ERROR com.singularity.CONFIG.ConfigurationChannel - Fatal transport error: Connection refused WARN com.singularity.CONFIG.ConfigurationChannel - Could not connect to the controller/invalid response from controller, cannot get initialization information, controller host \x.x.x.x\, port 8090, exception Fatal transport error: Connection refused	Try to ping <your-host-name> from the machine where you have configured the Application Server Agent. If it works, then confirm if the Controller port is not blocked in your network.  To check if a port was blocked in the network; use command: **netstat -an** for Windows and nmap for Linux.  * **For on-premise Controller installations:** Default port value is 8090 for HTTP and 8181 for HTTPS. • **For Controller SaaS service:** Default port value is 80 for HTTP and 443 for HTTPS.

Troubleshooting Incorrect File Permissions

Following table lists the typical error message when the file permissions are not correct:

Error Message	Solution
ERROR com.singularity.JavaAgent - Could Not Start Java Agent com.singularity.ee.agent.appagent.kernel.spi.c: Could not start services"	This is usually caused because of incorrect permissions for log files. Confirm if the user who is running the server, has read and write permission on the agent directories. If the user has **chmod a-r** equivalent permission, change the permission to **chmod a+r** "<agent_directory>"

Learn More

- Install the App Agent for Java
- App Agent for Java Configuration Properties

Administer App Agent for Java FAQ

- App Agent for Java Administration FAQ
  - Q. Why am I seeing "WARN AsyncHandOffIdentificationInterceptor - Reached maximum limit 500 of async handoff call graph samples. No more samples will be taken" message in the agent log files?

App Agent for Java Administration FAQ

Q. Why am I seeing "WARN AsyncHandOffIdentificationInterceptor - Reached maximum limit 500 of async handoff call graph samples. No more samples will be taken" message in the agent log files?

In AppDynamics 3.6 and 3.7, all Runnables, Callables and Threads are instrumented by default except for the ones that are excluded by the agent configuration in app-agent-config.xml. In some environments, this could result in too many classes being instrumented, or cause common classes in a framework that implements the Runnable interface to be mistaken for asynchronous activity when it is not, for example Groovy application using Closures.
To debug the cause of the message, check the call graph to see if so many asynchronous activities are legitimate. If they are not, then exclude the packages that are not really asynchronous activities. See Configure Multi-Threaded Transactions (Java).

Configure AppDynamics for Java

Configure Custom Exit Points (Java)

- Default Backends Discovered by the App Agent for Java
- Configure Custom Exit Points for Java Backends
- To create a custom exit point
  - To split an exit point
  - To group an exit point
- To define custom metrics for a custom exit point
- To define transaction snapshot data collected
- Learn More

AppDynamics provides default automatic discovery for commonly-used backends. If a backend used in your environment is not discovered, first compare the list of default backends to determine whether you need to modify the default configuration. If it is not on

the list then configure a custom exit point according to these instructions.

Default Backends Discovered by the App Agent for Java

The default backends for Java are:

- HTTP
- JDBC
- JMS
- MQ
- RMI
- Thrift (not currently configurable)
- Web Service

To configure a default backend see Configure Backend Detection.

Configure Custom Exit Points for Java Backends

Configure Custom Exit Points

Custom exit points provide identification for backend types that are not automatically detected, such as file systems, mainframes etc. For example, you can define a custom exit call to monitor the file system read method. Custom exit points appear as unresolved backends in the flow maps. Unresolved backends are shown on flow maps with this icon  .

You define a custom exit point by specifying the class and method used to identify the backend. If the method is overloaded, you need to add the parameters to identify the method uniquely.

You can restrict the method invocations for which you want AppDynamics to collect metrics by specifying match conditions for the method. The match conditions can be based on a parameter or the invoked object.

You can also optionally split the exit point based on a method parameter, the return value, or the invoked object.

You can also configure custom metrics and transaction snapshot data to collect for the backend.

See Configurations for Custom Exit Points for suggested custom configurations for some common backends.

To create a custom exit point

1. In the left navigation panel, click **Configure -> Instrumentation**.

2. Click the **Backend Detection** tab.

3. Click the tab corresponding to the backend platform.

4. Select the application or tier for which you are configuring the custom exit point. Backend detection configuration is applied on a hierarchical inheritance model. See Hierarchical Configuration Model.

5. Scroll down to Custom Exit Points.

6. Click **Add** (the + icon).

7. In the Create Custom Exit Point window, click the **Identification** tab if it is not selected.

8. Enter a name for the exit point. This is the name that identifies the backend.

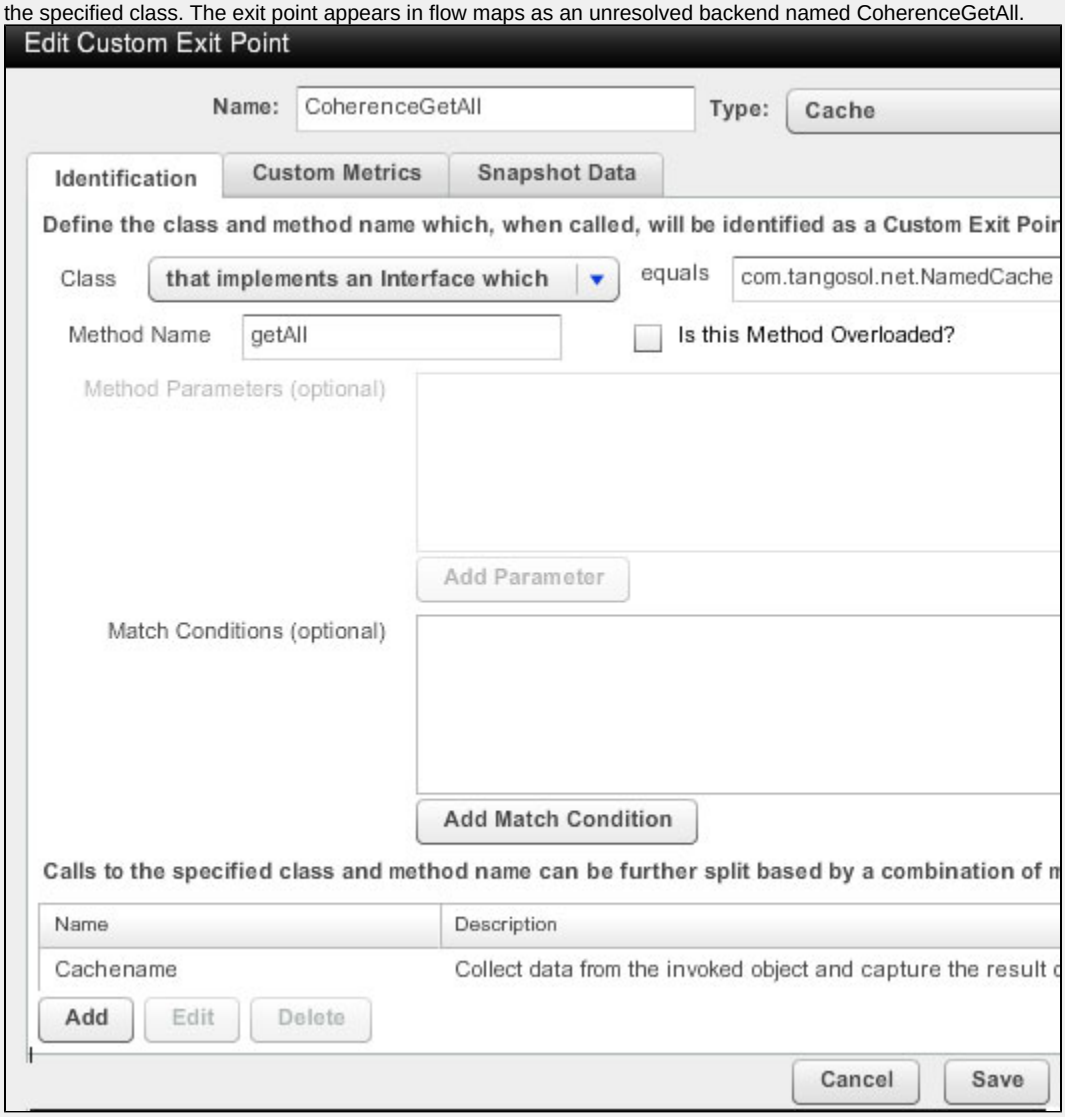
9. Select the type of backend from the Type drop-down menu or check Use Custom if the type is not listed.

10. Configure the class and method name that identify the custom exit point.
If the method is overloaded, check the Overloaded check box and add the parameters.

11. If you want to restrict metric collection based on a set of criteria that are evaluated at runtime, click **Add Match Condition** and define the match condition(s).
For example, you may want to collect data only if the value of a specific method parameter contains a certain value.

12. Click **Save**.

The following screenshot shows a custom exit point of Type Cache. This exit point is defined on the getAll() method of the specified class. The exit point appears in flow maps as an unresolved backend named CoherenceGetAll.



To split an exit point

1. Click **Add**.

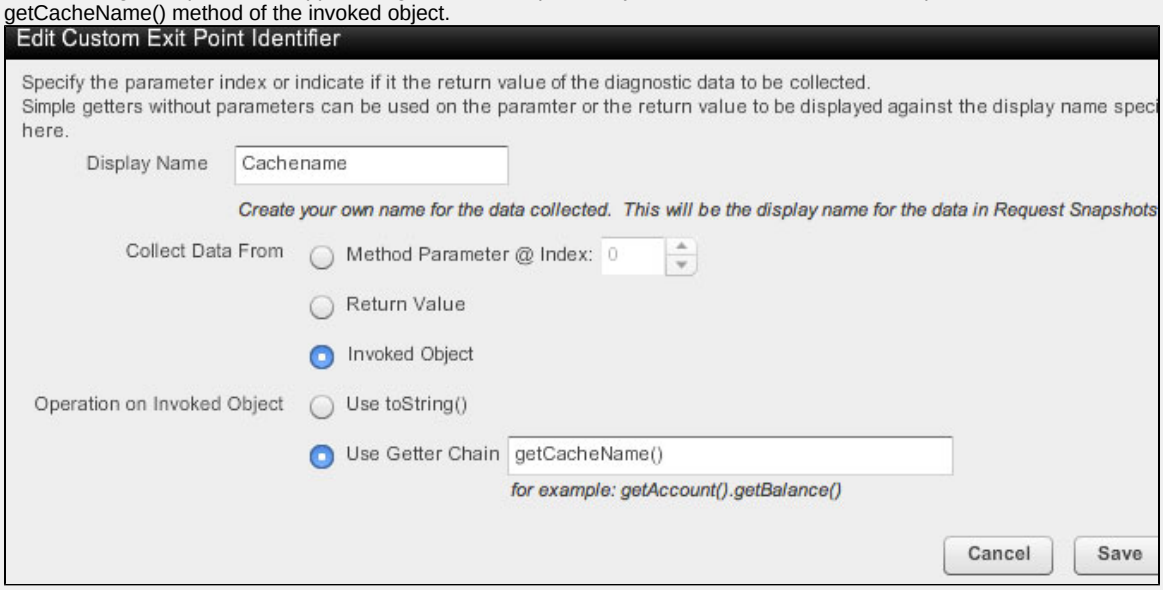
2. Enter a display name for the split exit point.

3. Specify the source of the data (parameter, return value, or invoked object).

4. Specify the operation to invoke on the source of the data (toString() or getter chain for complex objects).

5. Click **Save**.

The following example shows a pplit configuration of the previously created CoherenceGetAll exit point based on the getCacheName() method of the invoked object.

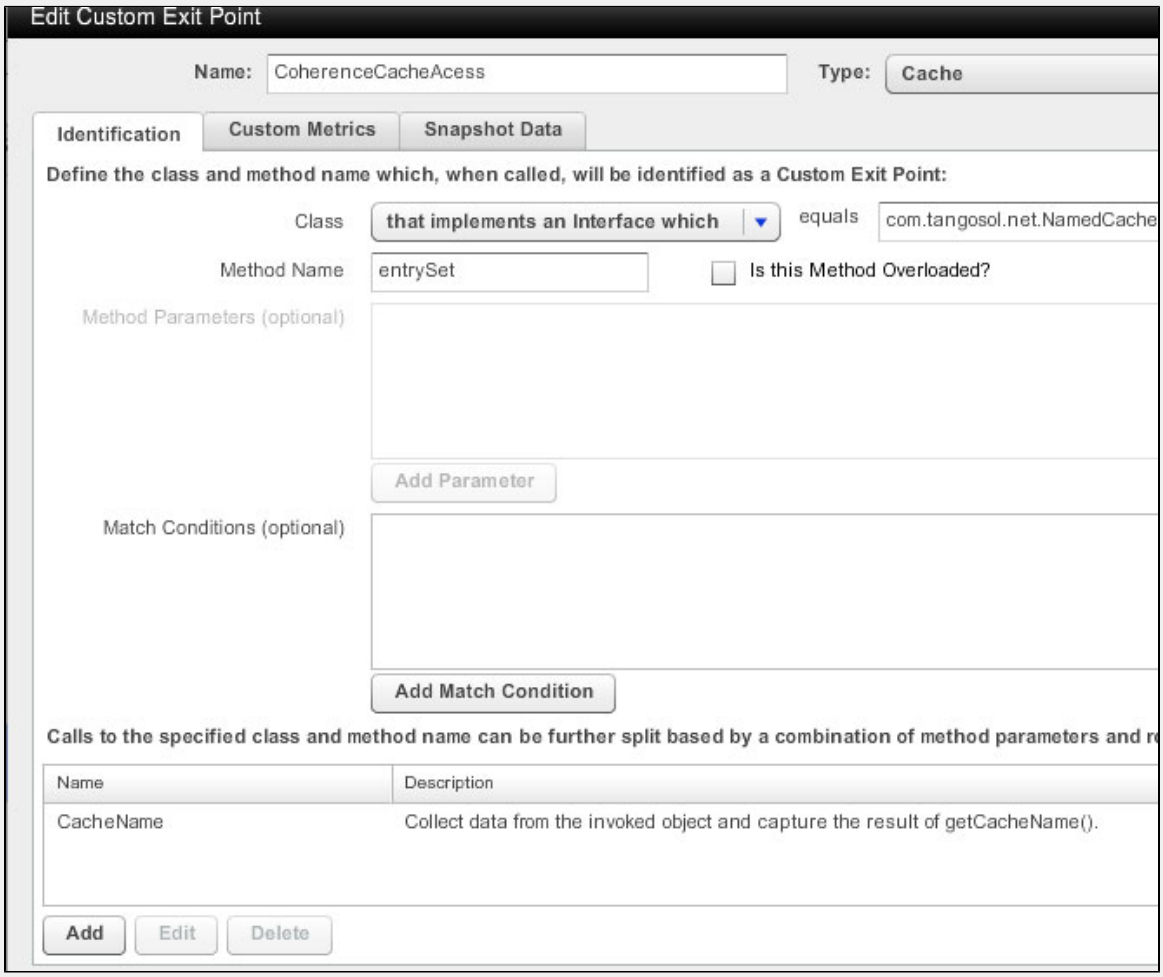


To group an exit point

You can group methods as a single exit point. The only requirement is that these methods point to the same key.

For example, ACME Online has an exit point for NamedCache.getAll. This exit point has a split configuration of getCacheName() on the invoked object as illustrated in the previous screenshot.

Suppose we also define another exit point for NamedCache.entrySet. This is another exit point, but it has the split configuration that has getCacheName() method of the invoked object.



If the getAll() and the entrySet() methods point to the same cache name, they will point to the same backend.

Matching name-value pairs identify the back-end. In this case, there is only one key, i.e. cache name, has to be matched. So, here both exit points have the same name for the cache and they resolve to the same backend.

To define custom metrics for a custom exit point

Custom metrics are collected in addition to the standard metrics.

The result of the data collected from the method invocation must be an integer value, which will either be averaged or added per minute, depending on your selection of data roll-up.

To configure custom business metrics that can be generated from the Java method invocation:

1. Click the **Custom Metrics** tab.

2. Click **Add**.

3. In the Add Custom Metric window, enter a name for the metric.

4. Select the Collect Data From radio button to specify the source of the metric data.

5. Select the Operation on Method Parameter to specify how the metric data is processed.

6. Select how the data should be rolled up (average or sum) from the Data Rollup drop-down menu.

7. Click **Create Custom Metric**.

To define transaction snapshot data collected

1. Click the **Snapshot Data** tab.
2. Click **Add**.
3. In the Add Snapshot Data window, enter a display name for the snapshot data.
4. Select the Collect Data From radio button to specify the source of the snapshot data.
5. Select the Operation on Method Parameter to specify how the snapshot data is processed.
5. Click **Save**.

Learn More

- Configurations for Custom Exit Points

Configurations for Custom Exit Points

- Configurations for Coherence Exit Points
- Configurations For Memcached Exit Points
- Configurations for DangaMemcached Exit Points
- Configurations for SAP Exit Points
- Configurations for EhCache Exit Points
- Configurations for Mail Exit Points
- Configurations for LDAP Exit Points
- Configurations for MongoDB Exit Points
- Learn More

This topic suggests custom exit point configurations that you can create for specific backends.

Configurations for Coherence Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name	Splitting Configuration
Coherence.Put	Cache	put	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.PutAll	Cache	putAll	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.EntrySet	Cache	entrySet	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.KeySet	Cache	keySet	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.Get	Cache	get	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()
Coherence.Remove	Cache	remove	that implements an interface which	com.tangosol.net.NamedCache	getCacheName()

Configurations For Memcached Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name

Memcached.Add	Cache	add	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Set	Cache	set	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Replace	Cache	replace	With a class name that	net.spy.memcached.MemcachedClient
Memcached.CompareAndSwap	Cache	cas	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Get	Cache	get	With a class name that	net.spy.memcached.MemcachedClient
Memcached.Remove	Cache	remove	With a class name that	net.spy.memcached.MemcachedClient

Configurations for DangaMemcached Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
Memcached.Add	Cache	add	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.Set	Cache	set	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.Replace	Cache	replace	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.Delete	Cache	delete	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.Get	Cache	get	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.GetBulk	Cache	getBulk	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.GetMultiArray	Cache	getMultiArray	With a class name that	com.danga.MemCached.MemCachedClient
Memcached.GetMulti	Cache	getMulti	With a class name that	com.danga.MemCached.MemCachedClient

Configurations for SAP Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
SAP.Execute	SAP	execute	With a class name that	com.sap.mw.jco.rfc.MiddlewareRFC$Client
SAP.Connect	SAP	connect	With a class name that	com.sap.mw.jco.rfc.MiddlewareRFC$Client
SAP.Disconnect	SAP	disconnect	With a class name that	com.sap.mw.jco.rfc.MiddlewareRFC$Client
SAP.Reset	SAP	reset	With a class name that	com.sap.mw.jco.rfc.MiddlewareRFC$Client
SAP.CreateTID	SAP	createTID	With a class name that	com.sap.mw.jco.rfc.MiddlewareRFC$Client
SAP.ConfirmTID	SAP	confirmTID	With a class name that	com.sap.mw.jco.rfc.MiddlewareRFC$Client

Configurations for EhCache Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name	Splitting Configuration
EhCache.Get	Cache	get	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.Put	Cache	put	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.PutIfAbsent	Cache	putIfAbsent	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.PutQuiet	Cache	putQuiet	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.Remove	Cache	remove	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.RemoveAll	Cache	removeAll	With a class name that	net.sf.ehcache.Cache	getName()

EhCache.RemoveQuiet	Cache	removeQuiet	With a class name that	net.sf.ehcache.Cache	getName()
EhCache.Replace	Cache	replace	With a class name that	net.sf.ehcache.Cache	getName()

Configurations for Mail Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
MailExitPoint.Send	Mail Server	send	With a class name that	javax.mail.Transport
MailExitPoint.SendMessage	Mail Server	sendMessage	With a class name that	javax.mail.Transport

Configurations for LDAP Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name
LDAPExitPoint.Bind	LDAP	bind	With a class name that	javax.naming.directory.InitialDirCont
LDAPExitPoint.Rebind	LDAP	rebind	With a class name that	javax.naming.directory.InitialDirCont
LDAPExitPoint.Search	LDAP	search	With a class name that	javax.naming.directory.InitialDirCont
LDAPExitPoint.ModifyAttributes	LDAP	modifyAttributes	With a class name that	javax.naming.directory.InitialDirCont
LDAPExitPoint.GetNextBatch	LDAP	getNextBatch	With a class name that	com.sun.jndi.ldap.LdapNamingEnum
LDAPExitPoint.NextAux	LDAP	nextAux	With a class name that	com.sun.jndi.ldap.LdapNamingEnum
LDAPExitPoint.CreatePooledConnection	LDAP	createPooledConnection	With a class name that	com.sun.jndi.ldap.LdapClientFactory
LDAPExitPoint.Search	LDAP	search	With a class name that	com.sun.jndi.ldap.LdapClientFactory
LDAPExitPoint.Modify	LDAP	modify	With a class name that	com.sun.jndi.ldap.LdapClientFactory

Configurations for MongoDB Exit Points

Name of the Exit Point	Type	Method Name	Match Criteria value for the Class	Class/Interface/ Superclass/Annotation Name	Splitting configuration	Snapshot data
MongoDB.Insert	JDBC	insert	With a class name that	com.mongodb.DBCollection	Invoked_Object.getDB(). getName()	Parameter_0.toString()
MongoDB.Find	JDBC	find	With a class name that	com.mongodb.DBCollection	Invoked_Object.getDB(). getName()	Parameter_0.toString()
MongoDB.Update	JDBC	update	With a class name that	com.mongodb.DBCollection	Invoked_Object.getDB(). getName()	Parameter_0.toString()
MongoDB.Remove	JDBC	remove	With a class name that	com.mongodb.DBCollection	Invoked_Object.getDB(). getName()	Parameter_0.toString()
MongoDB.Apply	JDBC	apply	With a class name that	com.mongodb.DBCollection	Invoked_Object.getDB(). getName()	Parameter_0.toString()

Learn More

- Configure Backend Detection
- Configure Custom Exit Points

Code Metric Information Points (Java)

- Code Metric Information Points for Java System Classes
  - To instrument a Java system class
  - Learn More

Code Metric Information Points for Java System Classes

System classes like java.lang.* are by default excluded by AppDynamics. To enable instrumentation for a system class, use code metric information points.

The overhead of instrumenting Java system classes is based on the number of calls. AppDynamics recommends that you instrument only a small number of nodes and monitor the performance for these nodes before adding configuring all the nodes in your system.

To instrument a Java system class

1. Open the <agent_home>/conf/app-agent-config.xml file for the node where you want to enable the metric.

2. Add the fully-qualified system class name to the override exclude section in the XML file. For example, to configure the java.lang.Socket class connect method, modify following element:

```
<override-system-exclude filter-type="equals" filter-value="java.lang.Socket"/>
```

3. Restart those JVMs for which you have modified the XML file.

Learn More

- Code Metrics
- Configure Code Metric Information Points

Configure JMX Metrics from MBeans

- JMX Metric Rules and Metrics
  - Using the JMX Metric Rules Configuration Panel
    - To create one or more JMX metrics in the JMX Metric Rules panel
  - Using the MBean Browser
    - To create a metric from an MBean attribute in the MBean Browser
- Learn More

This topic describes how to create persistent JMX metrics from MBean attributes.

For background information about creating JMX metrics see Monitor JVMs and Monitor JMX MBeans.

JMX Metric Rules and Metrics

A JMX Metric Rule maps a set of MBean attributes from one or more MBeans into AppDynamics persistent metrics. You configure a metric rule that creates one or more metrics in the AppDynamics system. You may want to create new metrics if the preconfigured metrics do not provide sufficient visibility into the health of your system.

After the MBean attribute is configured to provide a persistent metric in AppDynamics, you can use it to configure health rules. For details see Health Rules.

To view the MBeans that are reporting currently in your managed environment use the Metric Browser.

You can use the JMX Metrics Rules Panel or the Using the MBean Browser to create new metrics.

Using the JMX Metric Rules Configuration Panel

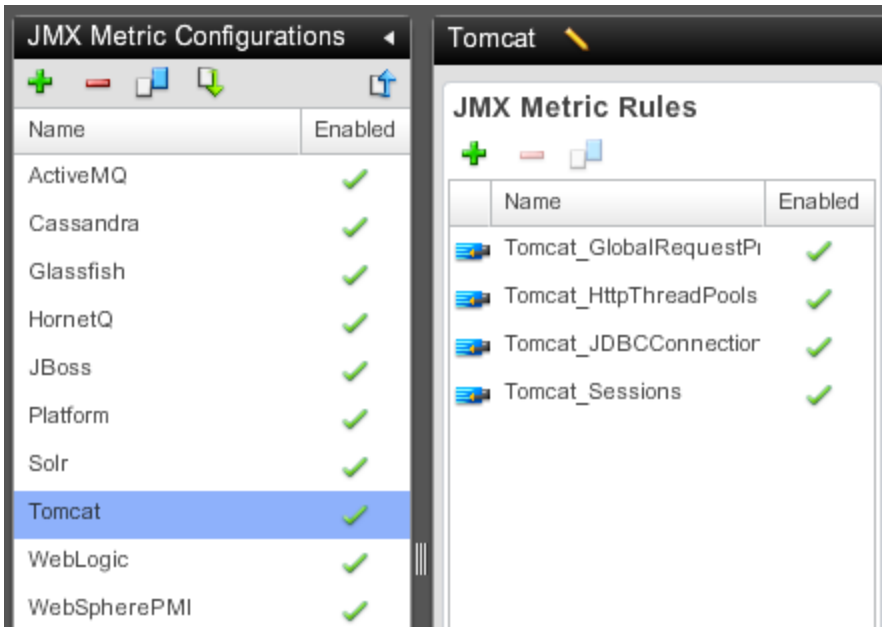
The JMX Metric Rules Panel is the best way to create metrics for multiple attributes based on the same MBean or for complex matching patterns.

To create one or more JMX metrics in the JMX Metric Rules panel

1. In the left navigation pane, click **Configure -> Instrumentation**.

2. Click the **JMX** tab.

3. In the JMX Metric Configurations pane, click the Java platform for which you are configuring metrics.

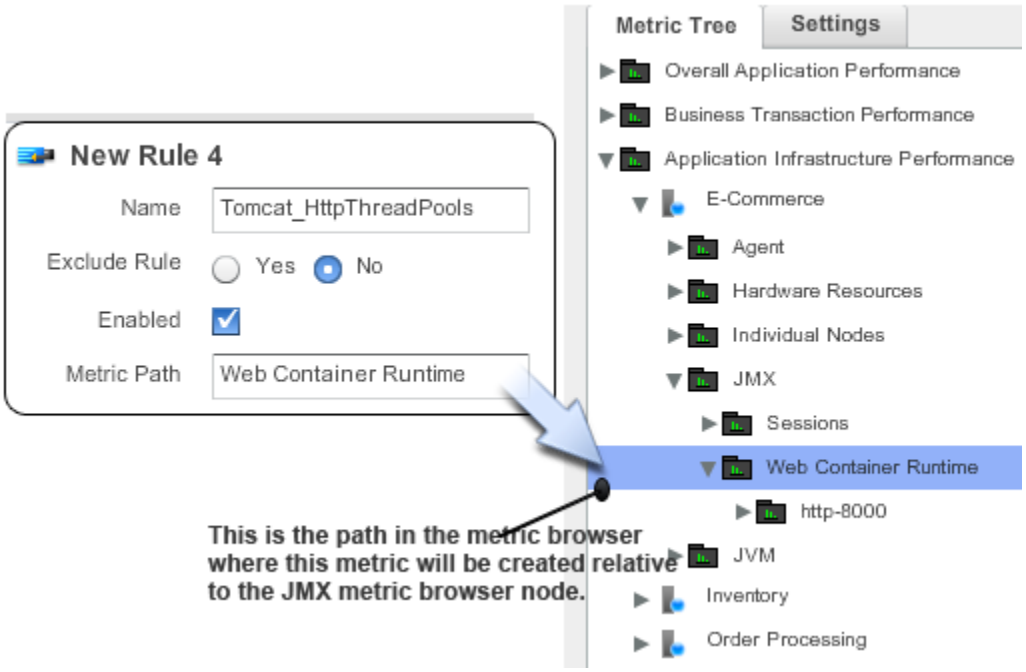


4. Click the + icon . An panel called "New Rule" and the next incremented number opens.

5. Provide the name and settings for this rule:

- The **Name** is the identifier you want to display in the UI.
- An **Exclude Rule** is used for excluding existing rules, so leave the default **No** option.
- **Enabled** means that you want this rule to run, so leave it selected.
- The **Metric Path** is the category as shown in the Metric Browser where the metrics will be displayed. A metric path groups the metrics and is relative to the Metric Browser node.

For example, the following screenshot displays how the JMX Metric Rule "Tomcat_HttpThreadPools" is defined for the ACME Online demo. The metric path is "Web Container Runtime", the category on Metric Browser where all metrics configured under the "Tomcat_HttpThreadPools" Metric Rule will be available.

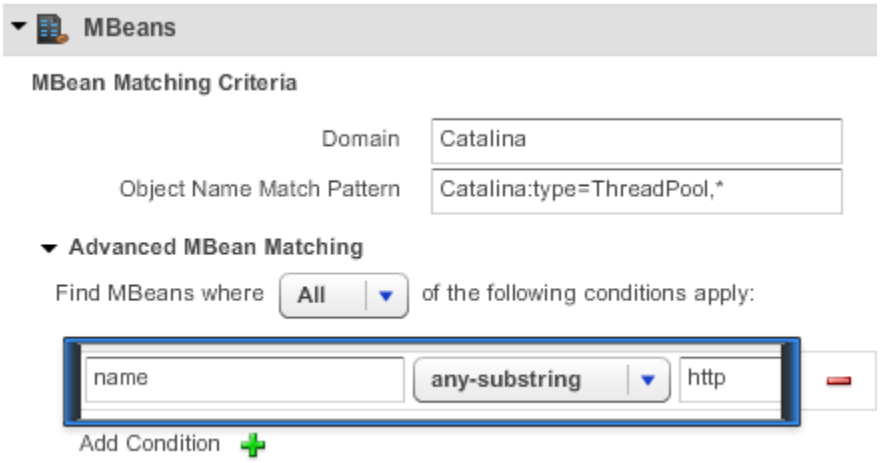


6. Click + icon. An panel called "New Rule" and the next incremented number opens. Specify details for the MBeans that you want to monitor.

- The **Domain name** is the Java domain. This property must be the exact name; no wildcard characters are supported.
- The **Object Name Match Pattern** is the full object name pattern. The property may contain wildcard characters, such as the asterisk for matching all the name/value pairs.

- The **Advanced MBean Matching Criteria** is optional for more complex matching. Use one of the following:
  - any-substring
  - final-substring
  - equals
  - initial-substring

For example, the following screenshot displays the MBean matching criteria for the "Tomcat_HTTPThreadPools" rule.



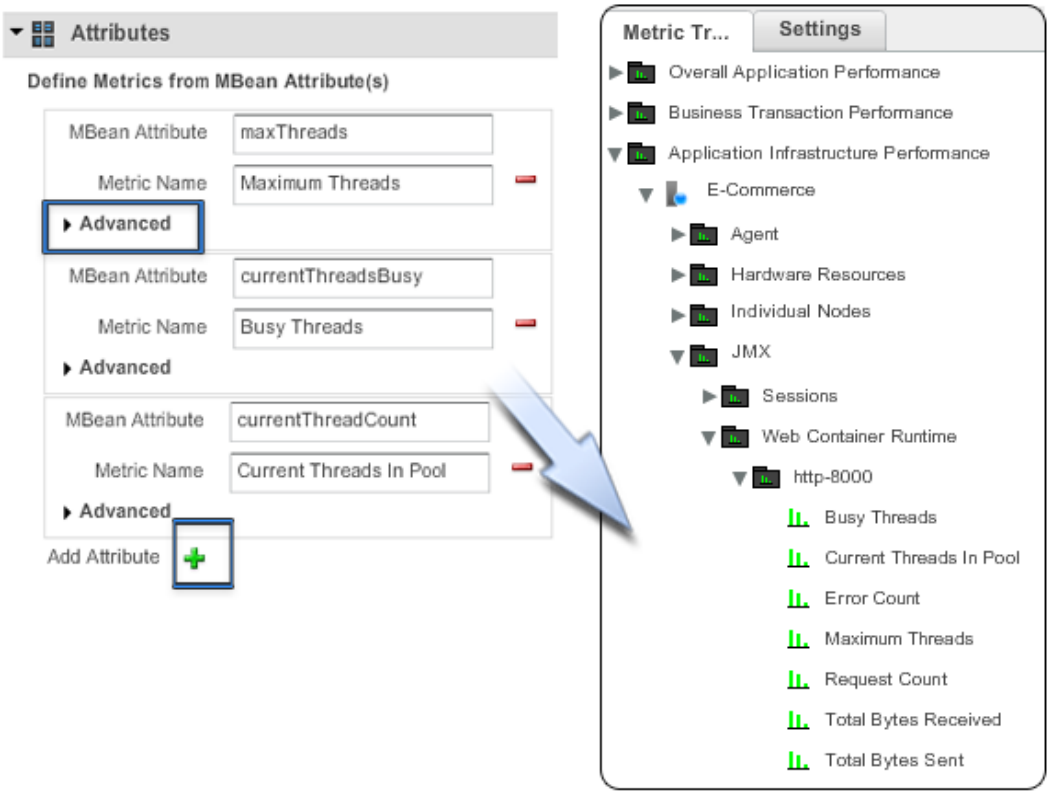
For all MBeans that match the preceding criteria, you can define one or more metrics for the attributes of those MBeans.

7. In the Attributes panel click **Add Attribute** to specify the MBean attributes.

- Provide the name of the attribute and the metric name.
  The metric name will be used to represent the metric in the AppDynamics metric browser.

- Specify any of the following "Advanced Properties" for the attribute:
  - **Metric Time Rollup** determines how the metric will be aggregated over a period of time. You can choose to either average or sum the data points, or use the latest data point in the time interval.
  - **Metric Cluster Rollup** defines how the metric will be aggregated for a tier, using the performance data for all the nodes in that tier. You can either average or sum the data.
  - **Metric Aggregator Rollup** defines how the Agent rolls up multiple individual measurements (observations) into the observation that it reports once a one minute. For performance reasons, Agents report data to the Controller at one minute intervals. Some metrics, such as Average Response Time, are measured (observed) many times in a minute. The Metric Aggregator Rollup setting determines how the Agent aggregates these metrics. You can average or sum observations on the data points or use the current observation. Alternatively you can use the delta between the current and previous observation (*new in 3.4*).

The following screenshot shows how the MBean attributes configured for the Tomcat_HttpThreadPools rule will be displayed in the Metric Browser.

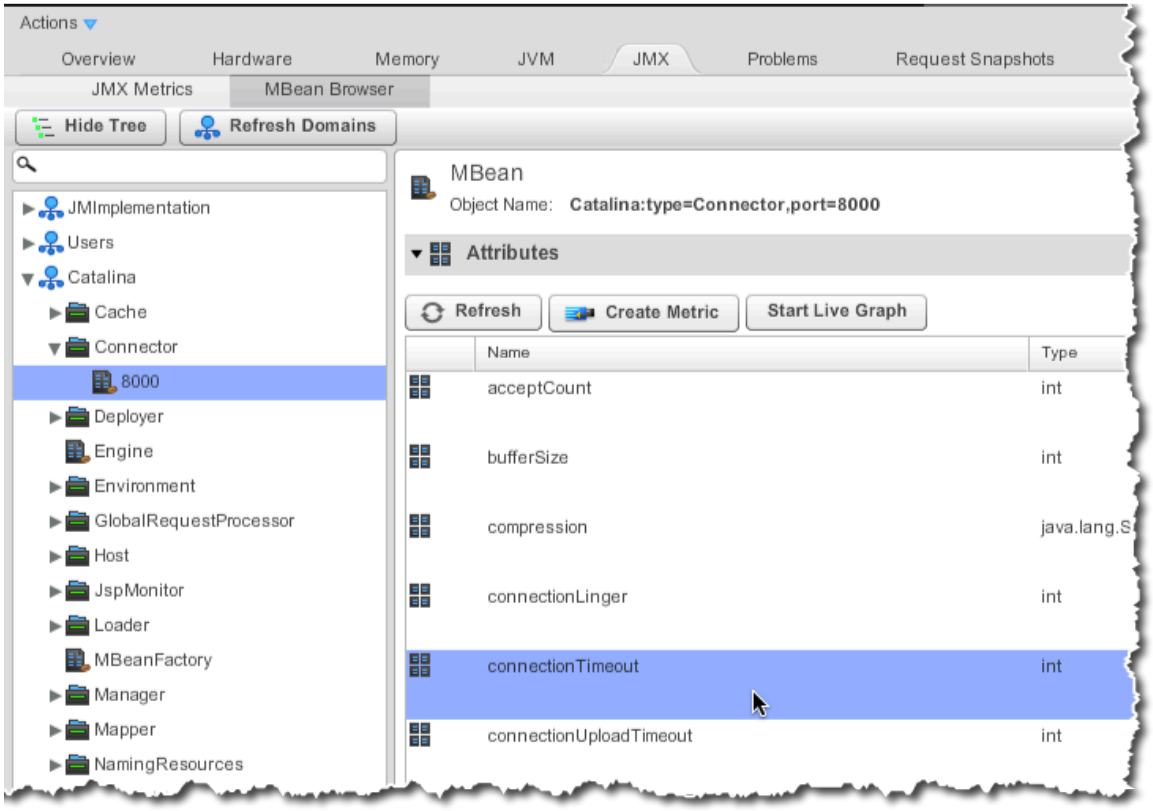


Using the MBean Browser

Sometimes you know exactly which particular MBean attribute you want to monitor. You can select the attribute in the MBean Browser and create a JMX Metric Rule for it.

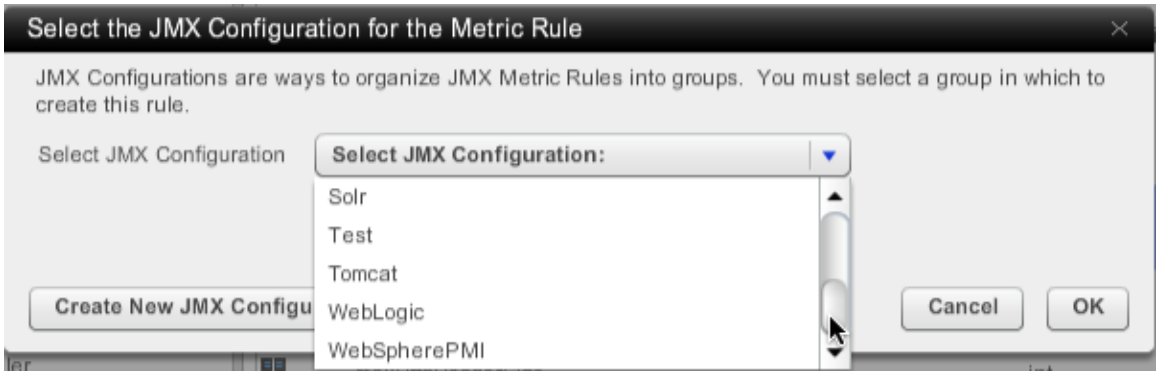
To create a metric from an MBean attribute in the MBean Browser

1. Navigate to the attribute and select it in the MBean Browser.



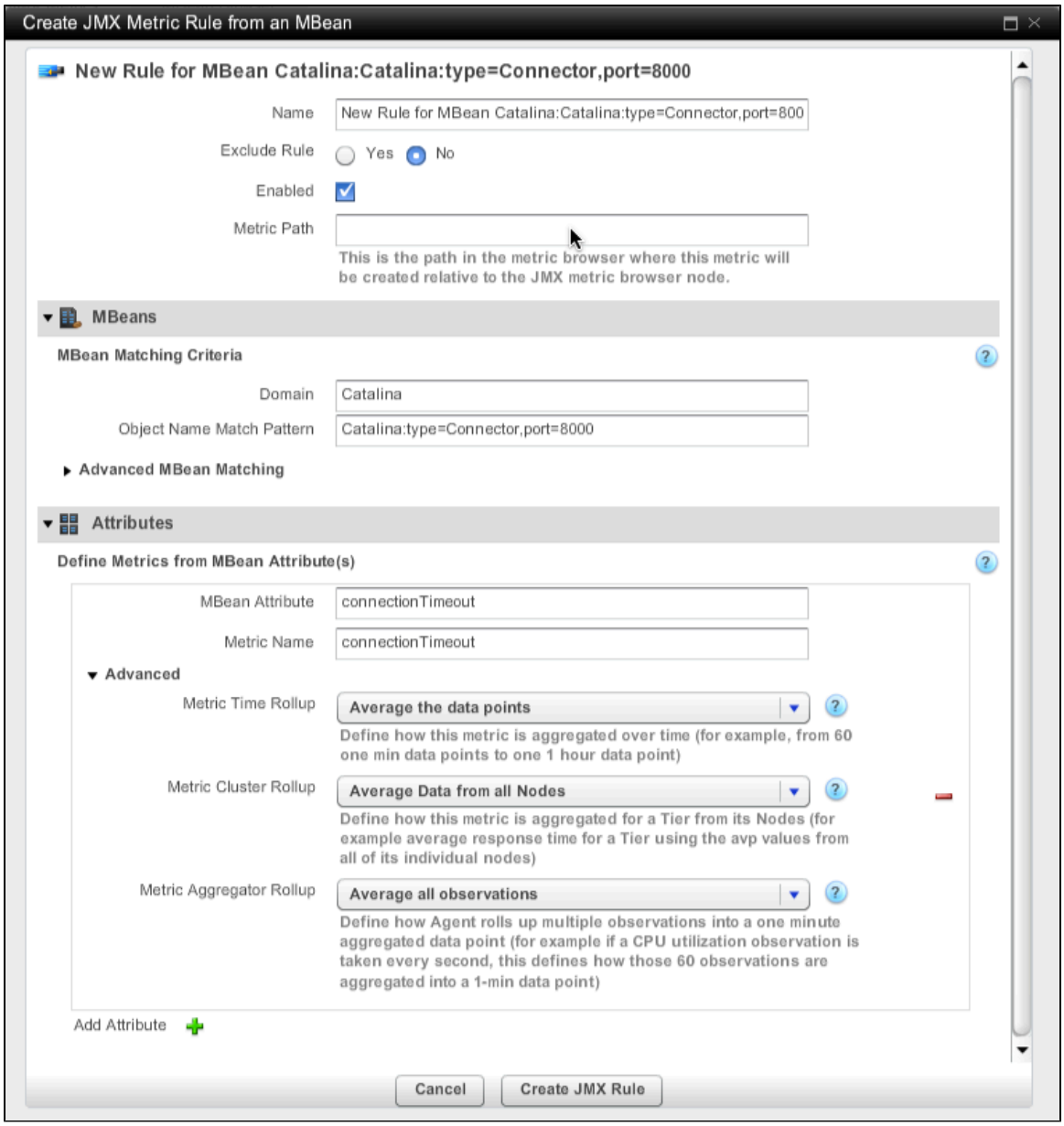
2. Click **Create Metric**.

3. In the **Select JMX Configuration for the Metric Rule** window, select the group in which to create the rule from the **Select JMX Configuration** pulldown. The categories that already exist on your system are listed.



Alternatively, you can create a new group with an name of your choosing. Click **Create New JMX Configuration** to make a new category. This is useful if you want to separate out the custom metrics from the out-of-the-box metrics.

The Create JMX Metric Rule from an MBean window opens.



4. Supply the **Metric Path**, the category as shown in the Metric Browser where the metrics will be displayed. For more discussion of the Metric Path see Step 5 of the previous section.

5. Review the **MBean Matching Criteria** and modify it as needed. Since this is the MBean you selected, you probably do not need to change it.

6. Review the **MBean Attribute** and **Metric Name**. This is the MBean attribute you originally selected. By default the name is the same as the attribute. You can change it if you want to be more specific about its use.

7. Review the **Advanced** panel rollup criteria and update as needed. For more description of these options see Step 7 of the previous section.

8. Click **Create JMX Rule**. The new metric displays in the JMX Metric Browser.

Learn More

- Monitor JVMs
- Monitor JMX MBeans

- Exclude JMX Metrics

Exclude JMX Metrics

- Tuning What Metrics are Gathered
  - To exclude a metric
- Learn More

This topic describes how to exclude MBean attributes from being monitored as JMX metrics.

For background information about JMX metrics see Monitor JVMs and Monitor JMX MBeans.

Tuning What Metrics are Gathered

AppDynamics provides a default configuration for certain JMX metrics. However, in situations where an environment has many resources, there may be too many metrics gathered. AppDynamics lets you exclude resources and particular operations on resources.

To exclude a metric

For example, suppose you want to exclude monitoring for HTTP Thread Pools. Create a new JMX Metrics Rule with the following criteria:

1. Set the Exclude Rule option to **Yes**.

2. Provide the **Object Name Match Pattern:**

```
Catalina:type=ThreadPool,*
```

2. Provide the Advanced MBean Pattern Matching value:

```
http
```

This configuration causes AppDynamics to stop monitoring metrics for HTTP Thread Pools.

Later, if you want to see all HTTP Thread Pool metrics, clear the Enabled checkbox to disable the rule.

Learn More

- Monitor JVMs
- Monitor JMX MBeans
- Configure JMX Metrics from MBeans

Exclude MBean Attributes

- Excluding MBean Attributes from the MBean Browser
  - To exclude an MBean attribute
- Learn More

Excluding MBean Attributes from the MBean Browser

Some MBean attributes contain sensitive information that you do not want the Java Agent to report. You can configure the Java Agent to exclude these attributes using the <exclude object-name> setting in the app-agent-config.xml file.

To exclude an MBean attribute

1. Open the AppServerAgent/conf/app-agent-config.xml file.

2. Locate the JMXService section:

```
<agent-service name="JMXService" enabled="true">
```

3. In the JMXService <configuration> section add the <jmx-metric-browser-excludes> section and the <exclude object-name> property as per the instructions in the comment.

```
<configuration>
                <!--
                Use the below configuration sample to create rules to exclude MBean attributes
from MBean Browser.
                <exclude object-name=<MBean name pattern> attributes=< * |comma separated list
of attribute names> >
                The example below will exclude all attributes of MBeans that match
"Catalina:*".
                <jmx-metric-browser-excludes>
                    <exclude object-name="Catalina:*" attributes="*"/>
                </jmx-metric-browser-excludes>
                -->
 </configuration>
```

4. Save the file.

The new configuration takes effect immediately if the agent-overwrite property is set to true in the app-agent-config.xml. If agent-overwrite is false, which is the default, then the new configuration will be ignored and you have to restart the agent.

Learn More

- App Agent for Java Directory Structure

Configure JMX Without Transaction Monitoring

- Collect Metrics without Transaction Monitoring
  - To turn off transaction detection
- Learn More

Collect Metrics without Transaction Monitoring

In some circumstances, such as for monitoring caches and message buses, you want to collect JMX metrics without the overhead of transaction monitoring.

You can do this by turning off transaction detection at the entry point.

To turn off transaction detection

1. In the left navigation panel, click **Configure -> Instrumentation**.
2. In the Select Application or Tier panel, select the application.
3. In the right panel, click **Java Transaction Detection**.
4. Expand the Entry Points list if it is not already expanded.
5. Clear all the relevant **Enabled** checkboxes in the Transaction Monitoring column.

Transaction monitoring on all selected entry points in the application is disabled. Exit point detection remains enabled.

Learn More

- Configure Business Transaction Detection

Resolve JMX Configuration Issues

- Unable to browse MBeans on WebSphere Application Server (WAS).

- Unable to get metrics from the Java App Server Agent on GlassFish
  - Unable to get JMX metrics for database connections on GlassFish
- Learn More

This topic describes how to resolve issues that may prevent AppDynamics from properly reporting JMX MBean metrics.

Unable to browse MBeans on WebSphere Application Server (WAS).

In certain situations, you may encounter the following exception in the agent.log file for a Java App Server Agent deployed on the WebSphere Application Server (WAS).

```
[AD Thread-Transient Event Channel Poller0] 17 Aug 2011 08:14:08,031
ERROR JMXTransientOperationsHandler - Error trying to lookup clz -
java.lang.ClassNotFoundException: com.ibm.ws.security.core.SecurityContext
```

To resolve this issue:

1. From the WAS administration console, navigate to the JVM settings for the server of interest: **Application servers -> <server> -> Process Definition -> Java Virtual Machine**.

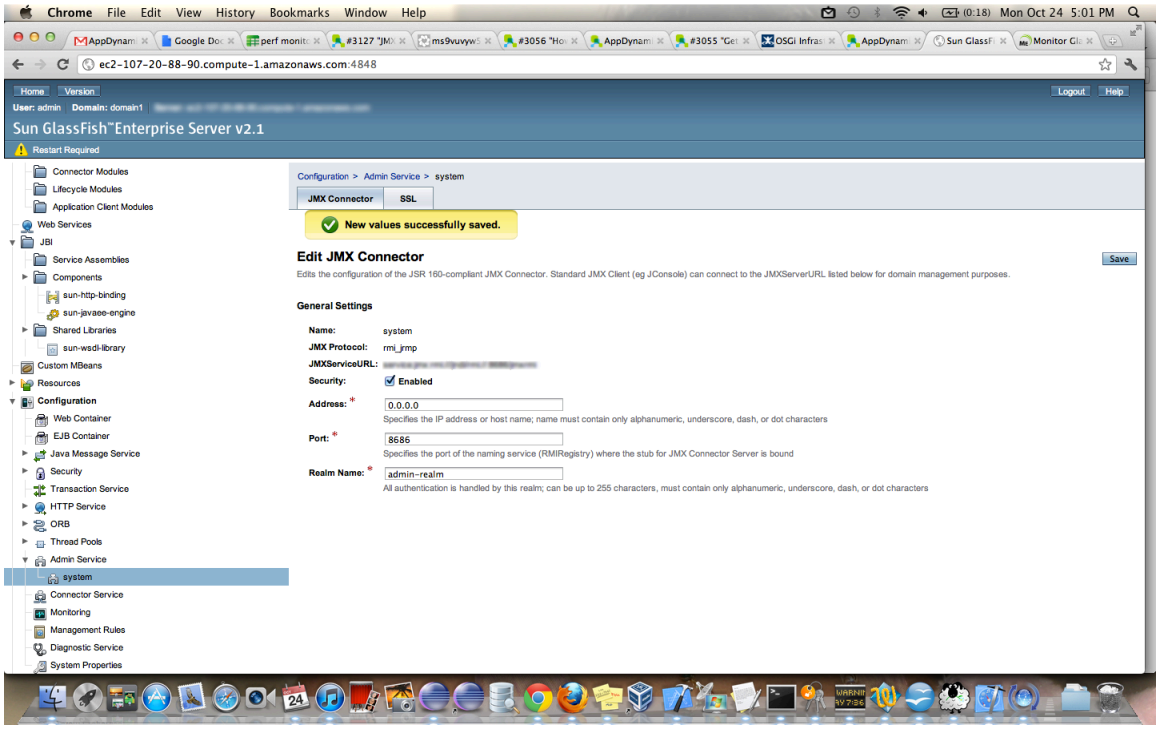
2. Remove the following setting from the generic JVM settings:

```
-Djavax.management.builder.initial = -Dcom.sun.management.jmxremote
```

Unable to get metrics from the Java App Server Agent on GlassFish

Under some situations JMX metrics from GlassFish are not reported. Also some metrics may not be enabled by default. Try these solutions:

1) Confirm that JMX monitoring is enabled in the GlassFish server. Refer to the following screenshot:



2) Copy the text below into an mbean-servers.xml file in the following directory:


```
<App_Agent_Dir>/conf/jmx/
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--<!DOCTYPE servers SYSTEM "mbean-servers.dtd"> -->

<servers xmlns="http://www.appdynamics.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.appdynamics.com
mbean-servers.xsd">
    <!--
    <server mbean-server-name="WebSphere"
mbean-name-pattern="WebSphere:*,type=Server,j2eeType=J2EEServer"
version-attribute="platformVersion" version-startsWith="7"
config-file="servers/websphere-7-jmx-config.xml" />

    <server mbean-server-name="WebSphere"
mbean-name-pattern="WebSphere:*,type=Server,j2eeType=J2EEServer"
version-attribute="platformVersion" version-startsWith="6"
config-file="servers/websphere-7-jmx-config.xml" />
    -->
    <server mbean-server-name="WebSphere"
mbean-name-pattern="WebSphere:*,type=Server"
config-file="servers/websphere-7-jmx-config.xml" />
    <server mbean-server-name="JBoss_4"
mbean-name-pattern="jboss.management.local:j2eeType=J2EEServer,name=Local"
version-attribute="serverVersion" version-startsWith="4"
config-file="servers/jboss-4-jmx-config.xml" />
    <server mbean-server-name="JBoss_5"
mbean-name-pattern="jboss.management.local:j2eeType=J2EEServer,name=Local"
version-attribute="serverVersion" version-startsWith="5"
config-file="servers/jboss-5-jmx-config.xml" />
    <server mbean-server-name="JBoss_6"
mbean-name-pattern="jboss.management.local:j2eeType=J2EEServer,name=Local"
version-attribute="serverVersion" version-startsWith="6"
config-file="servers/jboss-5-jmx-config.xml" />
    <server mbean-server-name="Tomcat_5.5"
mbean-name-pattern="Catalina:type=Server" version-attribute="serverInfo"
version-startsWith="Apache Tomcat/5.5"
config-file="servers/tomcat-5-jmx-config.xml" />
    <server mbean-server-name="Tomcat_6.0"
mbean-name-pattern="Catalina:type=Server" version-attribute="serverInfo"
version-startsWith="Apache Tomcat/6.0"
config-file="servers/tomcat-6-jmx-config.xml" />
    <server mbean-server-name="Tomcat_7"
mbean-name-pattern="Catalina:type=Server" version-attribute="serverInfo"
version-startsWith="Apache Tomcat/7"
config-file="servers/tomcat-7-jmx-config.xml" />
    <server mbean-server-name="Sun GlassFish_2.1"
mbean-name-pattern="com.sun.appserv:j2eeType=J2EEServer,name=server,category=runt
config-file="servers/glassfish-v2-jmx-config.xml" />
    <server mbean-server-name="WebLogic_10"
mbean-server-lookup-string="java:comp/jmx/runtime"
```



```
mbean-name-pattern="com.bea:*,Type=ServerRuntime"
version-attribute="WeblogicVersion" version-startsWith="WebLogic Server 10"
config-file="servers/weblogic-10-jmx-config.xml" />
    <server mbean-server-name="WebLogic_9"
mbean-server-lookup-string="java:comp/jmx/runtime"
mbean-name-pattern="com.bea:*,Type=ServerRuntime"
version-attribute="WeblogicVersion" version-startsWith="WebLogic Server 9"
config-file="servers/weblogic-9-jmx-config.xml" />
    <server mbean-server-name="ActiveMQ_5.3.2"
mbean-name-pattern="org.apache.activemq:*"
config-file="servers/activemq-5.3.2-jmx-config.xml" />
    <server mbean-server-name="Apache Solr 1.4.1" mbean-name-pattern="solr:*"
config-file="servers\solr-1.4.1-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.net:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.db:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.request:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <server mbean-server-name="Apache Cassandra 0.7.0"
mbean-name-pattern="org.apache.cassandra.internal:*"
config-file="servers\cassandra-0.7.0-jmx-config.xml" />
    <!-- If you are using Platform MBean server to report activemq metrics then
you may uncomment the following line.
    -->
    <!--
    <server mbean-server-name="Platform"
mbean-name-pattern="org.apache.activemq:*"
config-file="servers\activemq-5.3.2-jmx-config.xml" />
    -->

    <!-- If your app publishes custom jmx metrics to platform jmx server then
you may modify the platform-jmx-config.xml
    and update the mbean-name-pattern in the following line to start recording
your metrics by appdynamics agent
    -->

    <!--
    <server mbean-server-name="Platform" mbean-name-pattern="com.foo.myjmx:*"
config-file="servers\platform-jmx-config.xml" />
    -->
```



```
</servers>
```

You should see a new JMX node in the metrics tree.

Unable to get JMX metrics for database connections on GlassFish

JDBC connection pool metrics are not configured out-of-the-box for GlassFish. To configure them, uncomment the JDBC connection pool section and provide the relevant information in the following file:

```
<app_agent_install>/conf/jmx/servers/glassfish-v2-jmx-config.xml
```

Uncomment the following section and follow the instructions provided in the file.


```
<!-- The following config can be uncommented to monitor glassfish JDBC
connection pool. Please set the name of the connection
pool (not the datasource name) and enable monitoring for the JDBC Pools on
glassfish admin console. -->
<!--
<metric
mbean-name-pattern="com.sun.appserv:type=jdbc-connection-pool,category=monitor,nar
the name of pool>,*"
category="JDBC Connection Pools">
<attribute-counter-mappings>
<attribute-counter-mapping>
<attribute-name>numconnused-current</attribute-name>
<counter-name>Connections In Use</counter-name>
<counter-type>average</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>numconnused-highwatermark</attribute-name>
<counter-name>Max Connections Used</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>numpotentialconnleak-count</attribute-name>
<counter-name>Potential Leaks</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>averageconnwaittime-count</attribute-name>
<counter-name>Avg Wait Time Millis</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
<attribute-counter-mapping>
<attribute-name>waitqueuelength-count</attribute-name>
<counter-name>Current Wait Queue Length</counter-name>
<counter-type>observation</counter-type>
<time-rollup-type>average</time-rollup-type>
<cluster-rollup-type>individual</cluster-rollup-type>
</attribute-counter-mapping>
</attribute-counter-mappings>
</metric>
```

Learn More

- IBM WebSphere Startup Settings
- Glassfish Startup Settings

Import or Export JMX Metric Configurations

- To import JMX metrics configuration
- To export JMX metrics configuration
- To use a pre-3.3 configuration file for JMX metrics

This topic describes how to import or export your existing JMX configurations.

To import JMX metrics configuration

1. Click **Configure -> Instrumentation**.

2. Click the **JMX** tab.

3. Click the **Import JMX Configuration** icon.

4. In the JMX Configuration Import screen, click **Select JMX Config. File** and select the XML configuration file for your JMX metrics.

5. Click **Import**.

To export JMX metrics configuration

1. Click **Configure -> Instrumentation**.

2. Click the **JMX** tab.

3. Click the **Export JMX Configuration** icon.

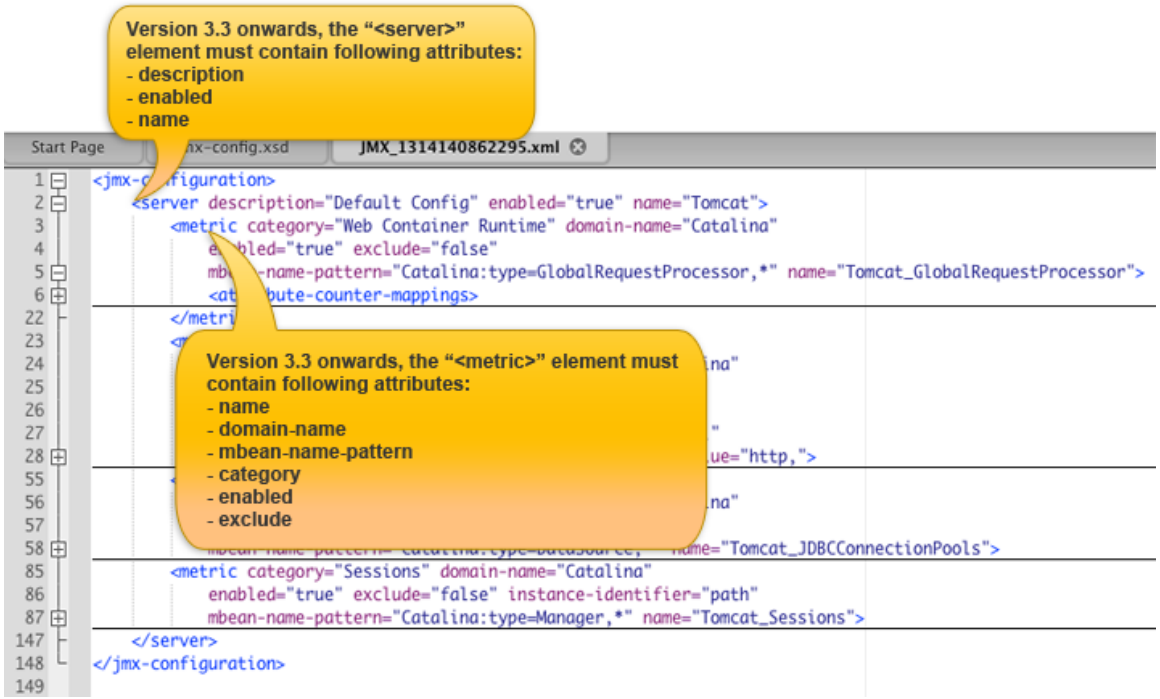
The configuration is downloaded as an XML file.

To use a pre-3.3 configuration file for JMX metrics

In AppDynamics Pro Version 3.3 the structure of the XML-based configuration file for JMX metrics changed. As a result, if you use a pre-3.3 version, you must provide additional attributes in the configuration file.

The following screenshot shows a sample XML configuration file for the JMX metrics for pre-3.3 versions of AppDynamics:

Beginning with AppDynamics version 3.3, the structure for this XML file is the following:



To be able to import your existing configurations for JMX metrics, add the following attributes for <server> element and **each** of the <metric> elements in your XML file.

The attributes for each element are listed below:

Attributes for the <server> element

Attribute Name	Attribute Type	Allowed Values	Mandatory/Optional
description	String		Mandatory
enabled	Boolean	true/false	Mandatory
name	String		Mandatory

Attributes for each <metric> element

For each <metric> element, add the mandatory attributes from the following list to your existing configuration file:

Attribute Name	Attribute Type	Allowed Values	Mandatory/Optional
name	String		Mandatory
domain-name	String	true/false	Mandatory
mbean-name-pattern	String		Mandatory
category	String		Mandatory *All those <metric> elements that have same value for this attribute will be grouped together on the metric browser.*
enabled	Boolean	true/false	Mandatory
exclude	Boolean	true/false	Mandatory
bean-name	String		Optional
query-attribute	String		Optional
query-expression-type		Use any one from the following values: • any-substring • final-substring • equals • initial-substring	Optional
query-value	String		Optional
instance-identifier	String		Optional
instance-name	String		Optional

Configure Custom Memory Structures (Java)

- Custom Memory Structures and Memory Leaks
  - Using Automatic Leak Detection vs Monitoring Custom Memory Structures
  - Supported JVMs
    - To identify custom memory structures
    - To Add a Custom Memory Structure
- Identifying Potential Memory Leaks
  - Diagnosing memory leaks
  - Isolating a leaking collection
  - Access Tracking
- Learn More

This topic describes how to configure custom memory structures and monitor the potential leaks in production environments.

Custom Memory Structures and Memory Leaks

Typically custom memory structures are used as caching solutions. In a distributed environment, caching can easily become a source of memory leaks. AppDynamics helps you to manage and track memory statistics for these memory structures.

AppDynamics provide visibility into:

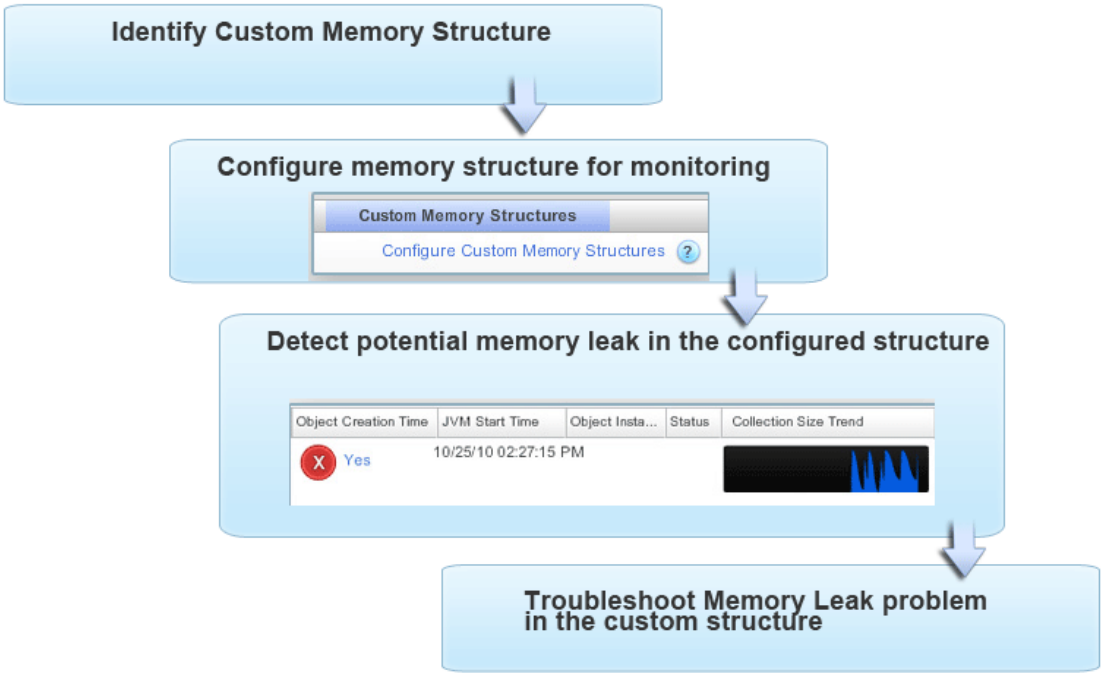
- Cache access for slow, very slow, and stalled business transactions
- Usage statistics, rolled up to the Business Transaction level
- Keys being accessed

- Deep size of internal cache structures

Using Automatic Leak Detection vs Monitoring Custom Memory Structures

The automatic leak detection feature captures memory usage data for all Collections objects in a JVM. However, custom memory structures might or might not contain Collections objects. For example, you can have a custom cache or a third party cache like Ehcache about which you can collect memory usage statistics.

The following illustration provides the work flow for configuring, monitoring, and troubleshooting custom memory structures. You have to configure custom memory structures manually.



Supported JVMs

The supported JVMs for the AppDynamics custom memory monitoring feature are:

- Sun JVM 1.51
- BEA JRockit JVM 1.5 (requires a JVM restart)
- IBM JVM 1.5 (content inspection functionality only)
- Sun JVM 1.6
- BEA JRockit JVM 1.6 (content inspection functionality only)
- IBM JVM 1.6 (content inspection functionality only)

To identify custom memory structures

Enable On-demand Access Tracking in automatic leak detection to capture information on what classes are accessing which Collections objects. Use this information to identify custom memory structures.

AppDynamics captures the top 1000 classes, by instance count.

To Add a Custom Memory Structure

1. From the left navigation pane select **Configure -> Instrumentation**.

2. Click the **Memory Monitoring** tab.

3. In the Tier panel select the tier for which you want to configure a custom memory structure.

4. In the Custom memory Structures section click **Add** to add a new memory structure. The Add Memory Structure window opens.

5. In the Create Memory Structure window

- Specify the configuration name.
- Check **Enabled**.

6. Specify the discovery method.
The discovery method provides three options to monitor the Custom Memory Structure. The discovery method determines how the agent gets a reference to the Custom Memory Structure. AppDynamics needs this reference to monitor the size of the structure. Select any of the three options for the discovery method:

- Discover using Static Field.
- Discover using Constructor.
- Discover using Method.

In many cases, especially with caches, the object for which a reference is needed is created early in the life cycle of the application.

Example for using static field	Example for using Constructor	Example for using method
`public class CacheManager { private static Map userCache<String> User>; }`	`public class CustomerCache { public CustomerCache(); }`	`public Class CacheManager{ public List<Order>; getOrderCache(); {} }`
Notes: Monitors deep size of this Map.	Notes: Monitors deep size of CustomerCache object(s).	Notes: Monitors deep size of this list.

Restart the JVM after the discovery methods are configured to get the references for the object.

5. (Optional) Define accessors.
Click **Define Accessors** to define the methods used to access the custom memory structure. This information is used to capture the code paths accessing the custom memory structure.

6. (Optional) Define the naming convention.
Click **Define Naming Convention**. These configurations differentiate between custom memory structures.

There are situations where more than one custom Caches are used, but only few of them need monitoring. In such a case, use the **Getter Chain** option to distinguish amongst such caches. For all other cases, use either value of the field on the object or a specific string as the object name.

6. Click **Save** to save the configuration.

Identifying Potential Memory Leaks

Start monitoring memory usage patterns for custom memory structures. An object is automatically marked as a potentially leaking object when it shows a positive and steep growth slope. The Memory Leak Dashboard provides the following information:

- **Collection Size:** It provides the number of elements in a Collection.
- **Potentially Leaking:** Potentially leaking Collections are marked as red. It is recommended to start diagnostic sessions on potentially leaking objects.
- **Status:** Indicates if a diagnostic session has been started on an object.
- **Collection Size Trend:** A positive and steep growth slope indicates potential memory leak.

Identify long-lived Collections by comparing the JVM start time and Object Creation Time. After the potentially leaking Collections are identified, start the diagnostic session.

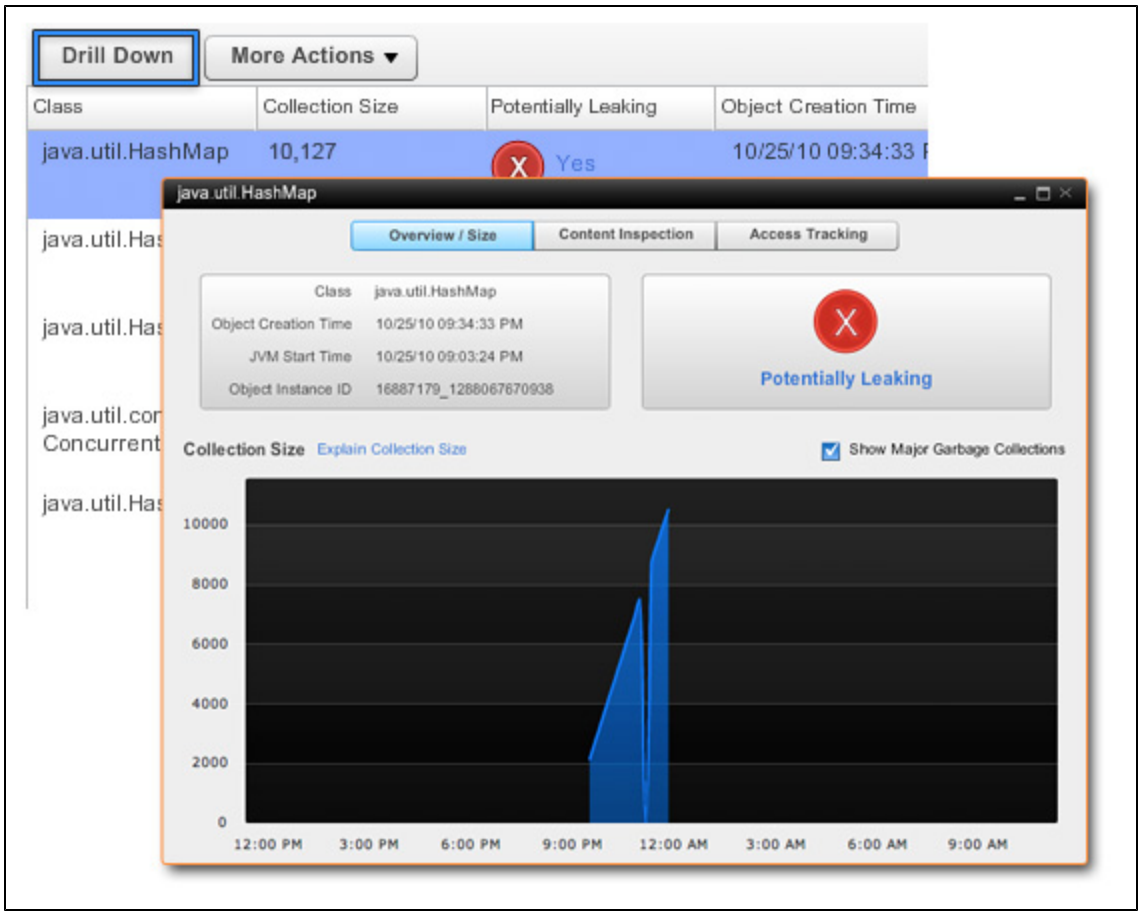
Diagnosing memory leaks

Select the class name to monitor and click **Drill Down** or right-click on the class name and select **Drill Down**.

Isolating a leaking collection

Use Content Inspection to identify to which part of the application the Collection belongs. It allows monitoring histograms of all the elements in a particular memory structure. Start a diagnostic session on the object and then follow these steps:

1. Select **Content Inspection**.
2. Click **Start Content Summary Capture Session**.
3. Enter the session duration. Allow at least 1-2 minutes for the data to generate.
4. Click **Refresh** to retrieve the session data.
5. Click on the particular snapshot to view the details about an individual session.



Access Tracking

Use Access Tracking to view the actual code paths and business transactions accessing the memory structure. Start a diagnostic session on the object and follow these steps:

1. Select the "Access Tracking".

2. Select "Start Access Tracking Session".

3. Enter the session duration. Allow at least 1-2 minutes for data generation.

4. Click the refresh button to retrieve the session data.

5. Click on the particular snapshot, to view the details about an individual session.

Learn More

- Troubleshoot Java Memory Leaks

Configure Object Instance Tracking (Java)

- Prerequisites for Object Instance Tracking
  - To enable object instance tracking on a node
- Tracking Specific Classes
  - To track instances of custom classes
- Learn More

This topic helps you understand how to configure object instance tracking. For more information about why you may need to configure

this, see Troubleshoot Java Memory Thrash.

Prerequisites for Object Instance Tracking

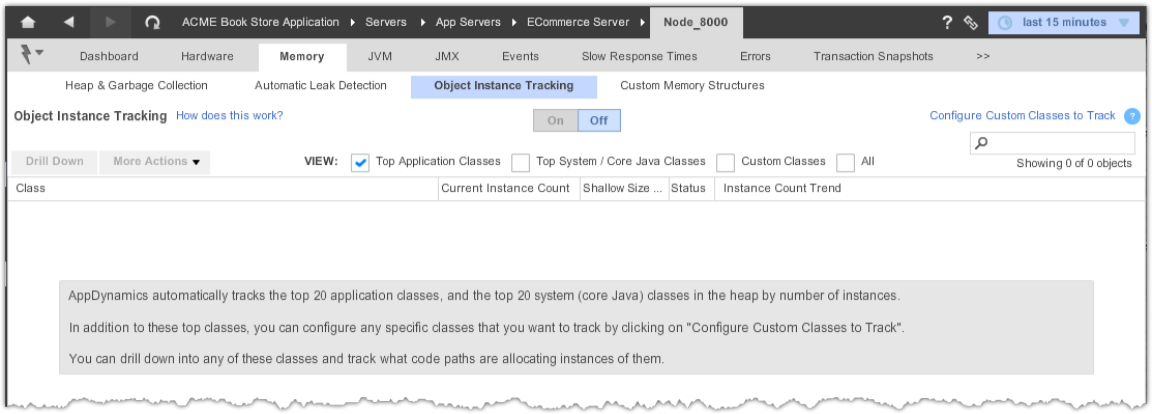
- Object Instance Tracking can be used only for Sun JVM v1.6.x and later.
- If you are running with the JDK then tools.jar will already be in the CLASSPATH, but if you are running with the JRE, you must add tools.jar to the CLASSPATH and restart the JVM for this feature to start working.

Adding tools.jar file:
For AppDynamics to read the tools.jar, add this file to jre/lib/ext directory and to the classpath as shown below (along with other -jar options):

```
java -classpath <complete-path-to-tools.jar>;%classpath% -jar myApp.jar
OR
java -classpath <complete-path-to-tools.jar>:$classpath -jar myApp.jar
```

To enable object instance tracking on a node

1. In the left navigation pane, click **Servers -> App Servers -> <tier> -> <node>**. The Node Dashboard opens.
2. Click the Memory tab.
3. Click the Object Instance Tracking subtab.
4. Click the ON button.



Tracking Specific Classes

Check against the required set of classes to enable instance tracking for each set. For improved performance, only the top 20 classes are tracked.

Use Configure Custom Classes to track instances of specific classes.

Classes configured here will only be tracked if their instance count is among the top 1000 instance counts in the JVM.

To track instances of custom classes

1. In the left navigation pane, click **Servers -> App Servers -> <tier> -> <node>**. The Node Dashboard opens.
2. Click **Configure Custom Classes To Track** on the rightmost corner of the screen.
3. Click **Add**.
4. Click **Enabled**.

Learn More

- Troubleshoot Java Memory Thrash

Configure Memory Monitoring (Java)

- Prerequisites for Object Instance Tracking
  - To enable object instance tracking
- Tracking Specific Classes
  - To Specify Custom Classes to Track
- Learn More

This topic helps you understand how to configure memory monitoring, also known as object instance tracking. For more information about why you may need to configure this, see Troubleshoot Java Memory Thrash.

Prerequisites for Object Instance Tracking

Object Instance Tracking can be used only for Sun JVM v1.6.x and later.

If you are running with the JDK, tools.jar will already be in the CLASSPATH, but if you are running with the JRE, you must add tools.jar to the CLASSPATH and restart the JVM for this feature to start working.

Adding tools.jar file:
For AppDynamics to read the tools.jar, add this file to jre/lib/ext directory and to the classpath as shown below (along with other -jar options):

```
java -classpath <complete-path-to-tools.jar>;%classpath% -jar myApp.jar
OR
java -classpath <complete-path-to-tools.jar>:$classpath -jar myApp.jar
```

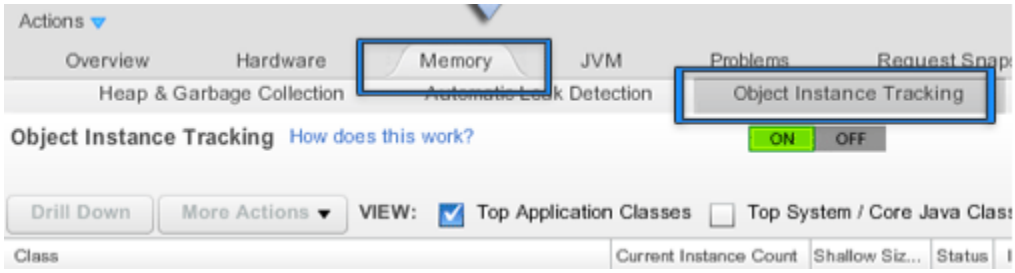
To enable object instance tracking

1. Navigate to the node dashboard of the node for which you want to enable object instance tracing..

2. Click the **Memory** tab.

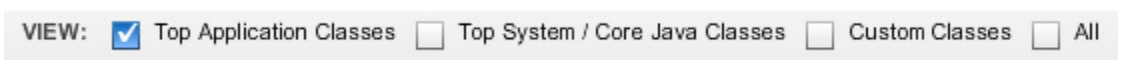
3. Select **Object Instance Tracking**.

4. Turn the object instance tracking mode ON.



Tracking Specific Classes

Check against the required set of classes to enable instance tracking for each set. For improved performance, only the top 20 classes are tracked.



Use Configure Custom Classes to track instances of specific classes.

Classes configured here will only be tracked if their instance count is among the top 1000 instance counts in the JVM.

To Specify Custom Classes to Track

1. Click **Configuration -> Instrumentation -> Memory Monitoring**.
2. Select the tier in which to configure object tracking.
3. In the Object Instance Tracking. Define Custom Classes to Track section, click **Add**.
4. In the Create New Instance Tracker window, check **Enabled**.
5. Enter the fully qualified class name of the class to track.
6. Click **Save**.

You can edit or delete the object tracing configuration after it has been created.

Learn More

- Troubleshoot Java Memory Thrash

Configure Multi-Threaded Transactions (Java)

- Default Configuration
- Custom Configuration
  - Managing Thread Correlation Using a Node Property
- Enabling and Disabling Asynchronous Monitoring
  - To Disable Asynchronous Monitoring
  - To Enable Asynchronous Monitoring
- Learn More

AppDynamics collects and reports key performance metrics for individual threads in multi-threaded Java applications. See Trace Multi-Threaded Transactions (Java) for details on where these metrics are reported.

Default Configuration

Classes for multi-threaded correlation are configured in the <excludes> child elements of the <fork-config> element in the <App_Server_Agent_Installation_Directory>/conf/app-agent-config.xml file.

The default configuration excludes the java, org, weblogic and websphere classes:

```
<fork-config>
    <!-- exclude java and org -->
    <excludes filter-type="STARTSWITH"
filter-value="java/,javax/,com.sun/,sun/,org/"/>
    <!-- exclude weblogic and websphere -->
    <excludes filter-type="STARTSWITH"
filter-value="com.bea/,com.weblogic/,weblogic/,com.ibm/"/>
    . . .
```

Custom Configuration

You can edit the app-agent-config.xml file to exclude additional classes from thread correlation. All classes not excluded are by default included.

You can also explicitly include sub-packages and subclasses of excluded packages and classes. Use the <includes> or <include> child element to specify the included packages and classes.

Use the <excludes> and <includes> elements to specify a comma-separated list of classes or packages. Use the <exclude> and <include> elements to specify a single class or package

Managing Thread Correlation Using a Node Property

You can also configure which classes or packages to include or exclude using a node property. See thread-correlation-classes and thread-correlation-classes-exclude.

Enabling and Disabling Asynchronous Monitoring

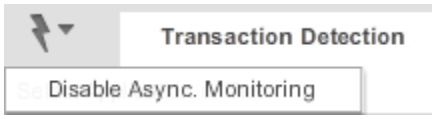
You should disable monitoring of multi-threaded transactions on all agents if all of your agents and your controller are not at AppDynamics version 3.6 or higher. You can enable the feature after all of your agents have been upgraded.

You must restart the agent after you enable or disable this feature.

To Disable Asynchronous Monitoring

1. In the left navigation pane click **Configure->Instrumentation->Transaction Detection**.

2. From the Actions menu in the upper left corner click **Disable Async Monitoring**.



To Enable Asynchronous Monitoring

1. In the left navigation pane click **Configure->Instrumentation->Transaction Detection**.

2. From the Actions menu in the upper left corner click **Enable Async Monitoring**.

Learn More

- Configure Business Transaction Detection
- App Agent Node Properties

Configure Background Tasks (Java)

- Pre-Configured Frameworks for Java Background Tasks
- Enabling Automatic Discovery for Background Tasks
  - To enable discovery for a background task using a common framework
- Configuring Background Batch or Shell Files using a Main Method
  - To instrument the main method of a background task
- Learn More

In a Java environment background tasks are detected using POJO entry points. It is the same basic procedure as defining entry points for business transactions, except that you check the Background Task check box. For instructions see Configure Background Tasks.

Pre-Configured Frameworks for Java Background Tasks

When enabled, AppDynamics provides discovery for the following Java background-processing task frameworks:

- Quartz
- Cron4J
- JCronTab
- JavaTimer

Configure Background Tasks

Enabling Automatic Discovery for Background Tasks

Automatic discovery of background tasks is disabled by default. When you know that there are background tasks in your application environment and you want to monitor them, first enable automatic discovery so that AppDynamics will detect the task.

AppDynamics provides preconfigured support for some common frameworks. If your application is not using one of the default frameworks you can create a custom match rule.

To enable discovery for a background task using a common framework

1. In the left navigation pane, click **Configure -> Instrumentation**.

2. On the Transaction Detection tab, select the tier for which you want to enable monitoring.

3. Click **Use Custom Configuration for this Tier**.

4. Scroll down to the Custom Match Rules pane.

5. Do one of the following

   - If you are using a pre-configured framework, select the row of the framework and click the pencil icon, or double-click on the row to open the Business Transaction Match Rule window. By default the values are populated with rule name and the class and method names for the particular framework. Verify that those are the correct names for your environment.
     OR
   - If you are using a custom framework, select the match criteria and enter the Class Name and Method Name.

The Background Task check box should be already checked.

6. Check **Enabled**.

7. Click **Save**.

The custom match rule for the background task will take effect and the background task will display in the Business Transaction List.

Once you enable discovery, every background task is identified based on following attributes:

- Implementation class name
- Parameter to the execution method name

Configuring Background Batch or Shell Files using a Main Method

Sometimes background tasks are defined in batch or shell files in which the main method triggers the background processing. In this situation, the response time of the batch process is the duration of the execution of the main method.

IMPORTANT: Instrument the main method only when the duration of the batch process is equal to the duration of the main method. Otherwise choose another method that accurately represents the unit of work for the background process.

To instrument the main method of a background task

1. In the left navigation pane, click **Configure -> Instrumentation**.

2. In the **Transaction Detection** section select the tier for which you want to instrument the main method.

3. In the **Custom Rules** section click **Add** (the "+" icon).

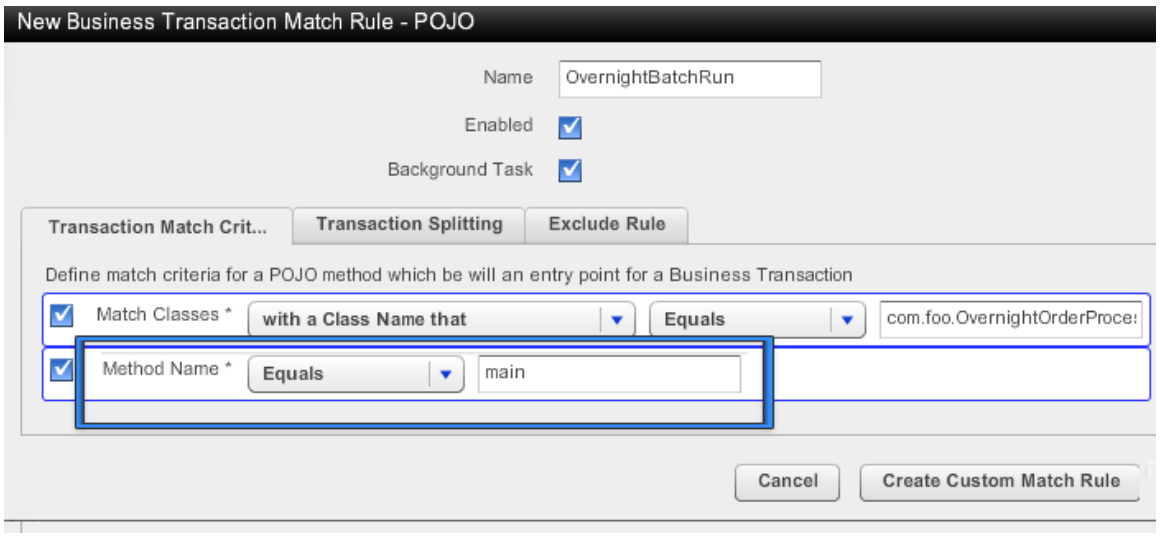
4. From the **Entry Point** type drop down list, click **POJO**.

5. Enter a name for the custom rule.

6. Check **Background Task**.

7. Check **Enabled**.

8. Enter "main" as the match value for **Method Name**.



9. Save the changes.

10. To ensure that the name of the script file is automatically picked up as a background task, configure your Java Agent for that node. See Configure App Agent for Java for Batch Processes.

Learn More

- Configure Background Tasks
- Configure App Agent for Java for Batch Processes
- POJO Entry Points

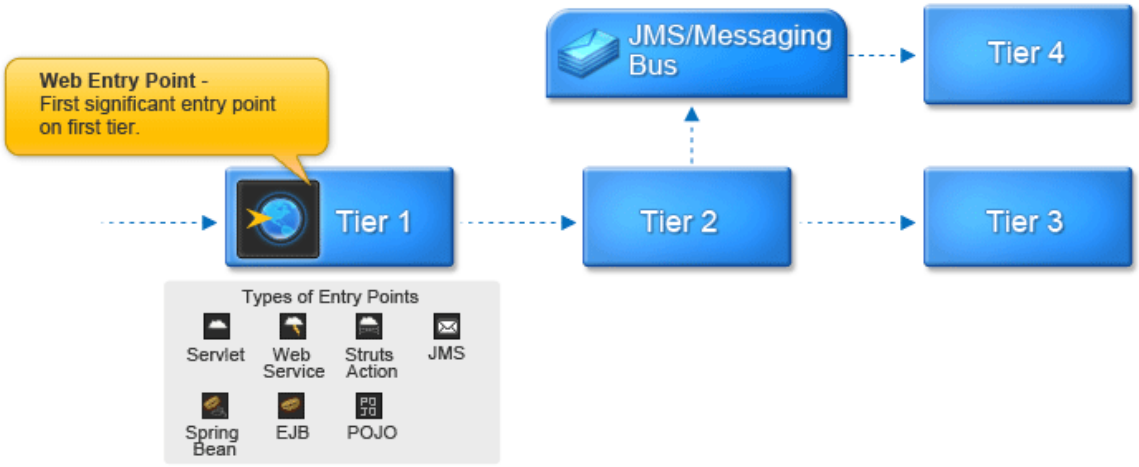
Web Application Entry Points

- Tiers and Web Application Entry Points
- Other Web Application Frameworks Based on Servlets or Servlets Filter
- Learn More

This section discusses web application entry points for different types of business transactions.

Tiers and Web Application Entry Points

A tier can have multiple entry points.



For example, for Java frameworks a combination of pure Servlets or JSPs, Struts, Web services, Servlet filters, etc. may all co-exist on the same JVM.

The middle-tier components like EJBs and Spring beans are usually not considered Entry Points because they are normally accessed using either the front-end layers such as Servlets or from classes that invoke background processes.

Other Web Application Frameworks Based on Servlets or Servlets Filter

AppDynamics provides out-of-the-box support for most of the common web frameworks that are based on Servlets or Servlet Filters. When using any of the frameworks listed below, refer to the Servlet discovery rules to configure transaction discovery.

- Spring MVC
- Wicket
- Java Server Faces (JSF)
- JRuby
- Grails
- Groovy
- Tapestry
- ColdFusion

Learn More

Collapse all
Expand all Collapse all

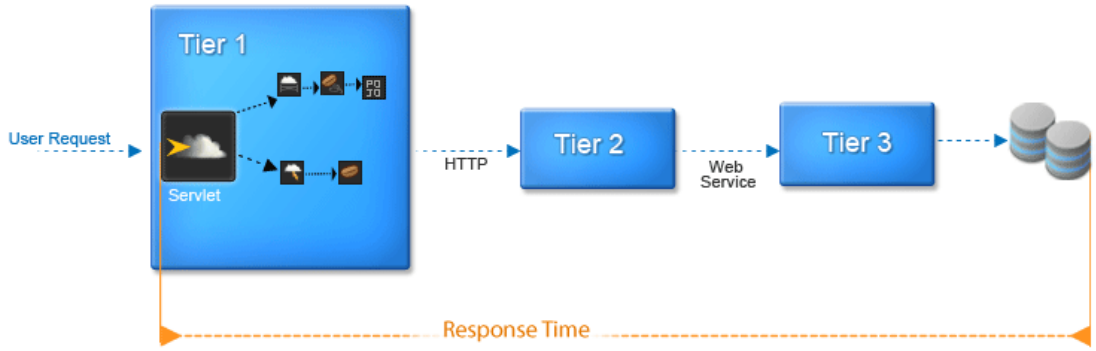
Servlet Entry Points

- Servlet-Based Business Transactions
    - Default Naming for Servlet-Based Transactions
    - Naming Using Other URI Patterns
    - Using Headers, Cookies, and Other Parts of HTTP Requests for identifying Transactions
    - Naming Servlet Based Transactions when Different Web Contexts Require Different Naming Strategies
    - Custom Match Rules for Servlet Based Transactions
        - Example 1: URL is http://acmeonline.com/store/checkout
        - Example 2: URL is http://acmeonline.com/secure/internal/updateinventory
        - Example 3: URL is http://acmeonline.com/orders/process?type=creditcard
    - Using Custom Expressions in Custom Match Rules
    - Request Attributes in the Custom Expression
- Configuring Transaction Identification for REST-Style URLs
- Configuring Transaction Identification Based on Information in the Body of the Servlet Request

This describes how to configure transaction entry points for Servlet-based methods that may or may not be used as part of an application framework.

Servlet-Based Business Transactions

AppDynamics allows you to configure a transaction entry point on the invocation of the service method of a Servlet. The response time for the Servlet transaction is measured when the Servlet entry point is invoked.



Default Naming for Servlet-Based Transactions

By default, AppDynamics identifies all Servlet-based transactions using the first two segments of the URI.

For example, if the URI for a checkout operation in an online store is

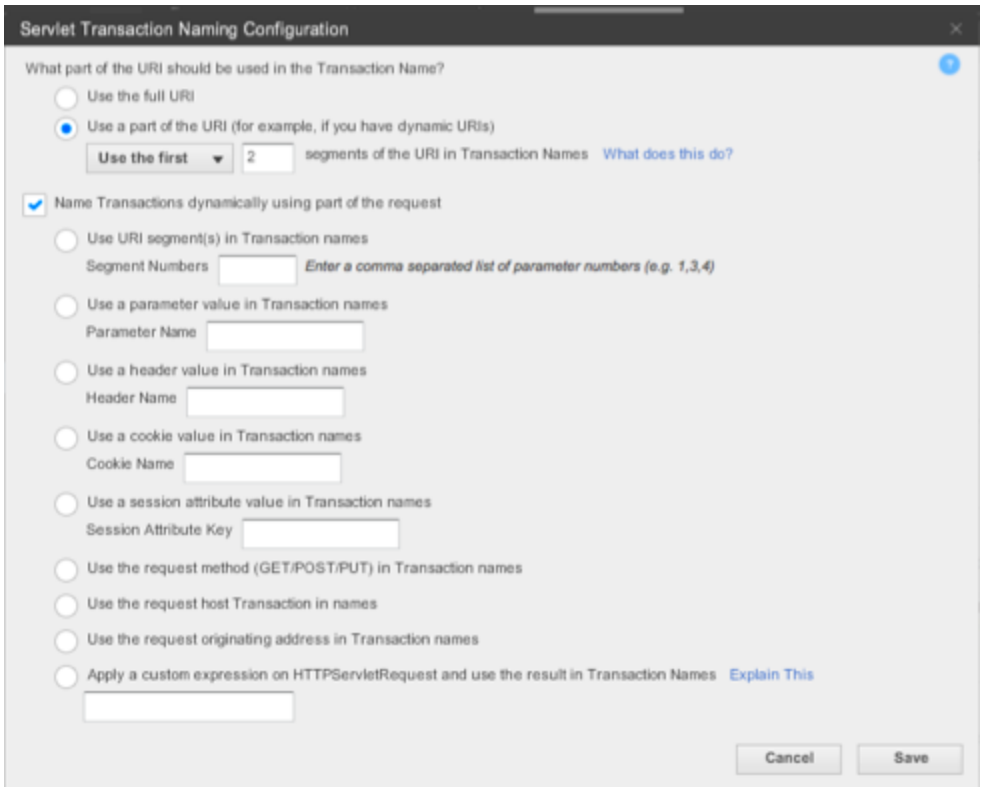
http://acmeonline.com/store/checkout

AppDynamics automatically names this transaction "store/checkout".

If the URI for a transfer funds operation in an online bank is

http://acmebank.com/account/transferfunds/northerncalifornia

AppDynamics automatically names this transaction "/account/transferfunds".



Naming Using Other URI Patterns

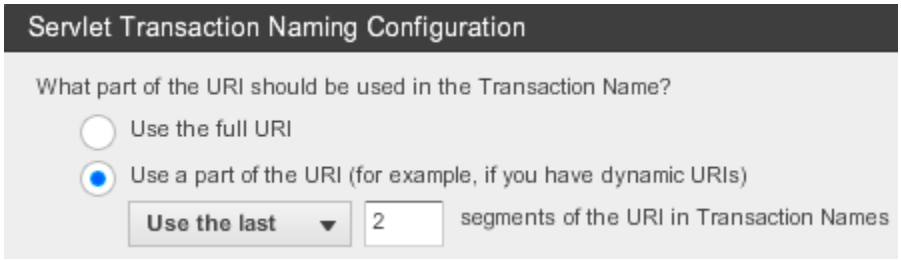
The AppDynamics default naming convention might not be optimal for all applications. You can configure AppDynamics to use different segments of the URI or other values (such as a header value, a cookie the request method, etc.).

For example the following URL represents checkout operation in ACME Online:

http://acmeonline.com/web/store/checkout

AppDynamics' default naming scheme identifies these transactions by the first two segments of the URI: "/web/store".

This naming convention does not indicate the functionality of the operation (checkout, add to cart, etc.) A better approach would identify such URLs using the last two segments: "store/checkout".



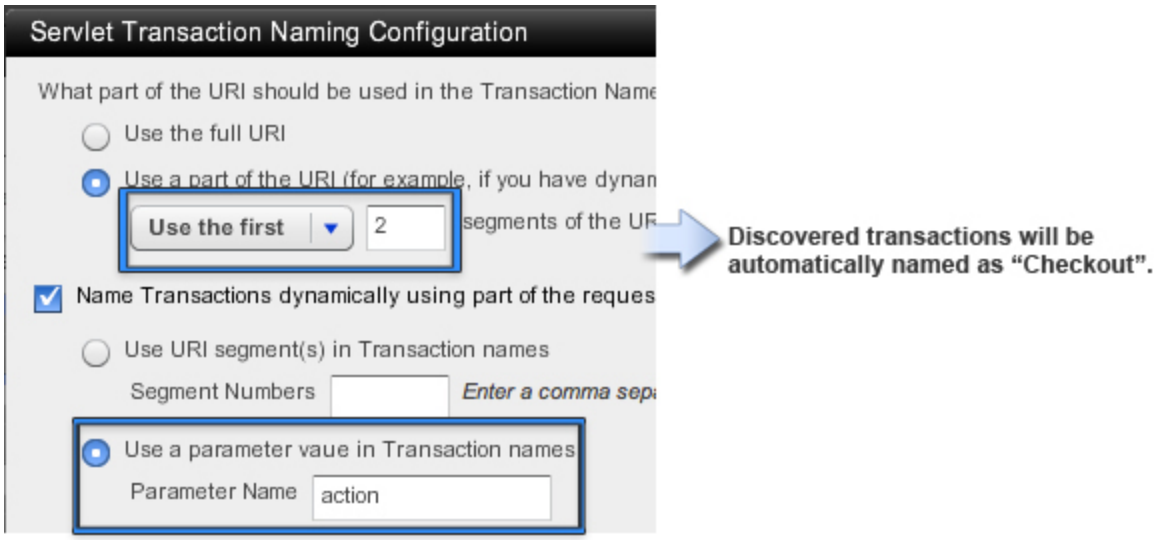
In certain situations, the URI might not contain enough information to be able to name the transaction. This is very common in dispatcher-style Servlet patterns where the Servlet dispatches requests based on a query parameter.

For example, for the following URL

http://acmeonline.com/dispatcher?action=checkout

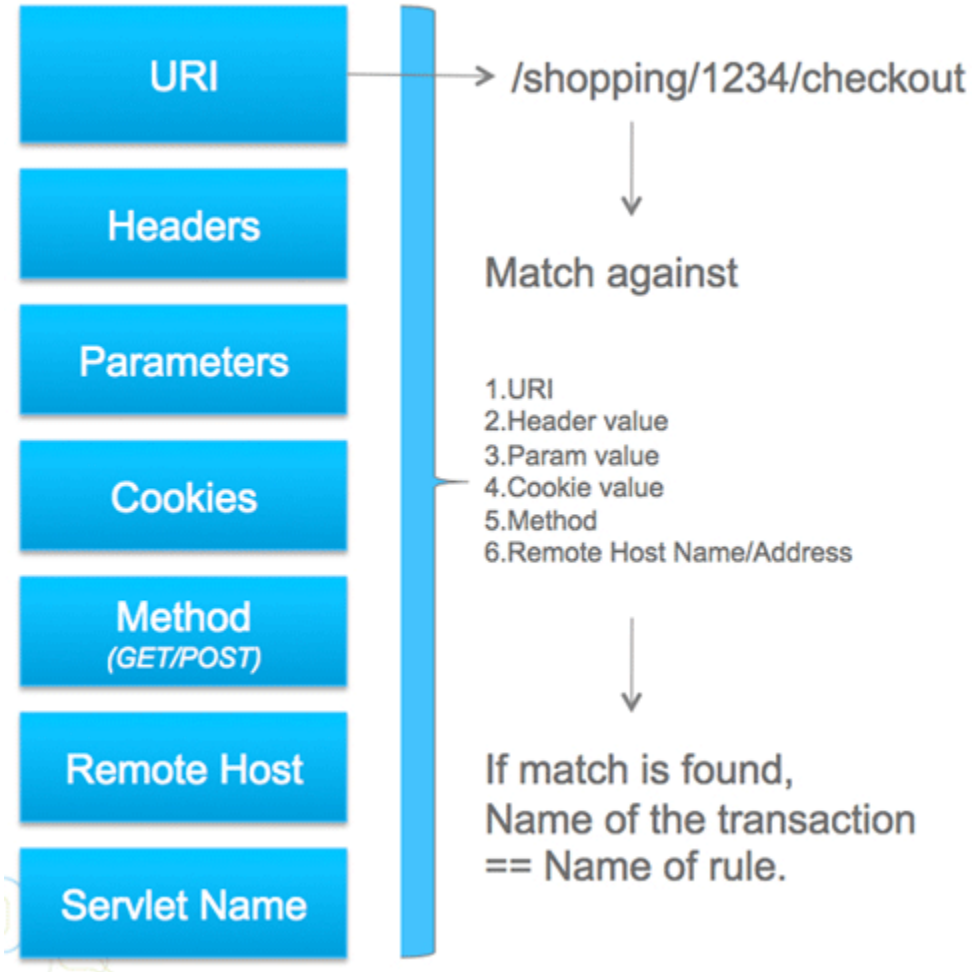
it is ideal to name the transaction based on the value of the action parameter.

To configure this, specify the transaction detection based on the parameter value for the "action" parameter.

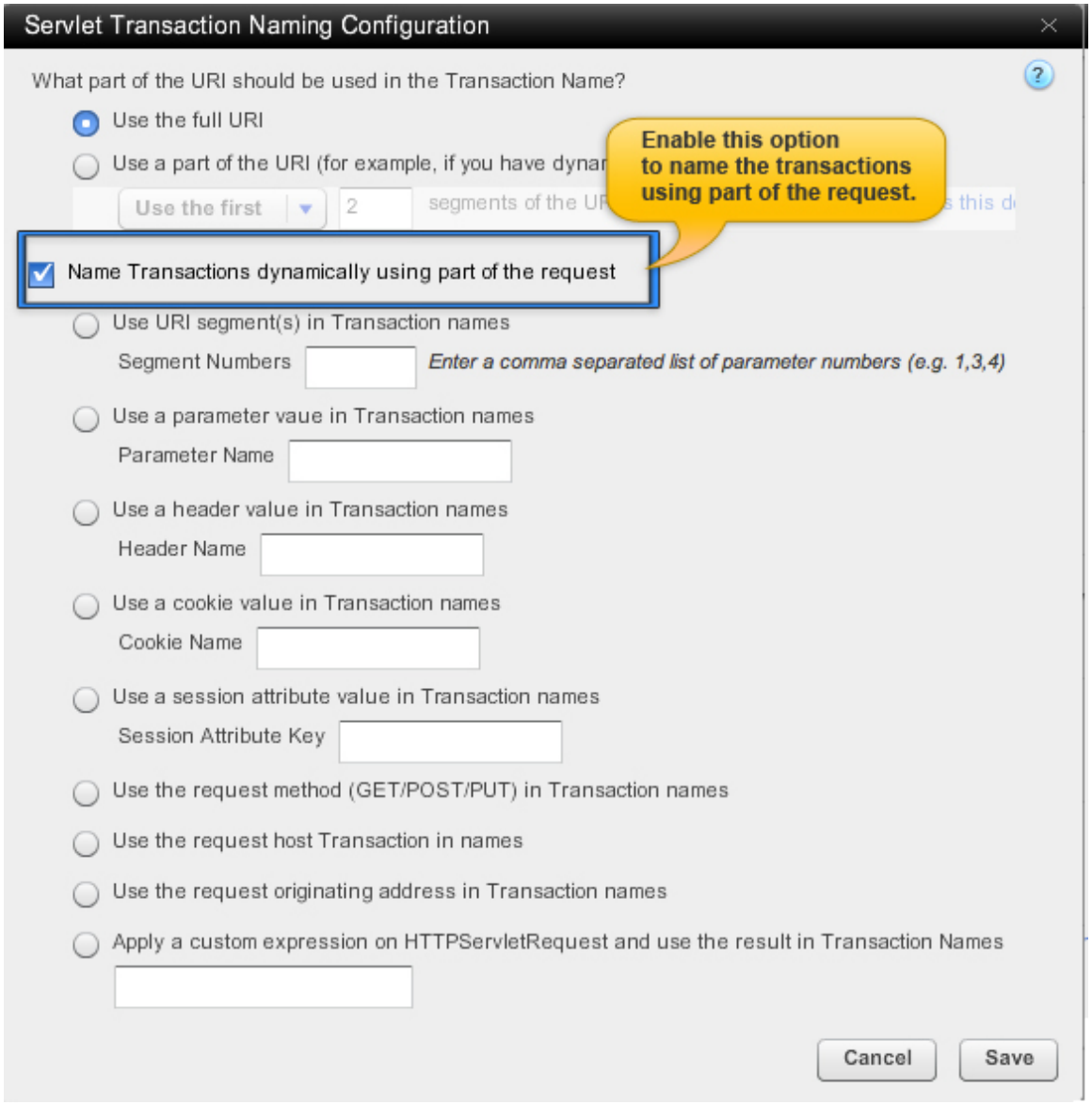


Using Headers, Cookies, and Other Parts of HTTP Requests for identifying Transactions

You can also configure the naming for your Servlet based transactions using headers, cookies, and other parts of HTTP requests.



To identify all your Servlet based transactions using particular parts of the HTTP request., use the Name Transactions dynamically using part of the request option:



Alternatively, you can also configure detection for only specific transactions using a custom match rule.

Naming Servlet Based Transactions when Different Web Contexts Require Different Naming Strategies

In certain situations, it is ideal to modify the existing naming (which is based on the URI patterns) and name your transactions based on different web contexts. You can also configure the transaction naming to directly map the WAR files deployed on the same tier.

For example, ACME Online's entry point tier has multiple contexts deployed on a single JVM, as listed in the table below. These web contexts represent two different parts of the same business application and therefore require following different naming strategies.

Web context	Ideal naming strategy
http://acmeonline.com/store/checkout	Should use first two segments to name these requests as: **/store/checkout**
http://acmeonline.com/secure/internal/updateinventory	Should use last two segments to name these requests as: **/internal/updateinventory**

http://acmeonline.com/orders/process?type=creditcard	Should use the combination of parameter value for "type" and the last two segments to name such requests as: **/orders/process.creditcard** Such a naming strategy will ensure that the credit card orders are differentiated from other orders.

To learn more about how to configure transaction identification for these situations see Custom match rules for Servlet based requests.

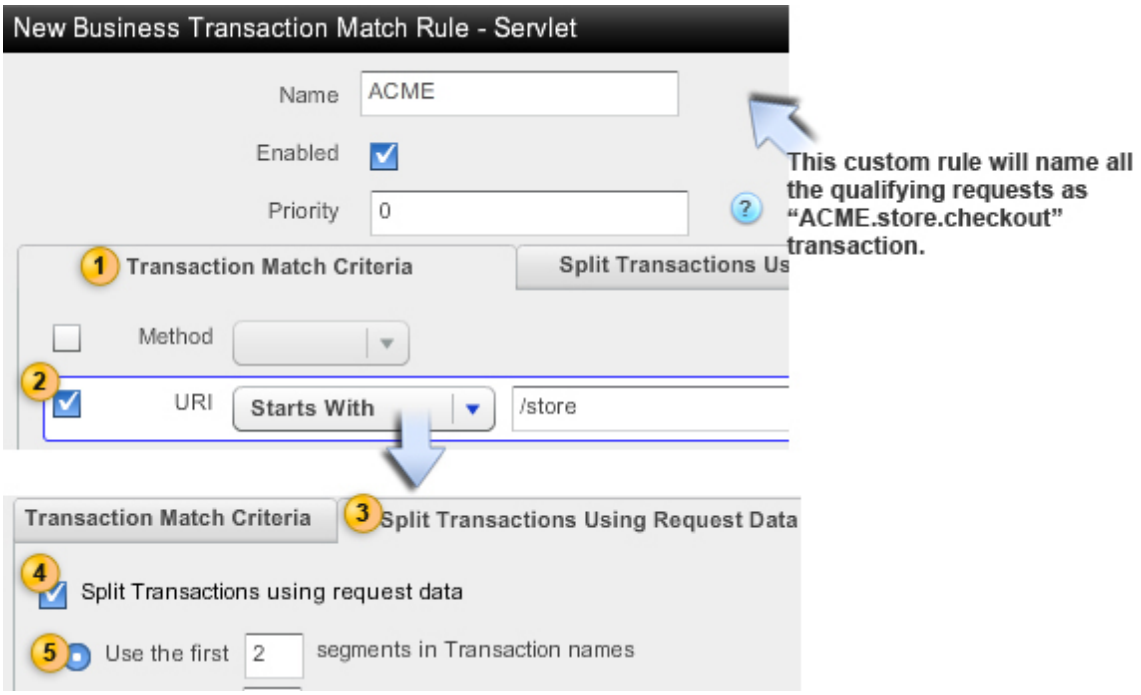
Custom Match Rules for Servlet Based Transactions

To handle situations as discussed in the previous section, use custom match rules for each of the web contexts or URIs. Custom match rules let you create mutually exclusive transaction names for specific contexts.

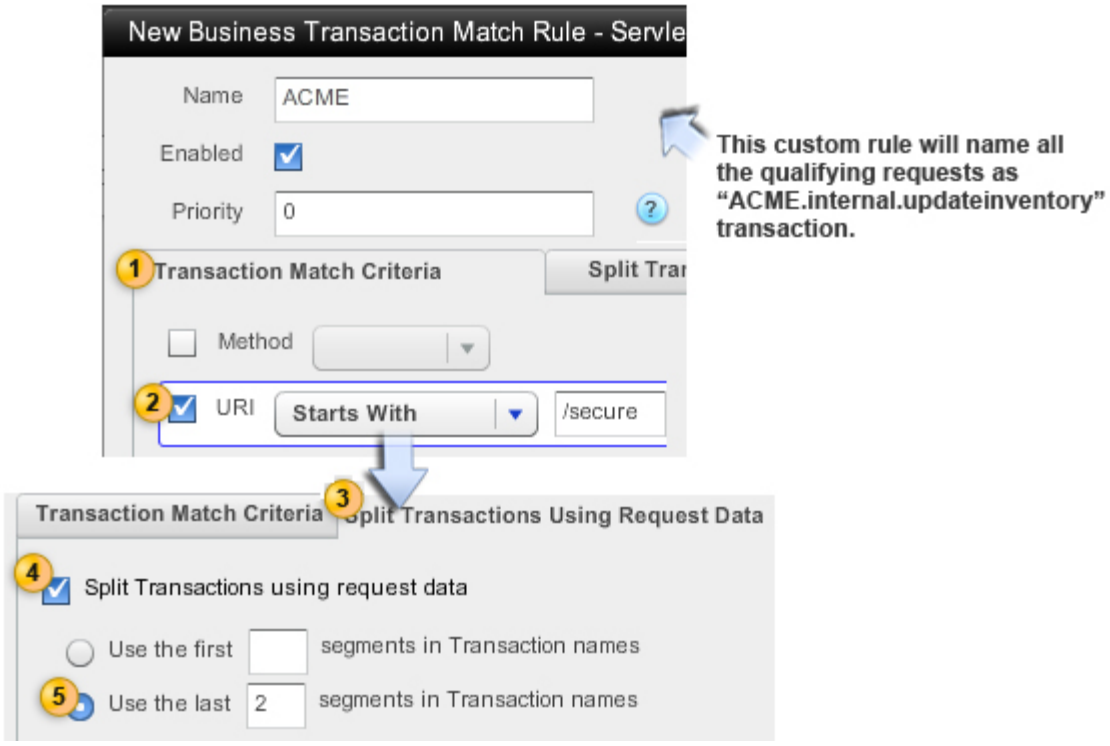
See Custom Match Rules for general information about accessing custom match rule configuration. To configure a rule for Servlets, select **Servlet** the drop-down list.

Here are some sample rules for different URLs.

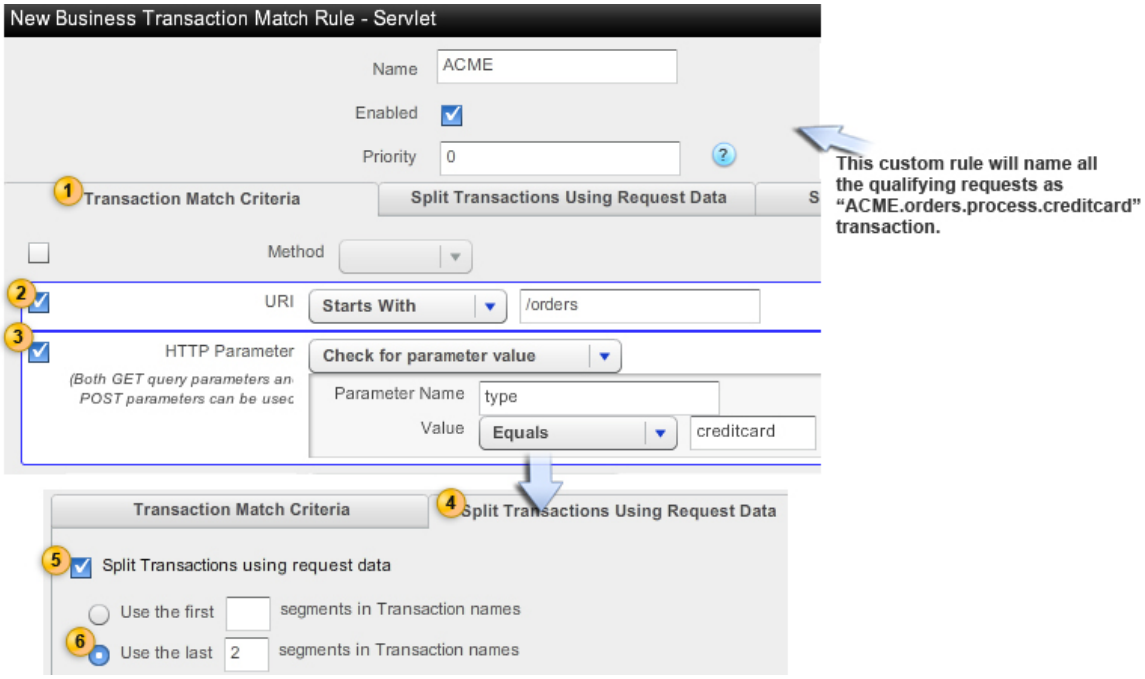
Example 1: URL is http://acmeonline.com/store/checkout



Example 2: URL is http://acmeonline.com/secure/internal/updateinventory



Example 3: URL is http://acmeonline.com/orders/process?type=creditcard



Using Custom Expressions in Custom Match Rules

You can create custom expressions to name transactions based on the evaluation of a custom expression on the HTTPServletRequestObject.

Suppose you want to monitor the HttpServletRequest request variable to identify the names and values of all the request parameters

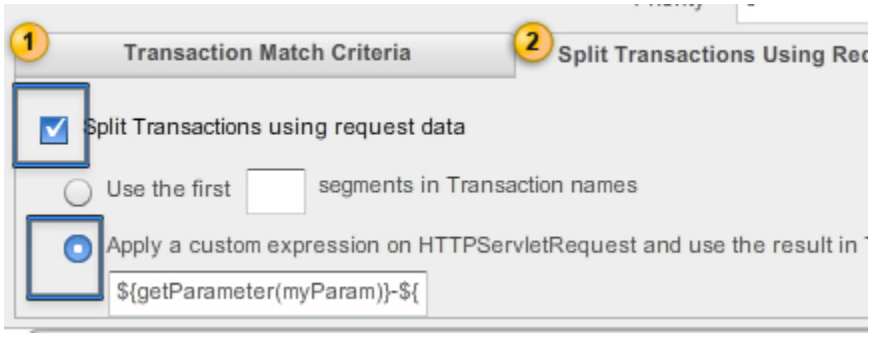
passed with the request.

The following example shows a custom rule configuration based on the expression:

${getParameter(myParam)}-${getUserPrincipal().getName()}

which evaluates to

```
request.getParameter("myParam")+"-"+request.getUserPrincipal()
```



You can create a custom expression on the HTTPServletRequest to identify all Servlet based requests (modify global discovery at the transaction naming level) or for a specific set of requests (custom rule).

Request Attributes in the Custom Expression

A custom expression can have a combination of any of the following getter chains on the request attributes:

Getters on Request Attributes	Transaction Identification
getAuthType()	Use this option to monitor secure (or insecure) communications.
getContextPath()	Use this option to identify the user requests based on the portion of the URI.
getHeader()	Identify the requests based on request headers.
getMethod()	Identify user requests invoked by a particular method.
getPathInfo()	Identify user requests based on the extra path information associated with the URL (sent by the client when the request was made).
getQueryString()	Identify the requests based on the query string contained in the request URL after the path.
getRemoteUser()	Identify the user requests based on the login of the user making this request.
getRequestedSessionId()	Identify user requests based on the session id specified by the client.
getUserPrincipal()	Identify user requests based on the current authenticated user.

For example, the following custom expression can be used to name the business transactions using the combination of header and a particular parameter:

```
${getHeader(header1)-${getParameter(myParam)}.
```

The identified transaction will be named based on the result of following expression in your application code:

```
request.getHeader("header1")+ "-"+ request.getParameter("myParam")
```


Configuring Transaction Identification for REST-Style URLs

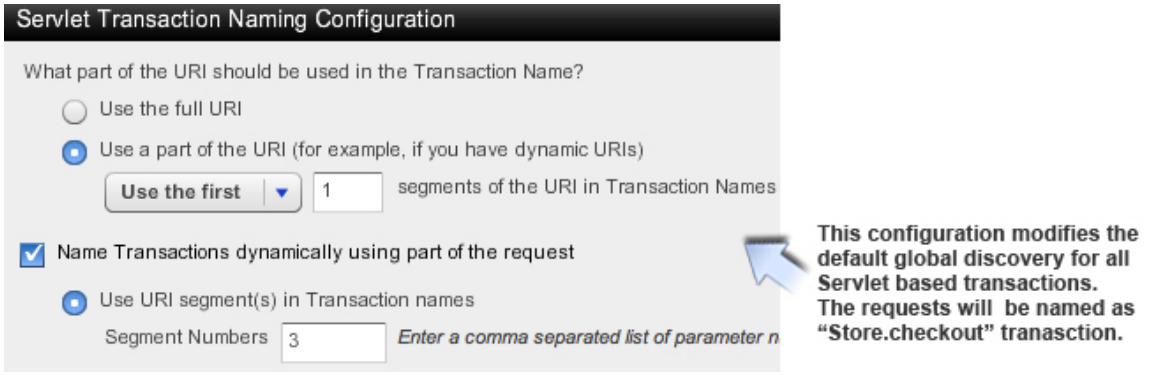
REST applications typically have dynamic URLs where the application semantics are a part of the URL.

For example, suppose customer id or the order id can be a part of the URL. The following URL represents the checkout transaction invoked by a customer with id 1234: http://acmeonline.com/store/cust1234/checkoutThe default discovery mechanism names this transaction "/store/cust1234". Ideally, all the customers performing a checkout operation should also be part of the checkout business transaction. Therefore, a better approach would be to name this transaction "/store/checkout".

To configure the transaction to be named ""/store/checkout"", use the first segment of the URL and split that URL using the third segment, thus avoiding the second (dynamic) segment.

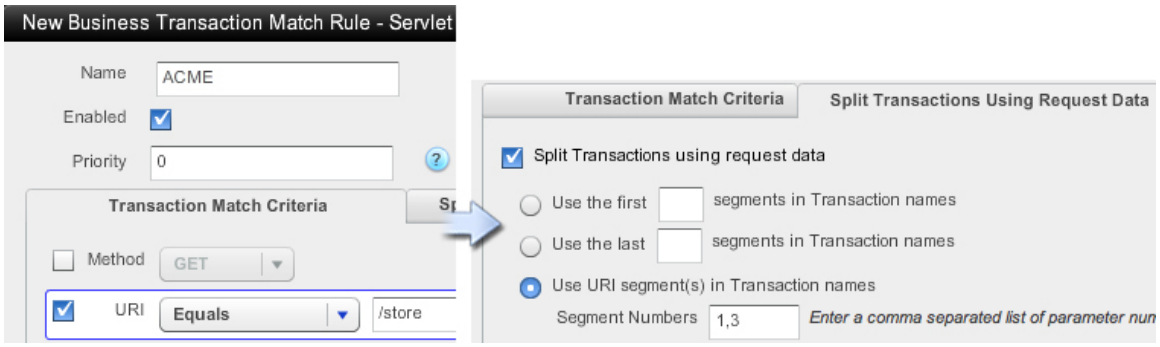
Configuration for global discovery

The following screenshot illustrates this configuration for global discovery.



Configuration for custom match rule:

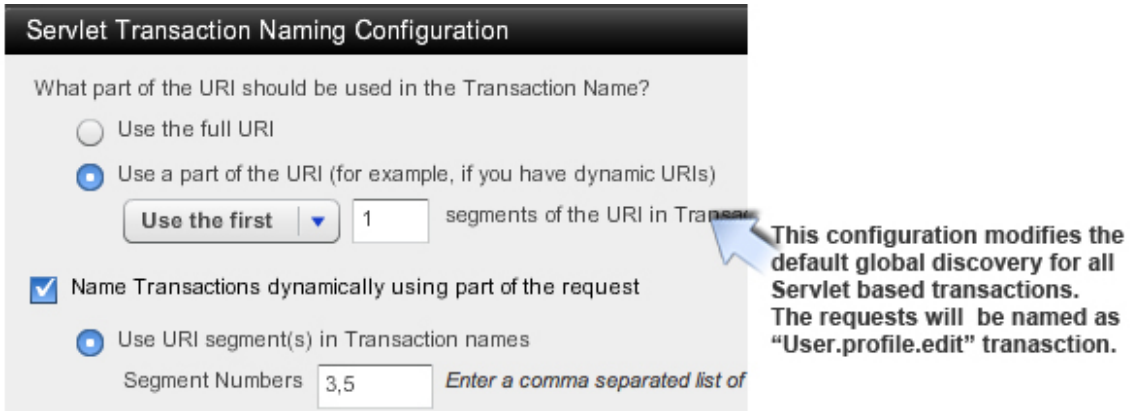
The following screenshot illustrates this configuration using a custom match rule.



Another example illustrates the rules for the following URL:http://acmeonline.com/user/foo@bar.com/profile/profile2345/editThe ideal detection strategy should avoid multiple segments (in this case, we need segments 1,3, and 5).

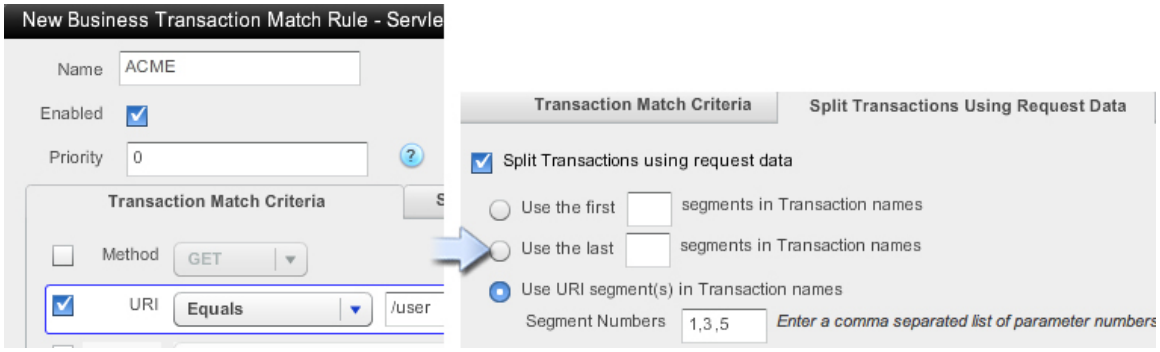
Configuration for global discovery:

The following screenshot illustrates the configuration for global discovery.



Configuration for custom match rule:

The following screenshot illustrates the configuration for creating a custom match rule.



Configuring Transaction Identification Based on Information in the Body of the Servlet Request

In certain situations, you might have an application that receives an XML/JSON payload as part of the POST request.

If the category of processing is a part of the XML/JSON, neither the URI nor the parameters/headers have enough information to name the transaction. Only the XML contents that can provide correct naming configuration.
See Identify Transactions Based on DOM Parsing Incoming XML Payload and Identify Transactions for Java XML Binding Frameworks.

Servlet Transaction Detection Scenarios

Identify Transactions Based on DOM Parsing Incoming XML Payload

- Business Transactions and XML Payload
  - Identifying the Business Transaction Using the XPath Expression
    - To configure a custom match rule
- Learn More

This topic describes how to identify transactions when an XML is posted to a Servlet.

Business Transactions and XML Payload

The XML contains the naming information for the Business Transaction. The Servlet uses a DOM parser to parse the posted XML into a DOM object.

Posted XML is parsed into a DOM object

Identifying the Business Transaction Using the XPath Expression

For example, the following XML posts an order for three items. The order uses credit card processing.

```
<acme>
    <order>
        <type>creditcard</type>
        <item>Item1</item>
        <item>Item2</item>
        <item>Item3</item>
    </order>
<acme>
```

The URL is:

```
http://acmeonline.com/store
```

The doPost method of the Servlet is:

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)

{
DocumentBuilderFactory docFactory =
DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
Document doc = docBuilder.parse(req.getInputStream());

Element  element = doc.getDocumentElement();

//read the type of order
//read all the items
processOrder(orderType,items)
.....
}
```

The XPath expression "//order/type" on this XML payload evaluates to "creditcard".

This value correctly identifies the type of the order and therefore should be used to name the "Order" transaction.

To identify the Business Transactions in this manner, first configure a custom match rule that automatically intercepts the method that parses the XML and gets the DOM object.

You use the XPath expression in the custom rule so that it names the transaction, for example "Store.order.creditcard". Though the name is not obtained until the XML is parsed, AppDynamics measures the duration of the business transaction to include the execution of the doPost() method.

To configure a custom match rule

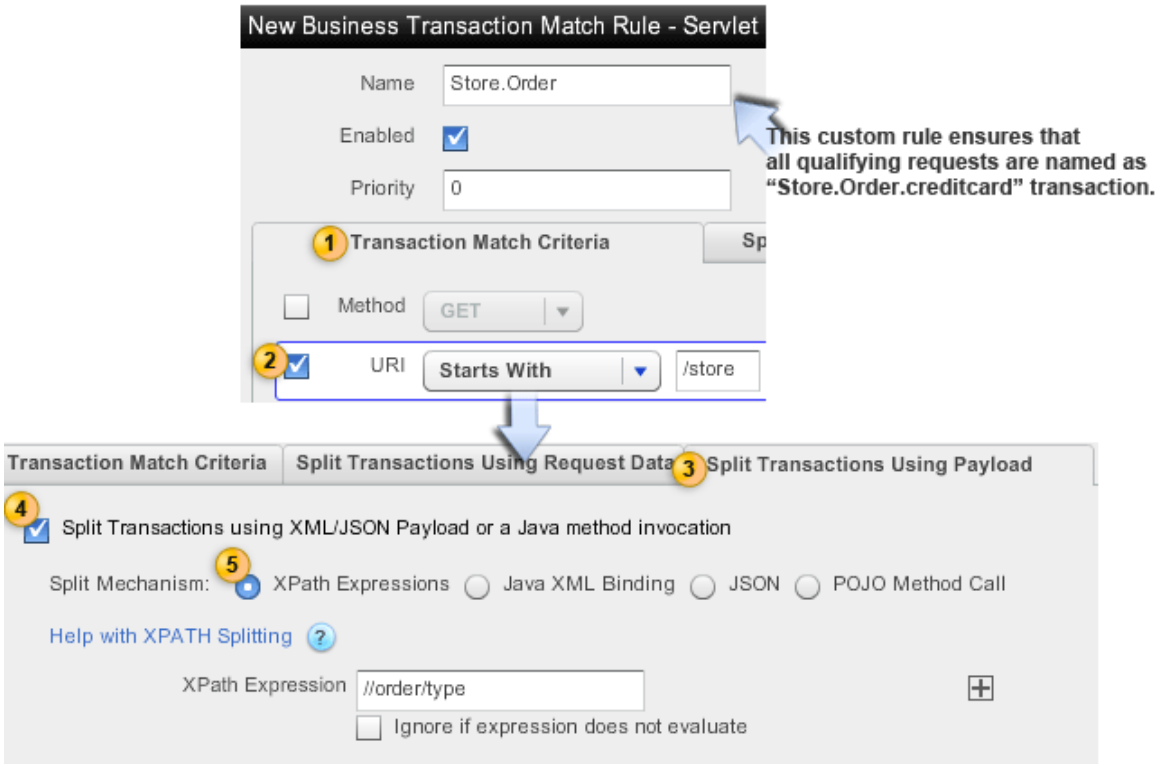
1. Navigate to the custom rule section for Servlets.

2. In the **Transaction Match Criteria** tab, specify the URI.

3. In the **Split Transactions Using Payloads** tab, enable **Split transactions using XML/JSON Payload or a Java method invocation**.

4. Set the split mechanism to **XPath Expressions**.

5. Enter the XPath expression that you want to set as the Entry Point. The result of the XPath expression will be appended to the name of the Business Transaction.



You can use one or more XPath expressions to chain the names generated for the Business Transaction.

If the expression does not evaluate to a value, the transaction will not be identified.

Learn More

- Servlet Entry Points

Identify Transactions Based on POJO Method Invoked By a Servlet

- Using a Java Method to Name a Transaction
  - To configure the custom match rule
- Learn More

Using a Java Method to Name a Transaction

You can use a Java method to name the transaction where:

- You might not have a clear URI pattern or
- You are using XML/JSON frameworks that are currently not supported by AppDynamics.

The following illustration shows how the Servlet that invokes the POJO method holds the transaction name.

For example, consider the processOrder() method. The order is parsed using any type of method and eventually when the processOrder() method is invoked, a correct approach to name transaction is to capture the first parameter to the processOrder() method.

The following URL is used by these requests: http://acmeonline.com/store. Following code snippet shows the doPost() method of the Servlet:

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)

{

//process the data from the sevlet request and get the orderType and the items
processOrder(orderType,item)
.....
}
public void processOrder(String orderType,String item)
{
//process order
}
```

The processOrder() method has the information which can correctly derive the type of the order and also the name of the transaction. To identify these requests as a single transaction, configure a custom match rule.

To configure the custom match rule

1. Go to the custom rule section for Servlet Entry Points

2. In the **Transaction Match Criteria** tab, specify the URI.

3. In the **Split Transactions Using Payload** tab, enable Split transactions using XML/JSON Payload or a Java method invocation.

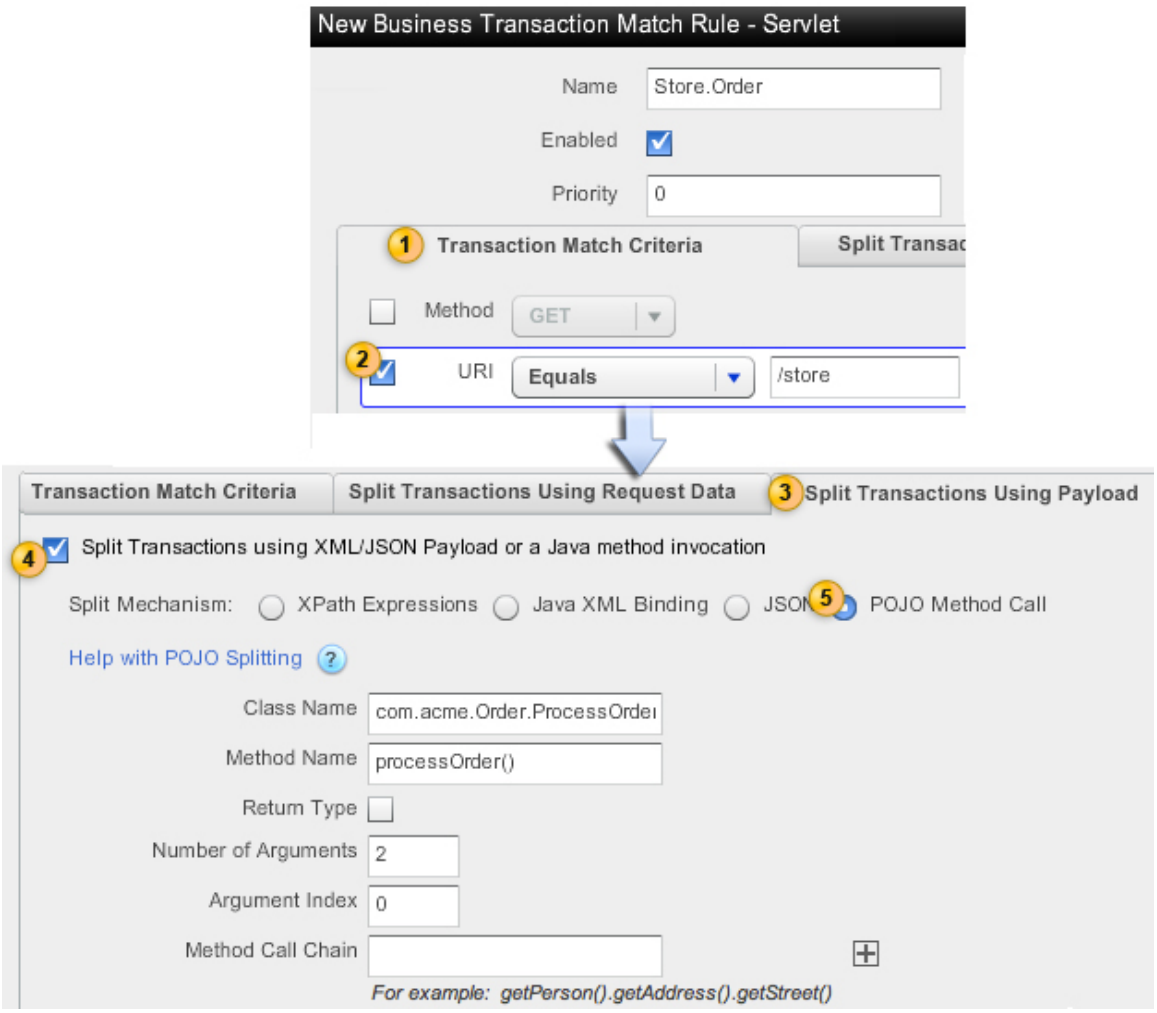
4. Select POJO Method Call as the split mechanism .

5. Enter the class and the method name.

6. If the method is overloaded, also specify the details for the arguments.

7. Optionally, you can name your transactions by defining multiple methods in a getter chain in the Method Call Chain field.

The following screenshot displays the configuration of a custom match rule which will name all the qualifying requests into a "Store.order.creditcard" transaction:



This custom rule ensures that the processOrder method is automatically intercepted.

Although the name is not obtained till the processOrder() method is called, the time for the transaction will include all of the doGet() method.

In addition to the parameter, you can also specify either the return type or a recursive getter chain on the object to name the transaction. For example, if the method parameter points to a complex object like PurchaseOrder, you can use something like getOrderDetails().getType() to correctly name the transaction.

Learn More

- Servlet Entry Points

Identify Transactions for Java XML Binding Frameworks

- To Configure the Custom Match Rule
- Supported Java XML data binding frameworks
- Learn More

The following illustration shows the situation in which an XML is posted to a Servlet. The Servlet uses an XML-to-Java binding framework, such as XMLBeans or Castor, to unmarshal and read the posted XML payload.

The XML payload contains the naming information for the transactions.

In the following example, an XML payload posts an order for three items. It uses a credit card to process the order.

The URL is: http://acmeonline.com/store

```
<acme>
   <order>
     <type>creditcard</type>
      <item>Item1</item>
      <item>Item2</item>
      <item>Item3</item>
   </order>
<acme>
```

The following code snippet shows the doPost() method of the Servlet:

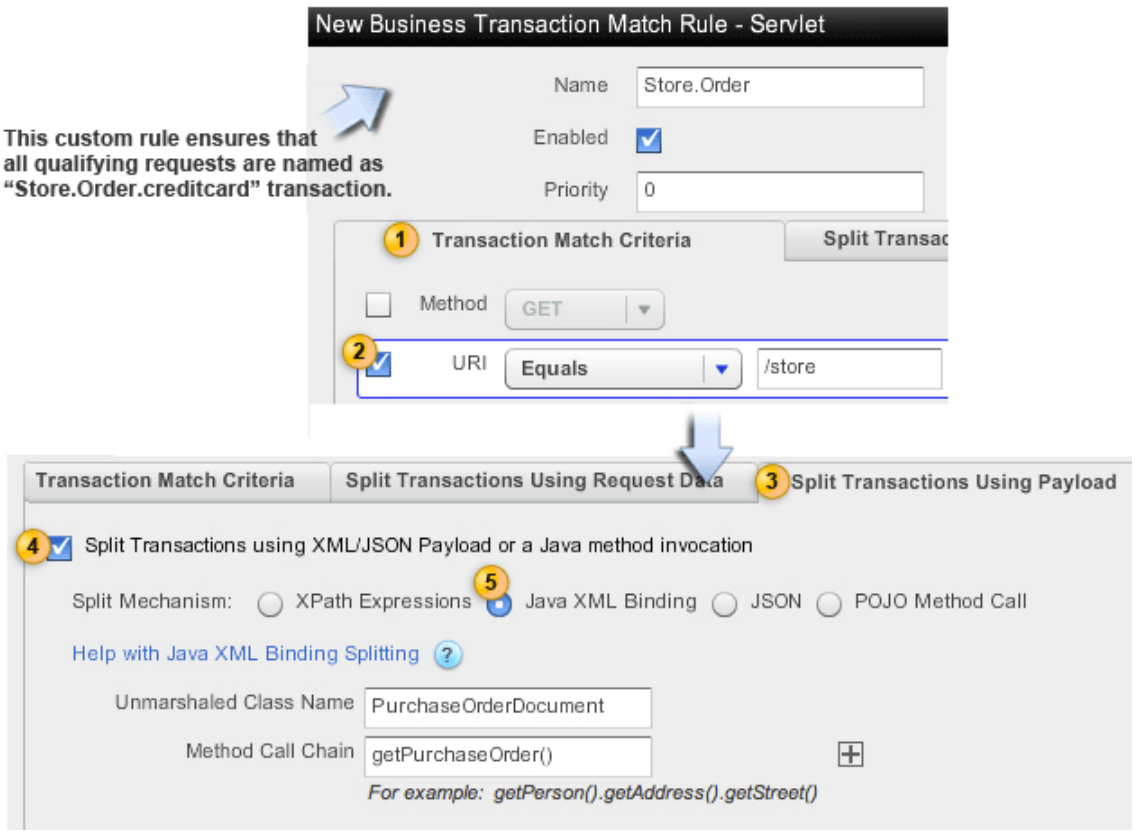
```
public void doPost(HttpServletRequest req, HttpServletResponse resp)

{
PurchaseOrderDocument poDoc = PurchaseOrderDocument.Factory.parse(po);

PurchaseOrder po = poDoc.getPurchaseOrder();
\\
String orderType = po.getOrderType();

//read all the items
processOrder(orderType,items)

...

}
```

After the posted XML is unmarshalled to the PurchaseOrder data object, the getOrderType() method should be used to identify the type of the order.

To Configure the Custom Match Rule

1. Navigate to the custom rule section for Servlet Entry Points.
2. In the **Transaction Match Criteria** tab, specify the URI.
3. In the **Split Transactions Using Payload** tab, check Split transactions using XML/JSON Payload or a Java method invocation.
4. Select Java XML Binding as the split mechanism.
5. Enter the class name and the method name.

The screenshot below shows a custom match rule which identifies the business transaction for this example as "Store.order.creditcard":



This custom rule ensures that the method in XMLBeans (which unmarshals XML to Java objects) is automatically intercepted. It also ensures that the getOrderType() method is applied on the Java data object only if it is the PurchaseOrder data object.

If the name of the transaction is not on a first level getter on the unmarshalled object, you can also use a recursive getter chain such as getOrderType().getOrder() to get the name.

Although the transaction name is not obtained until the XML is unmarshalled, the response time for the transaction is calculated from the doGet() method.

Supported Java XML data binding frameworks

The following Java XML data binding frameworks are supported:

- Castor
- JAXB
- JibX
- XMLBeans
- XStream

Learn More

- Servlet Entry Points

Identify Transactions Based on JSON Payload

- Example Configuration for a JSON Payload
- Learn More

Example Configuration for a JSON Payload

The following illustration shows a JSON payload posted to a Servlet and the Servlet unmarshalls the payload.

The JSON contains the naming information for the transactions.

For example, the following JSON payload posts an "order" for an item "car" and uses creditcard for processing the order.
The URL is:

http://acmeonline.com/store

```
order
:{
type:creditcard,
id:123,
name:Car,
price:23
}}
```

The following code snippet shows the doPost method of the Servlet:

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)

{

//create JSONObject from servlet input stream

String orderType = jsonObject.get("type");\\

//read the item for the order\\

processOrder(orderType,item)\\

.....\\

}
```

After the posted JSON payload is unmarshalled to the JSON object, the "type" key is required to identify the type of the order. In this case, this key uniquely identifies the business transaction.

To configure this rule:

1. Go to the custom rule section for Servlet Entry Points.

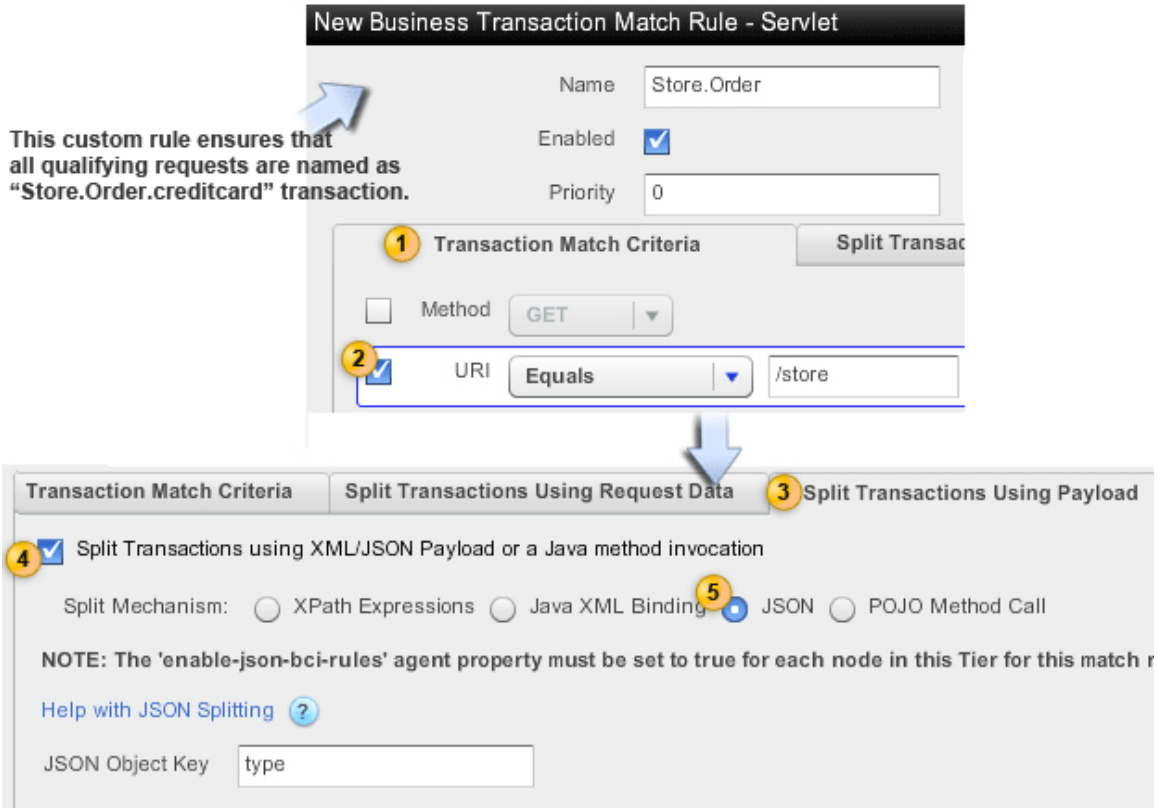
2. Under "Transaction Match Criteria" specify the URI.

3. Under "Split Transactions Using Payloads", enable "Split transactions using XML/JSON Payload or a Java method invocation".

4. Select the split mechanism as "JSON".

5. Enter the JSON object key.

The following screenshot displays the configuration for a custom match rule that will name all the qualifying requests into a single transaction called "Store.Order.creditcard".



6. Set the property "enable-json-bci-rules" to "true" for each node to enable this custom rule. See App Agent Node Properties.

This configuration ensures that the JSONObject and the get("$JSON_Object_Key") method on this object are intercepted automatically to get the name of the transaction. Although the transaction name is not obtained until the JSON object is unmarshalled, the response time for the transaction will be calculated from the doGet method.

Learn More

- Servlet Entry Points
- App Agent Node Properties

Struts Entry Points

- Struts-based Transactions
- Struts Request Names
- Custom Match Rules for Struts Transactions
- Exclude Rules for Struts Actions or Methods

Struts-based Transactions

When your application uses Struts to service user requests, AppDynamics intercepts individual Struts Action invocations and names the user requests based on the Struts action names. A Struts entry point is a Struts Action that is being invoked.

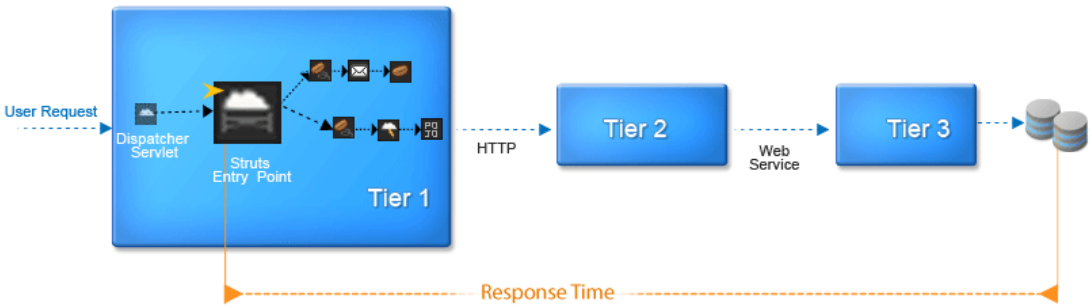
AppDynamics supports following versions of Struts:

- Struts 1.x
- Struts 2.x

Struts Action invocations are typically preceded by a dispatcher Servlet, but identification is deferred to the Struts Action. This ensures that the user requests are identified based on the Struts Action and not from the generic URL for Dispatcher Servlet.

The response time for the Struts-based transaction is measured when the Struts entry point is invoked.

The following figure shows the identification process for Struts Action entry points.



Struts Request Names

When a Struts Action is invoked, by default AppDynamics identifies the request using the name of Struts Action and the name of the method. All automatically discovered Struts-based transactions are thus named using the convention <Action Name>.<Method Name>.

For example, if an action called ViewCart is invoked with the SendItems(), the transaction is named "ViewCart.SendItems".

For Struts 1.x the method name is always "execute".

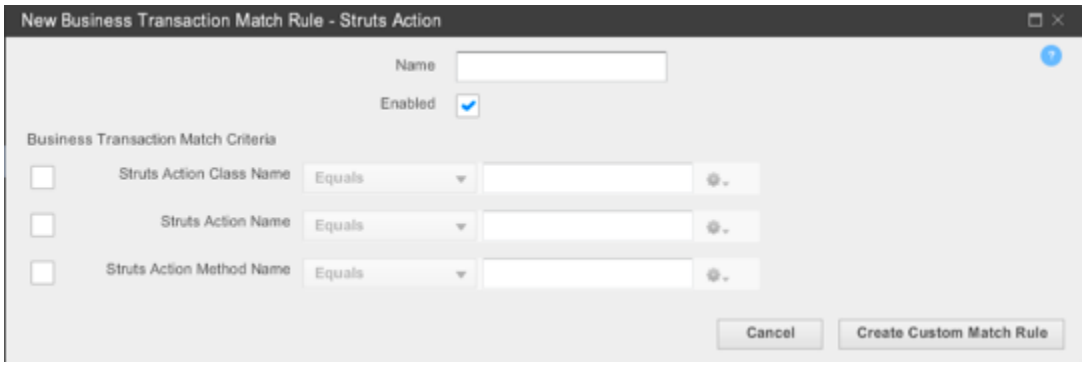
You can rename or exclude auto-discovered transactions. See Business Transaction List Operations.

Custom Match Rules for Struts Transactions

For finer control over the naming of Struts-based transactions, use custom match rules.

A custom match rule lets you specify customized names for your Struts-based requests. You can also group multiple Struts invocations into a single business transaction using custom match rules. See Custom Match Rules for information about accessing the configuration screens.

The matching criteria for creating the rule are: Struts Action class names, Struts Action names, and Struts Action method names.



Exclude Rules for Struts Actions or Methods

To prevent specific Struts Actions and methods from being monitored add an exclude rule. See Exclude Rules. The criteria for Struts exclude rules are the same as those for custom match rules.

Web Service Entry Points

- Web Services-based Transactions
  - Default Naming
  - Grouping Web Service Actions or Operation Names into a Business Transaction
  - Exclude Rules

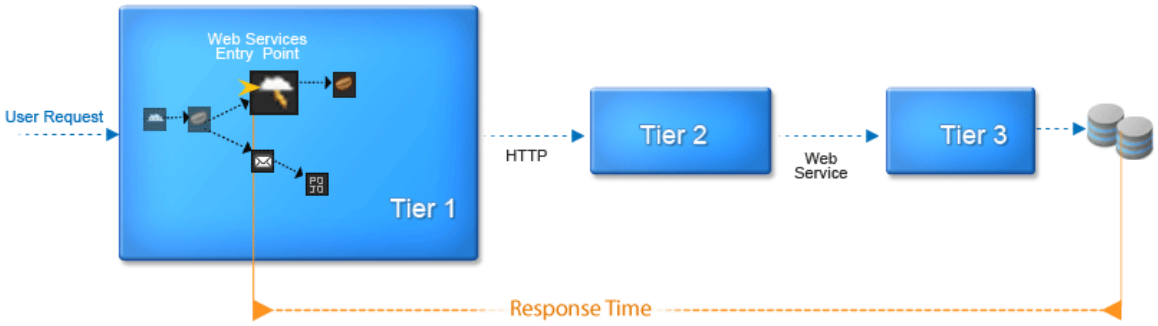
This topic discusses Web Service Entry Points.

Web Services-based Transactions

When your application uses Web Services to service user requests, AppDynamics intercepts the Web Service invocations and names requests based on the Web Service action names and operation name. A Web Service entry point is a Web Service end point that is being invoked.

This is relevant only when the Web Service invocation is part of the entry point tier and not in a downstream tier.

Web Service invocations are usually preceded by a dispatcher Servlet, but identification is deferred to the Web Service endpoints. This configuration ensures that the requests are identified based on the Web Service and not based on the generic URL for the dispatcher Servlet.



Default Naming

When the Web Service end point is invoked, the request is named after the Web Service name and the operation name.

For example, if a service called CartService is invoked with the Checkout operation, the is named "CartService.Checkout".

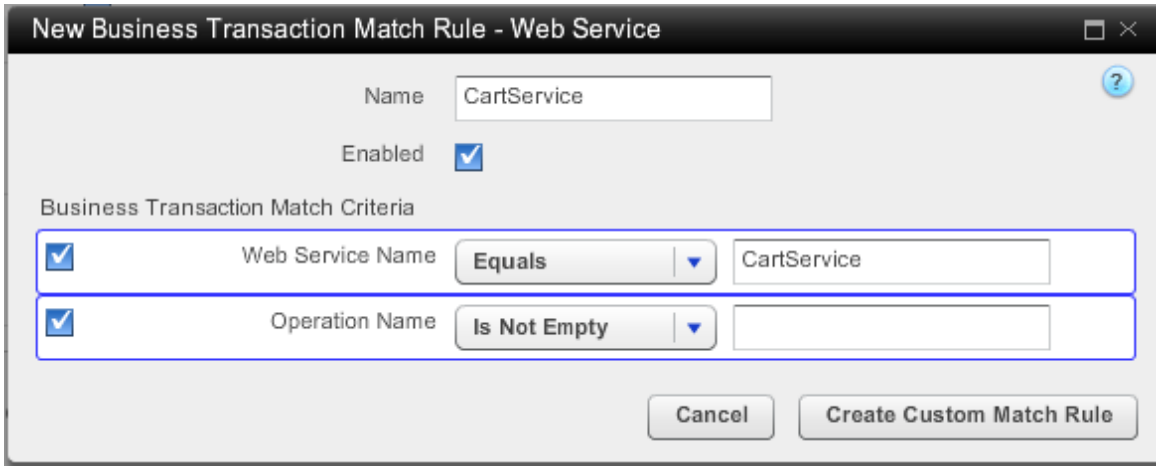
You can rename or exclude these automatically discovered transactions. See Business Transaction List Operations.

Grouping Web Service Actions or Operation Names into a Business Transaction

If you want finer control over naming of Web Service requests, you can create custom match rules for Web Services. See Custom Match Rules for information about accessing the configuration screens.

The matching criteria for Web Service rules are Web Service Name and Operation Name

The following example names all operations for the Web Service named "CartService":



Exclude Rules

To exclude specific Web Services or operation names from detection, add an exclude rule. See Exclude Rules. The criteria for Web Service exclude rules are the same as those for custom match rules.

POJO Entry Points

- Default Identification of POJO Transactions
- Recommended Practices for Defining a POJO Entry Point
- Custom Match Rules for POJO Transactions
  - POJO Transaction as a Background Task
  - Names for Custom POJO-Based Transactions
- Splitting POJO-Based Transactions
  - To configure transaction splitting
  - Using Method parameter for dynamically naming the transactions
- Exclude Rules for POJO Transactions

Not all business processing can be implemented using Web entry points for popular frameworks. Your application may perform batch processing in all types of containers. You may be using a framework that AppDynamics does not automatically detect. Or maybe you are using pure Java.

In these situations, to enable detection of your business transaction, configure a custom match rule for a POJO (Plan Old Java Object) entry point. The rule should be defined on the class/method that is the most appropriate entry point. Someone who is familiar with your application code should help make this determination. See Recommended Practices for Defining a POJO Entry Point.

AppDynamics measures performance data for POJO transactions as for any other transactions. The response time for the transaction is measured from the POJO entry point, and the remote calls are tracked the same way as remote calls for a Servlet's Service method.

Default Identification of POJO Transactions

By default, the AppDynamics discovery for POJO-based requests captures either a single or a very small group of Servlet URLs. AppDynamics identifies the POJO-based transactions automatically using the class name and method name convention: <Class Name>.<Method Name>. These transactions are displayed on the Business Transactions List.

In certain cases the POJO transactions are not identified separately either because the POJO does not implement a predefined interface or because it does not have a predefined annotation. Auto-discovery for these transactions is not possible.

In some circumstances, AppDynamics' default discovery rules for Servlets take precedence over the POJO-based transaction. For example, the ACME Online application has a Servlet-handled checkout operation with URI pattern: /cart. The Servlet handling the operation reads the HTTP POST parameters to determine if this is a checkout operation. With this type of user request, the Servlet dispatcher parses the request data to determine the requested operation and then dispatches the request to the POJO for handling.

Recommended Practices for Defining a POJO Entry Point

The POJO entry point is the Java method that starts the transaction.

The most important consideration in defining a POJO entry point is to choose a method that begins and ends every time the specific business transaction is invoked.

For example, consider the method execution sequence:

```
com.foo.threadpool.WorkerThread.run()
    calls com.foo.threadpool.WorkerThread.runInternal()
        calls com.foo.Job.run()
```

The first two calls to run() method are the blocking methods that accept a job and invoke it.

The Job.run() method is the actual unit of work, because Job is executed every time the business transaction is invoked and finishes when the business transaction finishes.

Methods like these are the best candidates for POJO entry points.

Custom Match Rules for POJO Transactions

If you are not getting the required visibility with auto-discovered transactions, create a custom match rule for a POJO transaction. See Custom Match Rules.

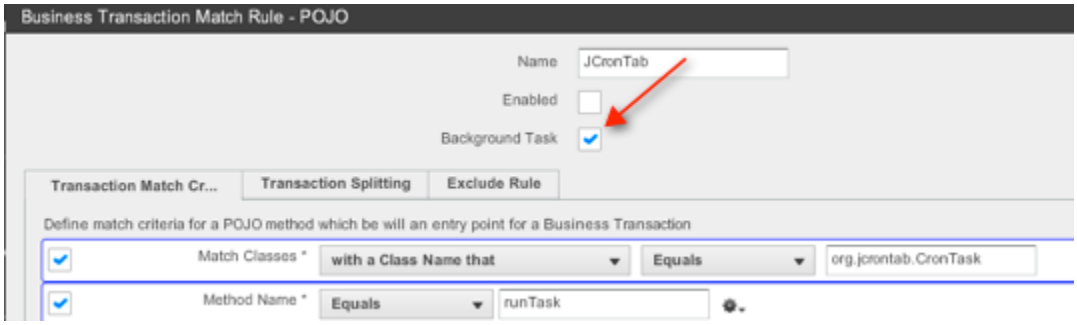
Creating a POJO transaction at a minimum involves setting the entry point.

You may optionally refine the naming of a POJO-based transaction by transaction splitting.

If you are running on IBM JVM v1.5 or v1.6, you must restart the JVM after defining the custom rules.

POJO Transaction as a Background Task

You can specify that a POJO request run as a background task by checking the Background Task check box in the configuration.



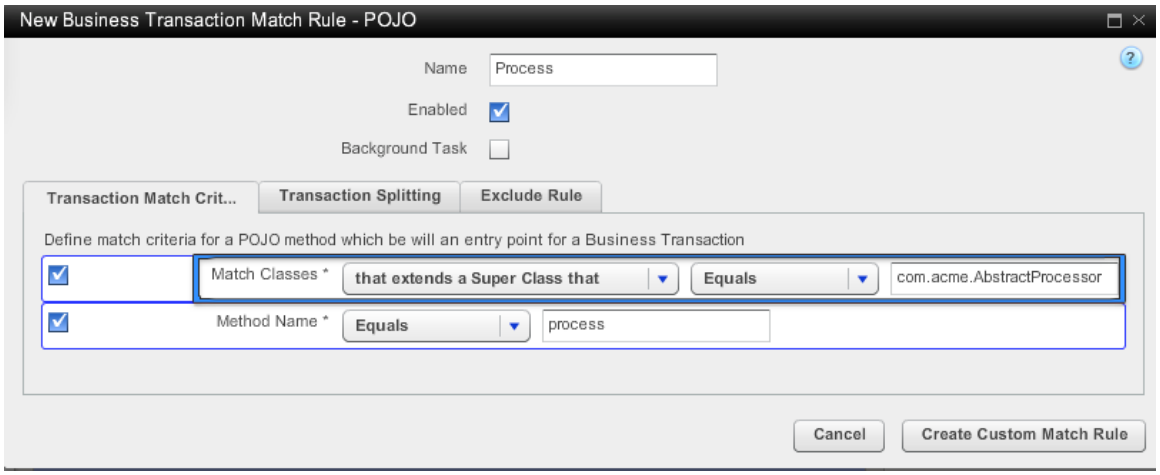
When a request runs as a background task, AppDynamics reports only Business Transaction metrics for the request. It does not aggregate response time and calls metrics at the tier and application levels for background tasks. This is ensures that background tasks do not distort the baselines for the business application. Also, you can set a separate set of thresholds for background tasks. See Background Task Monitoring

Names for Custom POJO-Based Transactions

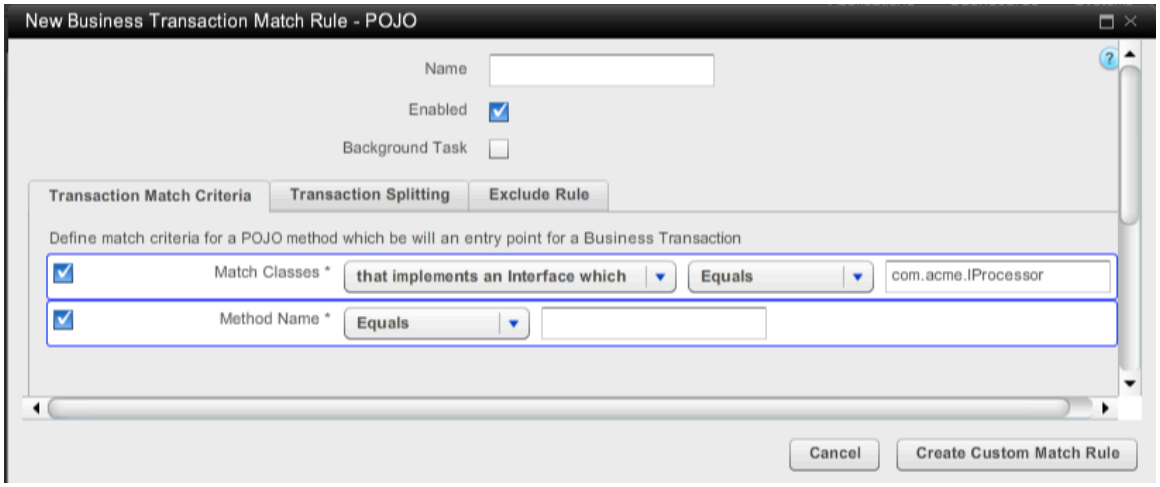
By default, when you create a custom match rule for a POJO-based transaction, the name of the transaction is based on the name of the custom rule.

For example, consider entry point based on the com.acme.AbstractProcessor superclass, which defines a process() method, which is extended by its child classes: SalesProcessor, InventoryProcessor, BacklogProcessor.

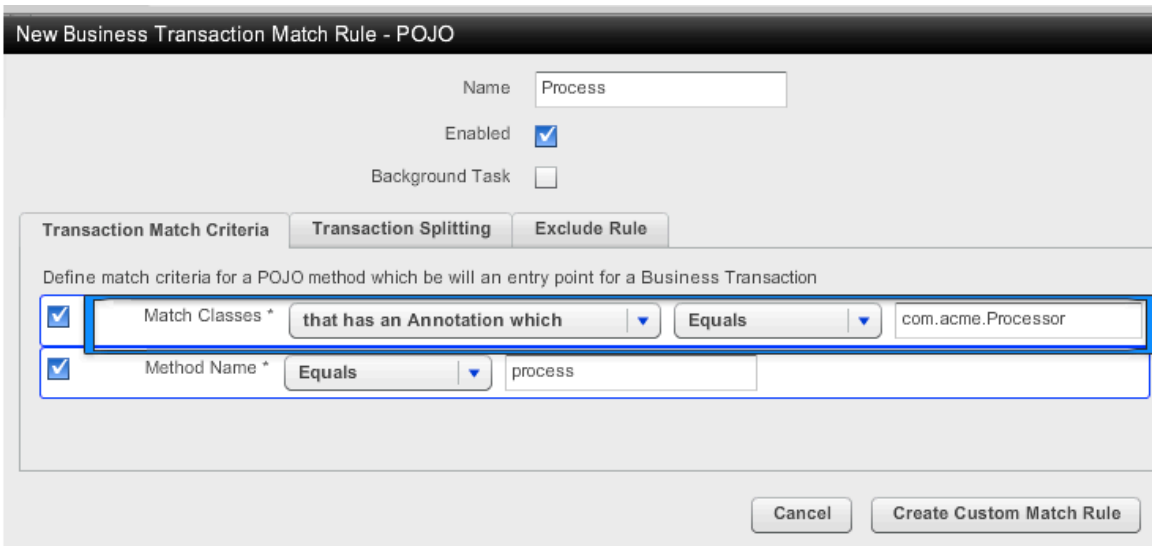
You can define the custom rule on the method defined in the superclass:



Similarly, you can define a custom rule matching on an interface named com.acme.IProcessor, which defines a process() method that is implemented by the SalesProcessor, InventoryProcessor, BacklogProcessor classes.



You can also configure a custom rule based on the Annotations.
For example, if all processor classes are annotated with "@com.acme.Processor", a custom rule should be defined using annotation.



By default, in these cases the business transaction started when any process() is invoked is named Process, based on the name of the

custom rule. To refine the transaction name to reflect the specific method called (Process.SalesProcessor, Process.InventoryProcessor, Process.BacklogProcessor) use transaction splitting.

Splitting POJO-Based Transactions

By default, when you create a custom rule for POJO transactions, all the qualifying requests are identified by the name of the custom rule.

However, in many situations it is preferable to split POJO-based transactions, especially for nodes that execute scheduled jobs.

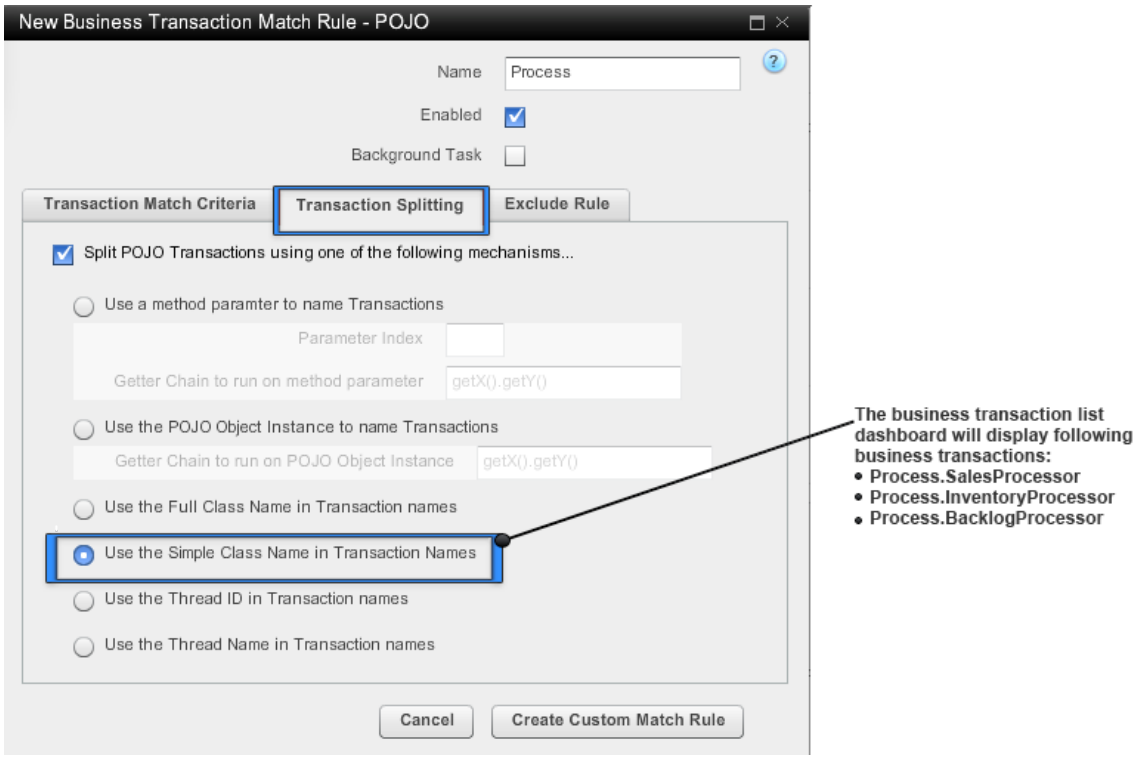
For example, if multiple classes have the same method and are instrumented using the same rule, when the method is invoked the class name of the instance being invoked can be used to classify the request.

If you split the transaction based on the simple class name, instead of one business transaction named Process, the transaction that is started when the process() method is invoked is named based on the rule name combined with the class name: either Process.SalesProcessor, Process.InventoryProcessor, or Process.BacklogProcessor.

In some cases you want to split the transaction based on the value of a parameter in the entry point method. For example, you could configure the split on the following process() method:

```
public void process(String jobType,String otherParameters...)
```

where the jobType parameter could be Sales, Inventory or Backlog.



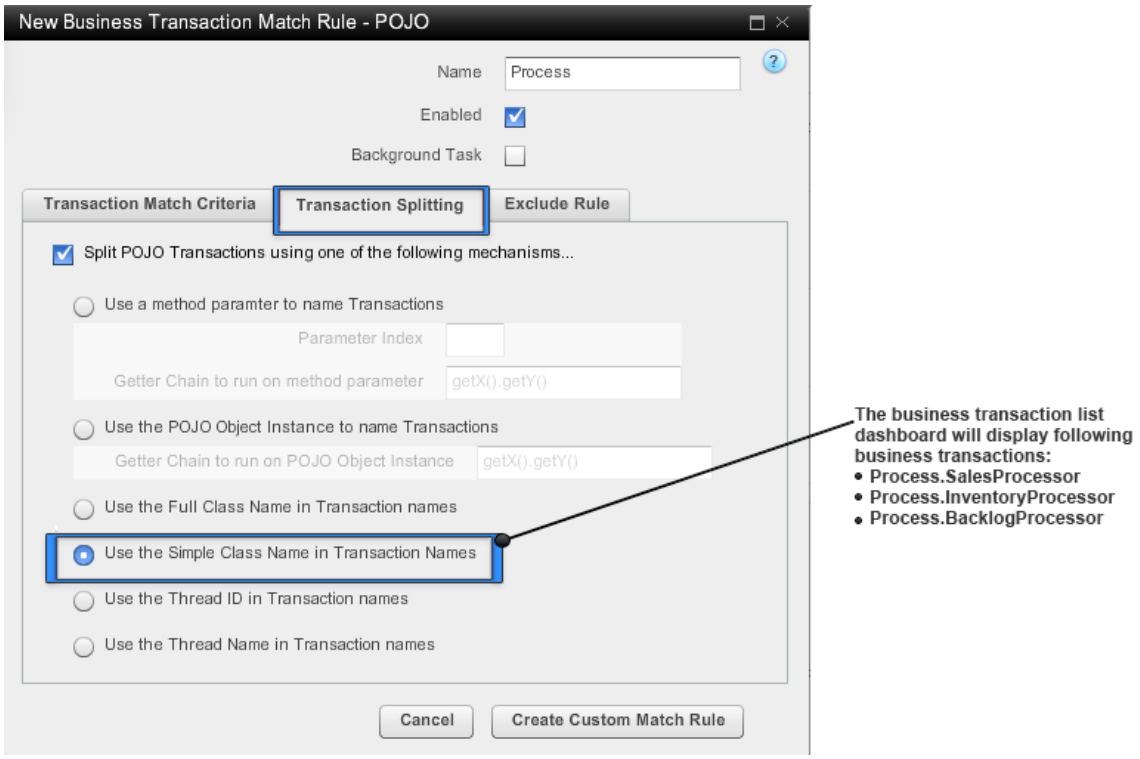
You can name POJO transactions dynamically using the following mechanisms:

- method parameter
- POJO object instance
- fully qualified class name
- simple class name
- thread ID
- thread name

In all cases, the name of the rule is prepended to the dynamically-generated name to form the business transaction name.

To configure transaction splitting

1. In the Transaction Splitting tab of the Business Transaction Match Rule - POJO window, check the Split POJO Transactions box to enable transaction splitting.
2. Select the mechanism to use to split the transaction.
If you are specifying a method parameter, enter the zero-based parameter index of the parameter
If the parameter is a complex type, specify the getter chain to use used to derive the transaction name.
If you are specifying a POJO object instance, specify the getter chain. See Getter Chains in .NET Configurations.
3. Click **Save**.



Using Method parameter for dynamically naming the transactions

Suppose in the ACME Online example, instead of the super-class or interface, the type of the processing is passed in as a parameter.

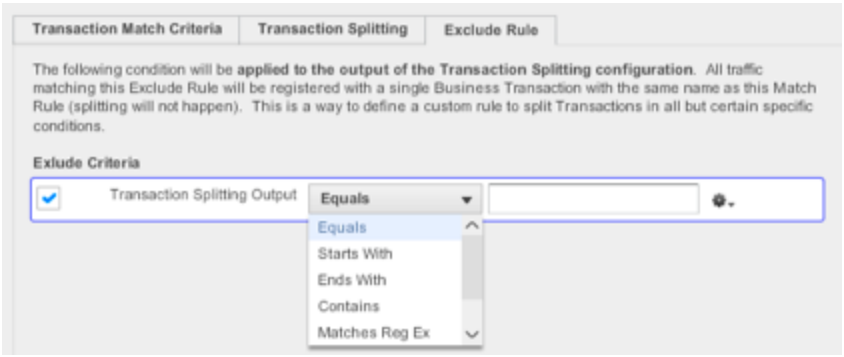
For example:

```
public void process(String jobType,String otherParameters...)
```

In this case, it would be appropriate to name the transaction based on the value of Job type. This Job type is passed as the parameter.
To specify a custom rule for method parameter:
1. Specify the details for the custom rule in the **Transaction Match Criteria** tab.
2. Click the **Transaction Splitting** tab.
3. Check the Split POJO transactions using following mechanisms checkbox.
4. Select the option for method parameter.
5. Specify the details for the parameters.
You can use a getter chain if the parameter is of complex type in order to derive a string value, which can then be used for the transaction name. See Getter Chains in .NET Configurations.

Exclude Rules for POJO Transactions

To prevent configured transaction splitting from being applied in certain situations, create an exclude rule defined on the output of the transaction splitting.



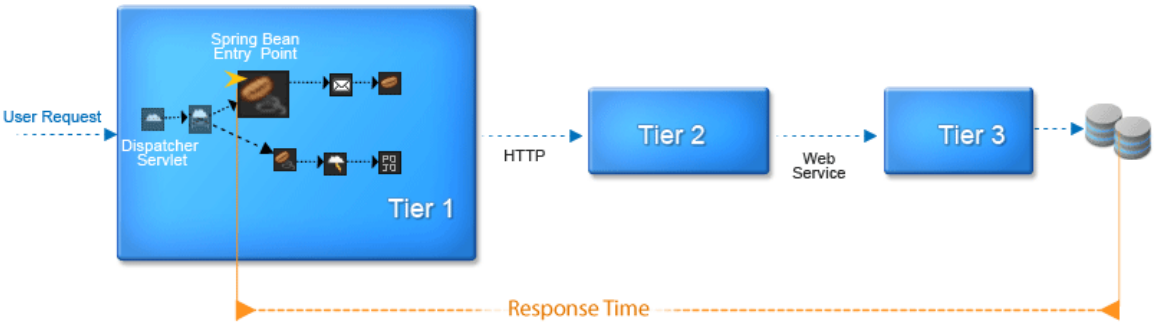
Spring Bean Entry Points

- Spring Bean-based Transactions
- Default Naming for Spring Bean Requests
  - To Enable Auto-discovery for Spring Bean entry points
- Custom Match Rules for Spring Bean Requests
- Exclude Rules Spring Bean Transactions

This topic describes how to configure transaction entry points for Spring Bean requests.

Spring Bean-based Transactions

AppDynamics allows you to configure a transaction entry point for a particular method for a particular bean in your environment. The response time is measured from when the Spring Bean entry point is invoked.



Default Naming for Spring Bean Requests

When the automatic discovery for a Spring Bean based request is turned on, AppDynamics automatically identifies all the Spring Beans based transactions and names these transactions using the following format:

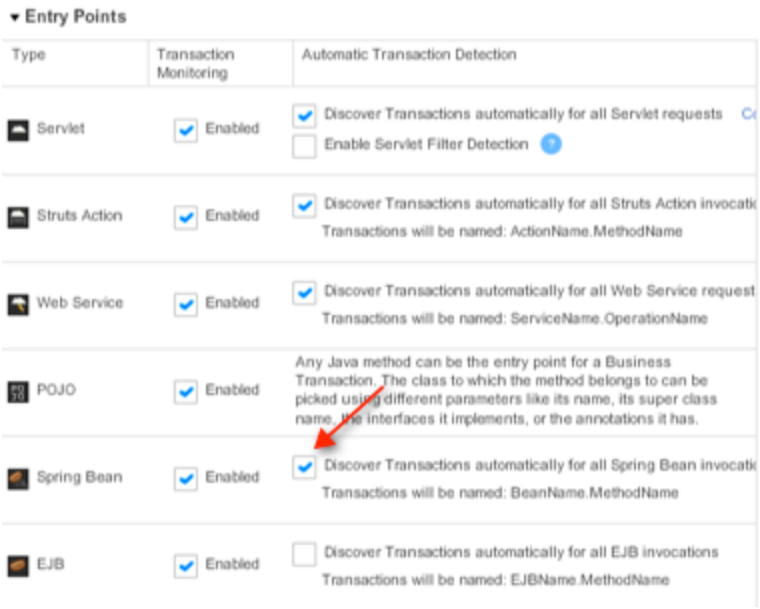
```
BeanName.MethodName
```

By default, the transaction discovery for Spring Bean-based requests is turned off.

To Enable Auto-discovery for Spring Bean entry points

1. Access the transaction detection configurations screen and select the tier to configure. See To Access Business Transaction Detection Configuration

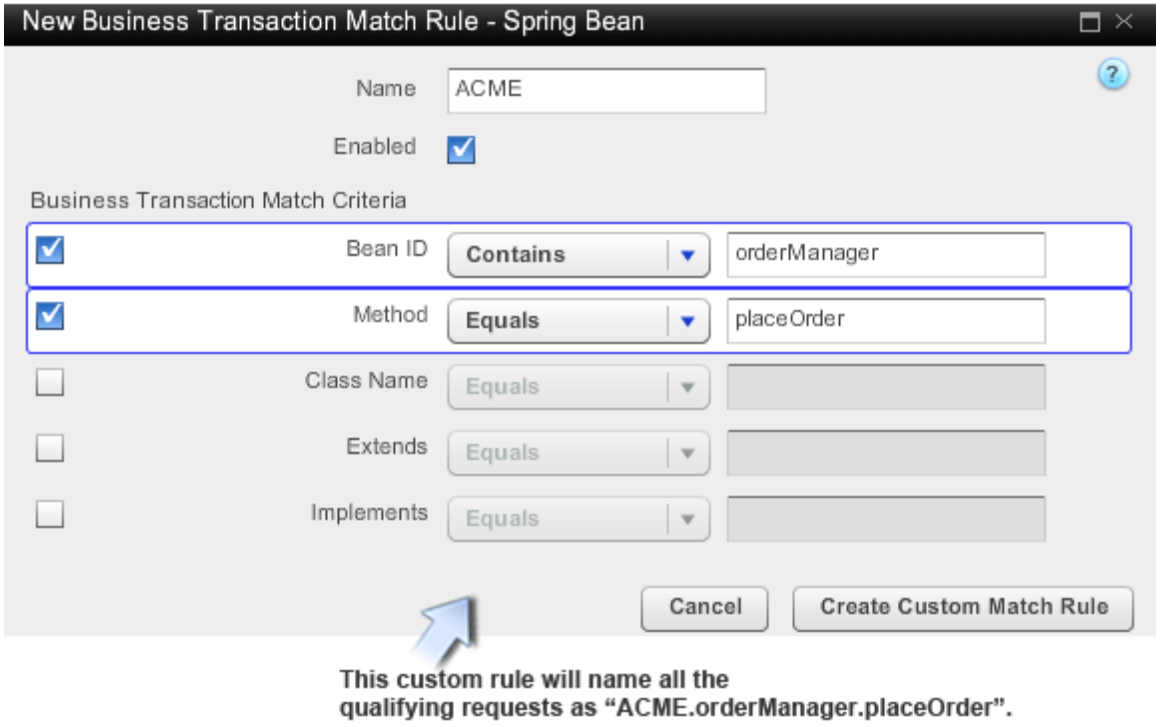
2. In the Spring Bean entry in the Entry Points section check the Automatic Transaction Detection check box.



Custom Match Rules for Spring Bean Requests

If you are not getting the required visibility with the auto-discovered transactions, you can create a custom match rule for a Spring Bean based transaction. See Custom Match Rules.

The following example creates a custom match rule for the placeOrder method in the orderManager bean.



Exclude Rules Spring Bean Transactions

To exclude specific Spring Bean transactions from detection add an exclude rule. See Exclude Rules. The criteria for Spring Bean exclude rules are the same as those for custom match rules.

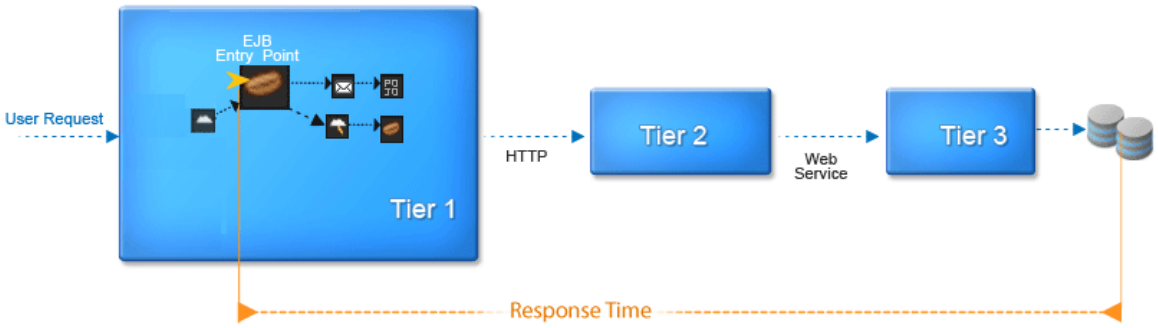
EJB Entry Points

- EJB-Based Business Transactions
- Default Naming for EJB Entry Points
    - To Enable the Auto-discovery for EJB Transactions
- Custom Match Rules for EJB based Transactions
- Exclude Rules for EJB Transactions

This topic describes how to configure transaction entry points for the EJB based requests.

EJB-Based Business Transactions

AppDynamics allows you to configure an EJB-based transaction entry point on either the bean name or method name. The response time for the EJB transaction is measured when the EJB entry point is invoked.



Default Naming for EJB Entry Points

AppDynamics automatically names all the EJB transactions <EJBName>.<MethodName>. By default, automatic transaction discovery for EJB transactions is turned off To get visibility into these transactions, enable the auto-discovery for EJB based transactions explicitly.

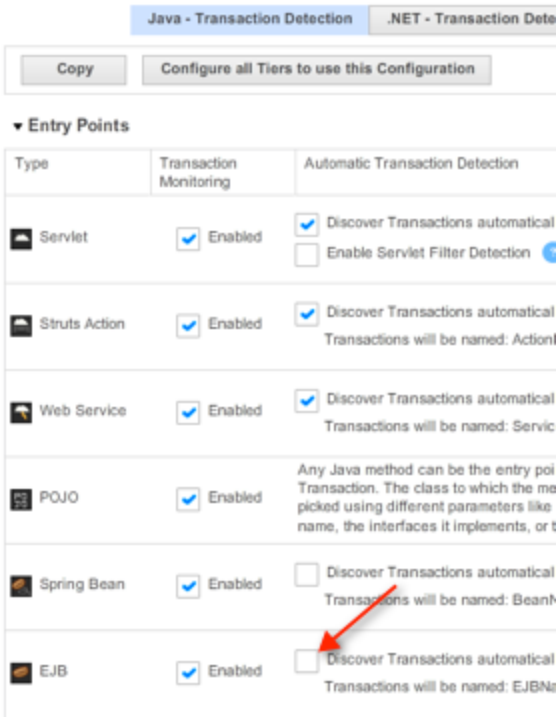
Keep in mind the following before you enable auto-discovery for EJB based transactions:

- If the EJBs use Spring Beans on the front-end, the transaction is discovered at the Spring layer and the response time is measured from the Spring Bean entry point. This is because AppDynamics supports distributed transaction correlation.
- AppDynamics groups all the participating EJB-based transactions (with remote calls) in the same business transaction. However, if your EJBs are invoked from a remote client where the App Server Agent is not deployed, these EJBs are discovered as new business transactions.

To Enable the Auto-discovery for EJB Transactions

1. Access the transaction detection configurations screen and select the tier to configure. See To Access Business Transaction Detection Configuration.

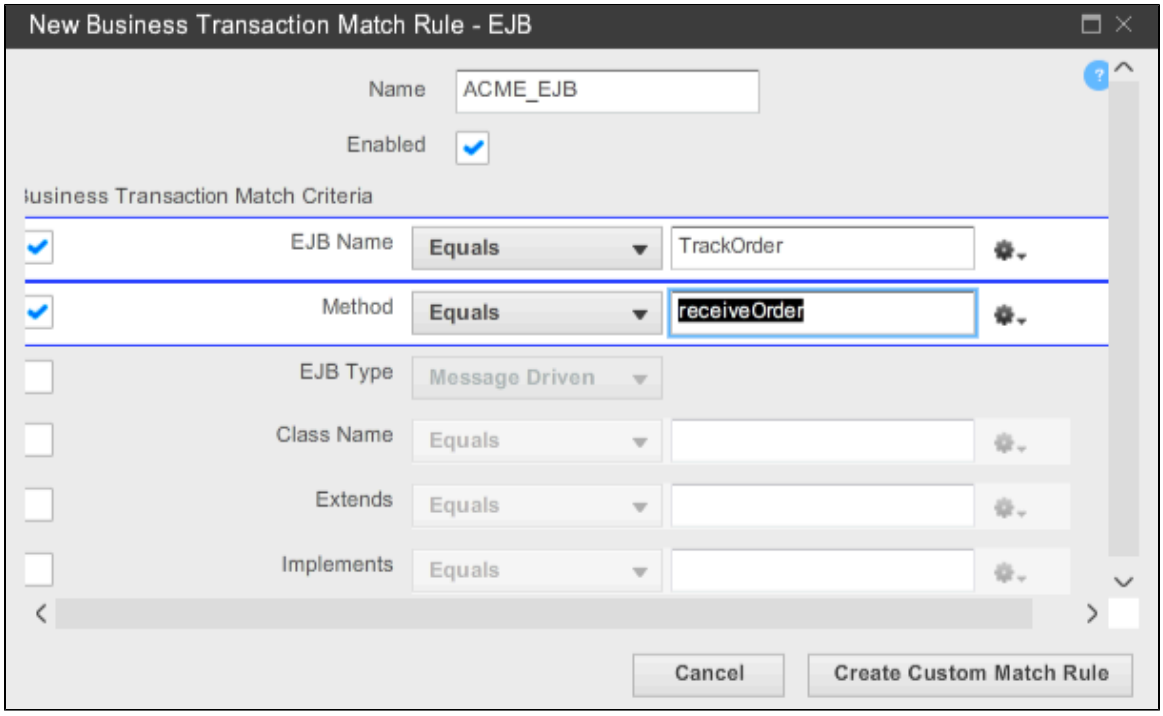
2. In the EJB entry in the Entry Points section check the Automatic Transaction Detection check box.



Custom Match Rules for EJB based Transactions

If you are not getting the required visibility with auto-discovered transactions, create a custom match rule for a EJB based transaction. See Custom Match Rules.

The following example creates a custom match rule for the receiveOrder method in the TrackOrder bean. The transactions are named " ACME_EJB.TrackOrder.receiveOrder".



In addition to the bean and method names, other match criteria that could be used to define the transaction are the EJB type, class

name, superclass name and interface name.

Exclude Rules for EJB Transactions

To exclude specific EJB transactions from detection add an exclude rule. See Exclude Rules. The criteria for EJB exclude rules are the same as those for custom match rules.

POCO Entry Points

- Default Discovery of POCO Transactions
- Defining a POCO Entry Point
    - Example POCO Entry Point
        - To specify a POCO entry point
    - POCO Transaction as a Background Task
- Further Refining POCO-Based Transactions using "Splitting"
        - To configure transaction splitting
        - To configure exclude rules

Some business processing is not implemented using common or popular frameworks. Your application may perform processing in all types of containers. It may be using a framework that AppDynamics does not automatically detect. You can specify entry points using custom match rules for POCOs (Plan Old C++ Objects). Once defined, AppDynamics measures performance data for POCO transactions as for any other transactions.

Define the custom match rule on the class/method that is the most appropriate entry point for the transaction. Someone who is familiar with your application code should help make this determination.

Default Discovery of POCO Transactions

By default, the AppDynamics discovery mechanism for POCO-based requests captures ... and displays them in the Business Transactions List. AppDynamics identifies the POCO-based transactions using the class name and method name convention:

```
<Class Name>.<Method Name>.
```

When AppDynamics does not automatically discover a POCO transactions, it may be because....?... the POCO does not implement a predefined interface or because it does not have a predefined annotation...?

AppDynamics' default discovery rules for.... takes precedence over the POCO-based transaction discovery...?

Defining a POCO Entry Point

The POCO entry point is the method that starts the transaction. Choose a method that begins and ends every time the specific business transaction is invoked. You use custom match rules to enable AppDynamics to discover the transaction.

You can refine the naming of a POJO-based transaction by transaction splitting.

Example POCO Entry Point

For example, consider the following method execution sequence: REDO CODE IF APPLICABLE

```
REDO CODE IF APPLICABLE


com.foo.threadpool.WorkerThread.run()
    calls com.foo.threadpool.WorkerThread.runInternal()
        calls com.foo.Job.run()
```

The first two calls to run() method are the blocking methods that accept a job and invoke it.

The Job.run() method is the actual unit of work, because Job is executed every time the business transaction is invoked and finishes when the business transaction finishes.

To specify a POCO entry point

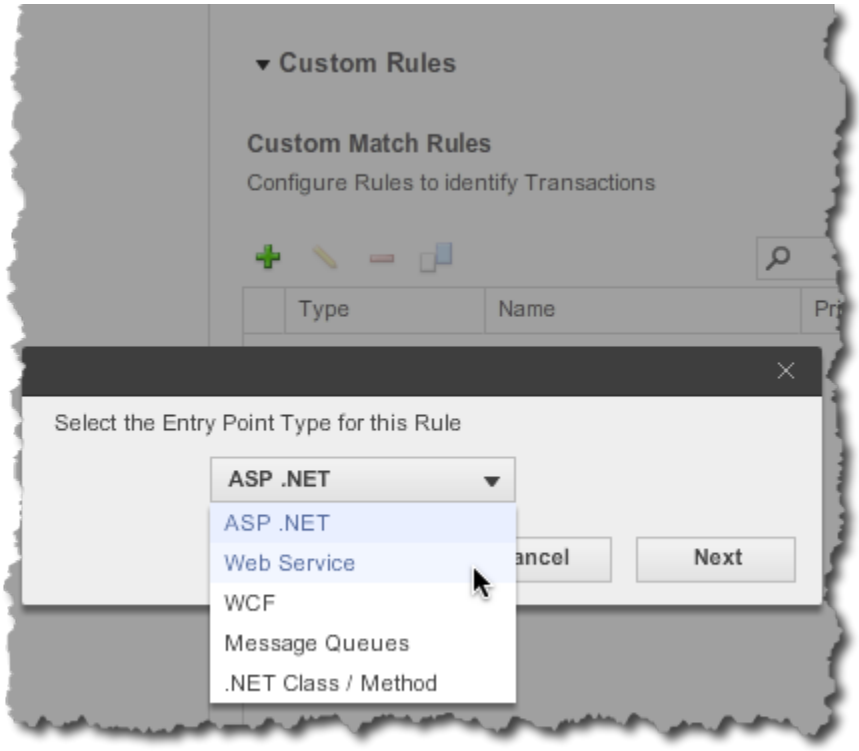
1. Click **Configure -> Instrumentation -> Transaction Detection**.

2. Click the **.NET - Transaction Detection** tab.

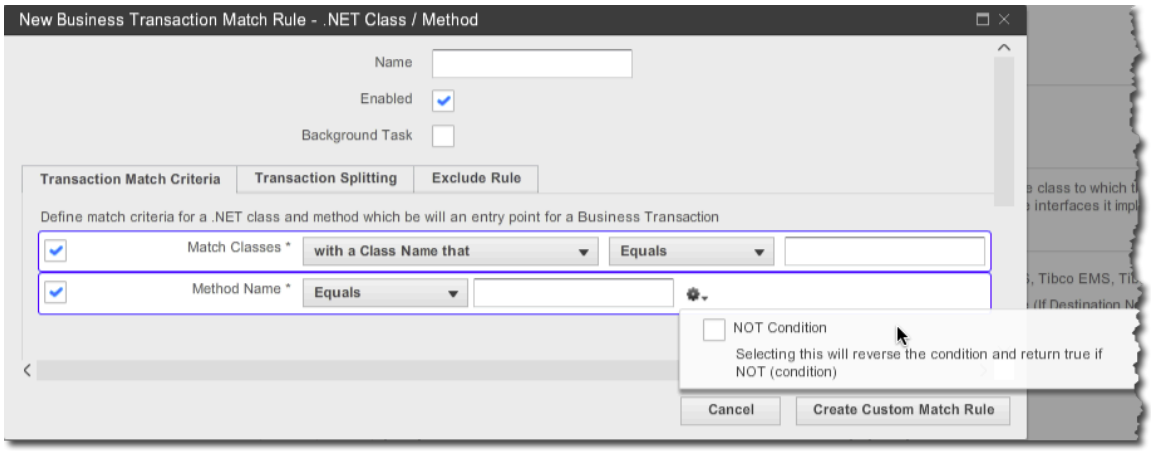
2. From the application and tier list at the left, select either:

- an application, to configure at the level of the entire business application.
- a tier, to configure at the tier level. At the tier level click **Use Custom Configuration for this Tier**. AppDynamics copies the application configuration to the tier level so that you can modify it for the tier.

3. In the Custom Rules panel under Custom Match Rules, click **Add** (the plus sign).



4. From the dropdown menu, select *.NET Class/Method].



5. Name the match rule.

- By default the rule is Enabled; you can disable it later if needed
- By default it is not a background task; see POCO Transaction as a Background Task.

6. In the Transaction Match Criteria, use the dropdowns to specify the class name:

- The type of class
- The match condition
- A string

7. If also needed, use the dropdowns to specify the method name:

- The match condition
- A string

X. If configuring an application level, click **Configure all Tiers to use this Configuration**.

POCO Transaction as a Background Task

You can indicate that a POCO transaction runs as a background task by checking the Background Task check box in the Match Rule window.

When a request runs as a background task, AppDynamics reports only Business Transaction metrics for the request. It does not aggregate response time and calls metrics at the tier and application levels for background tasks. This is ensures that background tasks do not distort the baselines for the business application. Also, you can set a separate set of thresholds for background tasks.

Further Refining POCO-Based Transactions using "Splitting"

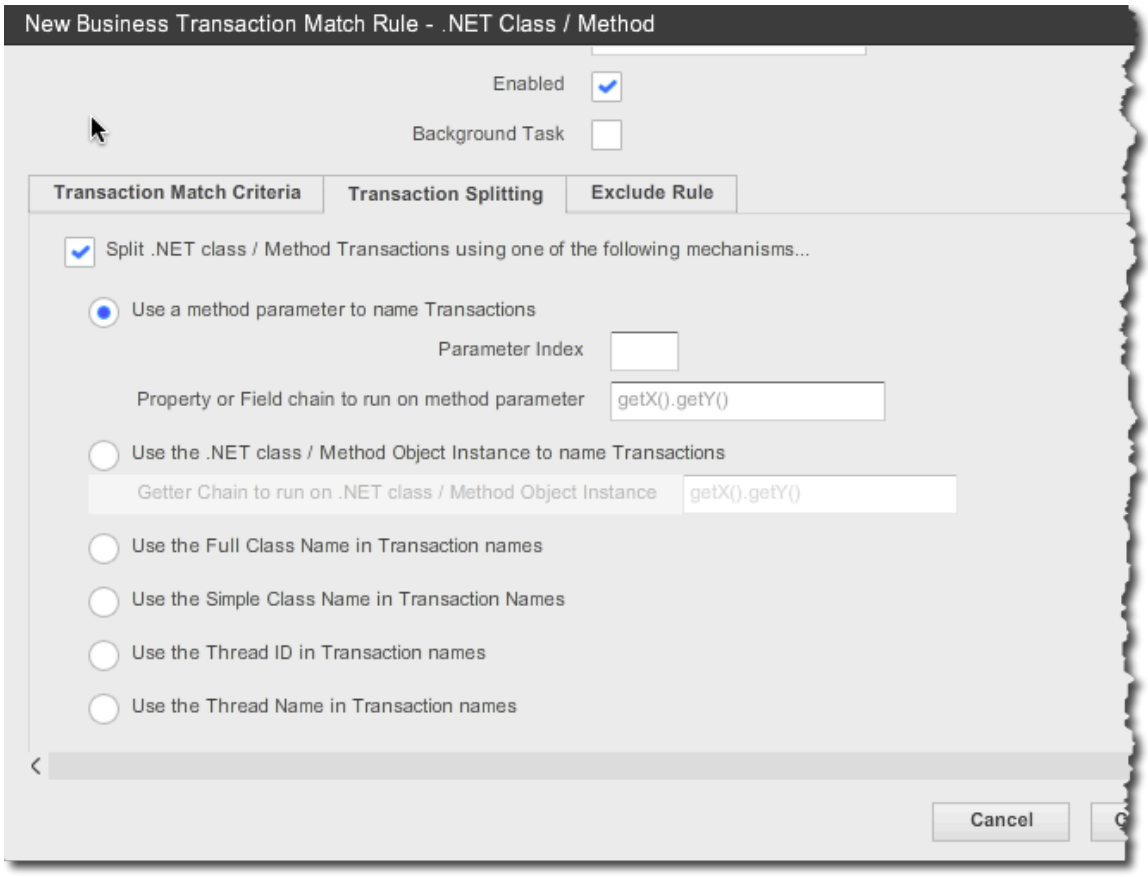
When you create a custom rule for POCO transactions all of the qualifying transactions follow the custom rule. In some situations you may need to further refine the discovery rules. For example:

- When you need to identify a transaction based on the value of a parameter in the entry point method.
- When multiple classes have the same method and are instrumented using the same rule, when the method is invoked the class name of the instance being invoked can be used to classify the request.

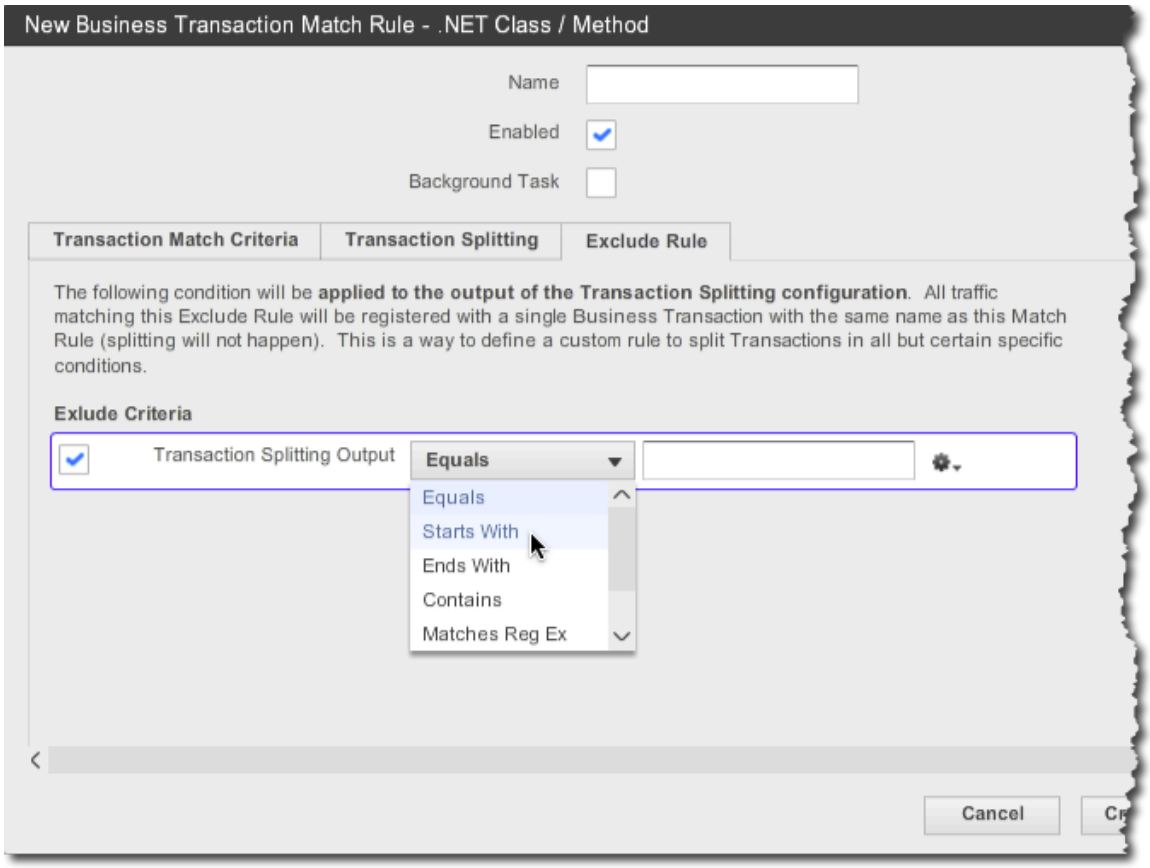
This process is called "transaction splitting".

In addition, if there are some transactions that result from the custom match and splitting rules that you want to ignore, you can specify custom exclude rules.

To configure transaction splitting



To configure exclude rules



Getter Chains in Java Configurations

- Separators in Getter Chains
- Getter Chain Examples
- Curly Braces Enclosing Getter Chains
- Learn More

This topic provides some guidance and examples of the correct syntax for getter chains in AppDynamics configurations.

Separators in Getter Chains

The following special characters are used as separators:

- comma (,) for separating parameters
- forward slash (/) for separating a type declaration from a value in a parameter
- Dot (.) for separating the methods and properties in the getter chain

If a slash or a comma character is used in a string parameter, use the backslash ()escape character.

If a literal dot is used in a string parameter, use the backslash escape character before the dot.

Getter Chain Examples

- Getter chain with integer parameters in the substring method using the forward slash as the type separator:

```
getAddress(appdynamics, sf).substring(int/0, int/10)
```


- Getter chain with various non-string parameter types:

```
getAddress(appdynamics, sf).myMethod(float/0.2, boolean/true, boolean/false, int/5)
```

- Getter chain with forward slash escaped; escape character needed here for the string parameter:

```
getUrl().split(\/) # node slash is escaped by a backward slash
```

- Getter chain with an array element:

```
getUrl().split(\/).[4]
```

- Getter chain with multiple array elements separated by commas:

```
getUrl().split(\/).[1,3]
```

- Getter chain using backslash to escape the dot in the string parameter;
  the call is getParam (a.b.c).

```
getAddress.getParam(a\.b\.c\.)
```

- In the following getter chain, the first dot requires an escape character because it is in a string method parameter (inside the parentheses). The second dot does not require an escape character because it is not in a method parameter (it is outside the parentheses).

```
getName(suze\.smith)getClass().getSimpleName()
```

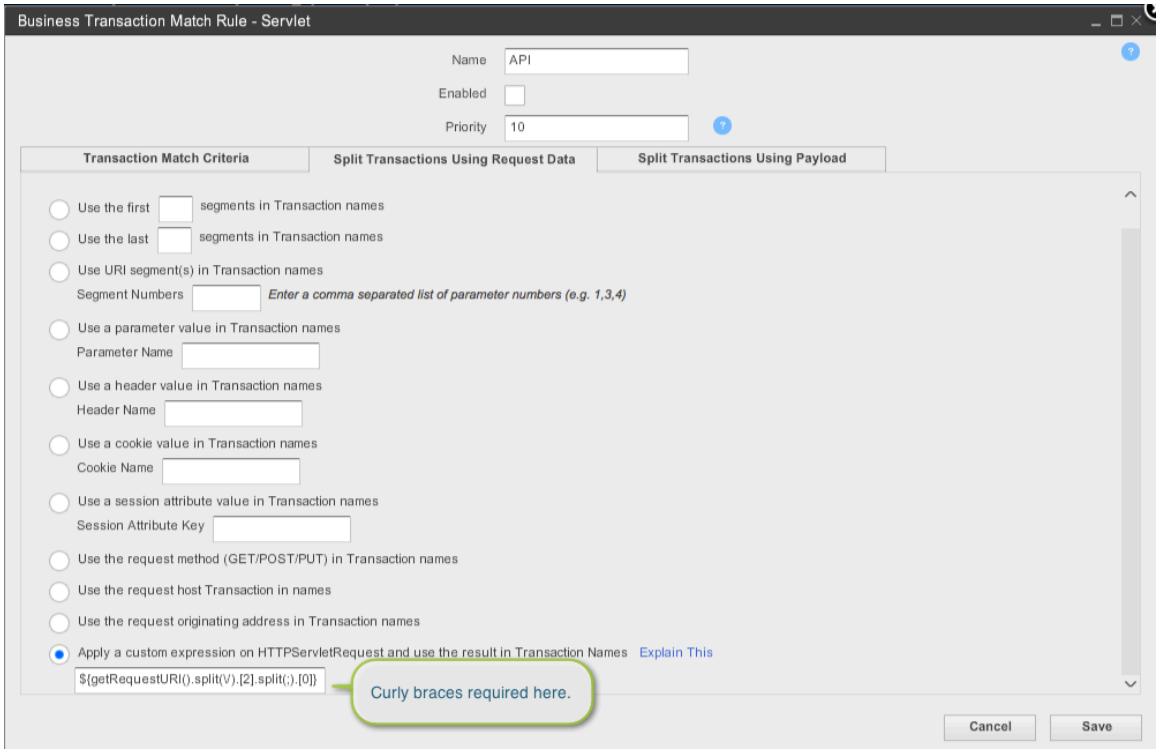
- The following getter chain is from a transaction splitting rule on URIs that use a semicolon as a delimiter; for example:
  /my-webapp/xyz;jsessionid=BE7F31CC0235C796BF8C6DF3766A1D00?act=Add&uid=c42ab7ad-48a7-4353-bb11-0dfeab
  .
  The getter chain splits on the API name, so the resulting split transactions are "API.abc", API."xyz" and so on.
  The call gets the URI using getRequestURI() and then splits it using the escaped forward slash. From the resulting array it takes the third entry (as the split treats the first slash as a separator) and inserts what before the slash (in this case, nothing) into the first entry. Then it splits this result using the semicolon, getting the first entry of the resulting array, which in this case contains the API name.

```
getRequestURI().split(\/).[2].split(;).[0]
```

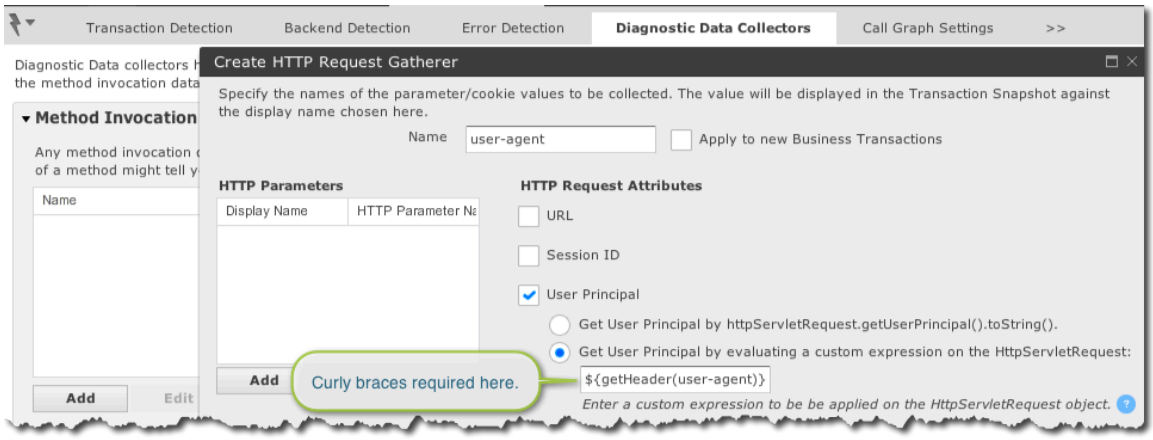
Curly Braces Enclosing Getter Chains

In most cases curly braces are not used to enclose getter chains in AppDynamics configurations. An exception is the use of a getter chain in a custom expression on the HTTPRequest object.

Custom expressions on the HTTP request are configurable in the Java Servlet Transaction Naming Configuration window and in the Split Transactions Using Request Data tab of the servlet custom match and exclude rules. In these cases, curly braces are required to delineate the boundaries of the getter chains.



Getter chains in custom expressions on the HTTP request in diagnostic data collector should also be enclosed in curly braces:



Learn More

- Configure Business Transaction Detection
- Configure Data Collectors

Troubleshoot Java Application Problems

Detect Code Deadlocks (Java)

- Code Deadlocks and their Causes
  - Detecting deadlocks using the AppDynamics UI
    - To Examine a Code Deadlock
  - Detecting deadlocks using the REST API
- Learn More

This topic describes how to use AppDynamics to detect code deadlocks.

Code Deadlocks and their Causes

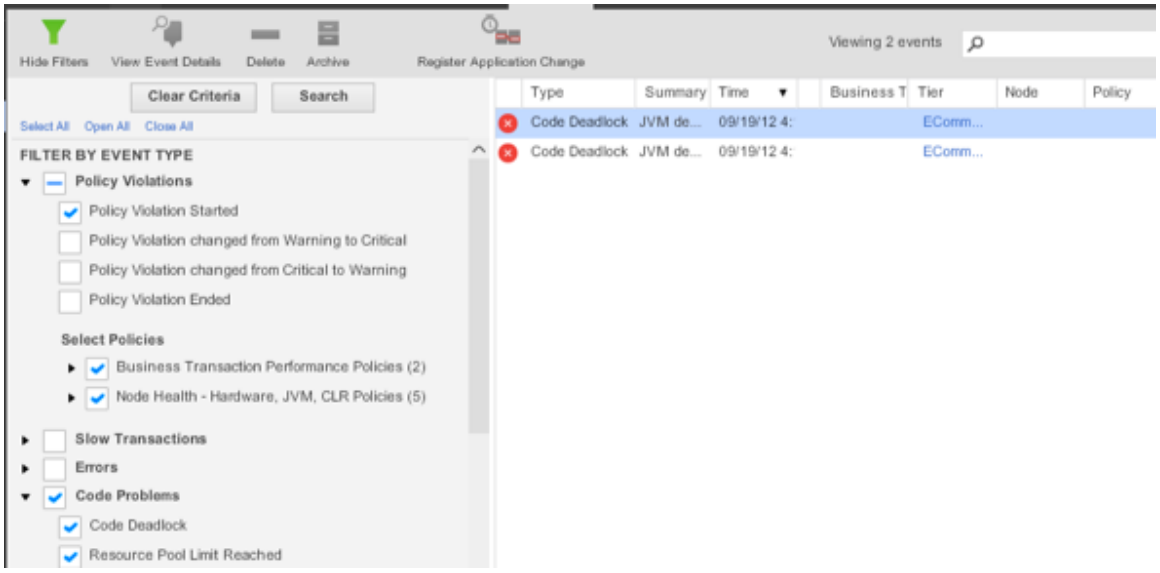
AppMan says: **Read a real-life story about how AppDynamics helped identify code deadlocks and reduce the risk to revenue!**

In a multi-threading development environment, it is common to use more than a single lock. However sometimes deadlocks will occur. Here are some possible causes:

- The order of the locks is not optimal
- The context in which they are being called (for example, from within a callback) is not correct
- Two threads may wait for each other to signal an event

Detecting deadlocks using the AppDynamics UI

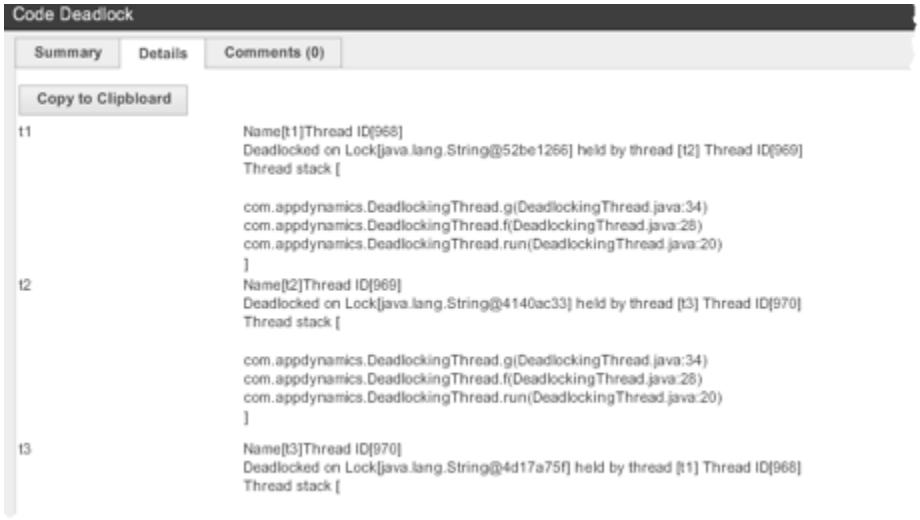
If Code Deadlock is checked in the Event configuration for an application, AppDynamics reports code deadlocks in the Events list. See Filter and Analyze Events. The following list shows two deadlocks in the ECommerce tier.



To Examine a Code Deadlock

1. Double-click the deadlock event in the events list.
The Code Deadlock **Summary** tab displays.

2. To see details about the deadlock click the **Details** tab and scroll down.



Detecting deadlocks using the REST API

You can detect a DEADLOCK event-type using the AppDynamics REST API. For details see the example Retrieve event data.

Learn More

- Use the AppDynamics REST API

Troubleshoot Java Memory Issues

Troubleshoot Java Memory Leaks

- Memory Leaks in a Java Environment
- AppDynamics Automatic Java Leak Detection
    - Prerequisites
    - Monitoring Memory for Potential JVM Leaks
- Enabling Memory Leak Detection
- Conditions for Troubleshooting Java Memory Leaks
- Workflow to troubleshoot memory leaks
- Troubleshooting Memory Leaks
    - Select the Collection Object that you want to monitor
    - Use Content Inspection
    - Use Access Tracking
- Learn More

Memory Leaks in a Java Environment

While the JVM's garbage collection greatly reduces the opportunities for memory leaks to be introduced into a codebase, it does not eliminate them completely. For example, consider a web page whose code adds the current user object to a static set. In this case, the size of the set grows over time and could eventually use up significant amounts of memory. In general, leaks occur when an application code puts objects in a static Collection and does not remove them even when they are no longer needed.

In high workload production environments if the Collection is frequently updated, it may cause the applications to crash due to insufficient memory. It could also result in system performance degradation as the operating system starts paging memory to disk.

AppDynamics Automatic Java Leak Detection

AppDynamics automatically tracks every Java Collection (for example, HashMap and ArrayList) that meets a set of criteria defined below. The Collection size is tracked and a linear regression model identifies whether the Collection is potentially leaking. You can then identify the root cause of the leak by tracking frequent accesses of the Collection over a period of time.

You can also monitor memory leaks for custom memory structures. Typically custom memory structures are used as caching solutions. In a distributed environment, caching can easily become a prime source of memory leaks. It is therefore important to manage and track memory statistics for these memory structures. To do this, you must first configure custom memory structures. For details, see Configure Custom Memory Structures (Java).

Prerequisites

For supported JVMs see Memory Monitoring Compatibility in Java Environments.

Monitoring Memory for Potential JVM Leaks

Use the Node Dashboard to identify the memory leak. A possible memory leak is indicated by a growing trend in the heap as well as the old/tenured generation memory pool.

An object is automatically marked as a potentially leaking object when it shows a positive and steep growth slope. The Memory Leak Dashboard shows:

- **Collection Size:** The number of elements in a Collection.
- **Potentially Leaking:** Potentially leaking Collections are marked as red. AppDynamics recommends that you start diagnostic sessions on potentially leaking objects.
- **Status:** Indicates if a diagnostic session has been started on an object.
- **Collection Size Trend:** A positive and steep growth slope indicates potential memory leak.

Tip: To identify long-lived Collections compare the **JVM start time** and **Object Creation Time**.

If you are not able to see any captured Collections, ensure that you have correct configuration for detecting potential memory leaks.

Enabling Memory Leak Detection

Before enabling memory leak detection, identify the potential JVMs that may have a leak.

Leak Detection mode tracks all frequently used Collections; therefore, using this mode results in a higher overhead. Turn on leak detection mode only when a memory leak problem is identified. Turn this mode off after you have identified and resolved the leak.

To achieve optimum performance, start diagnosis on an individual Collection at a time.

Ensure that you identify the potentially leaking JVMs before enabling memory leak mode.

Leak Detection mode tracks all frequently used collections; therefore, using this mode results in a higher overhead. Turn on leak detection mode only when a memory leak problem is identified. Turn this mode off after the leak has been identified and resolved.

To achieve optimum performance, start diagnosis on one individual Collection at a time.

Conditions for Troubleshooting Java Memory Leaks

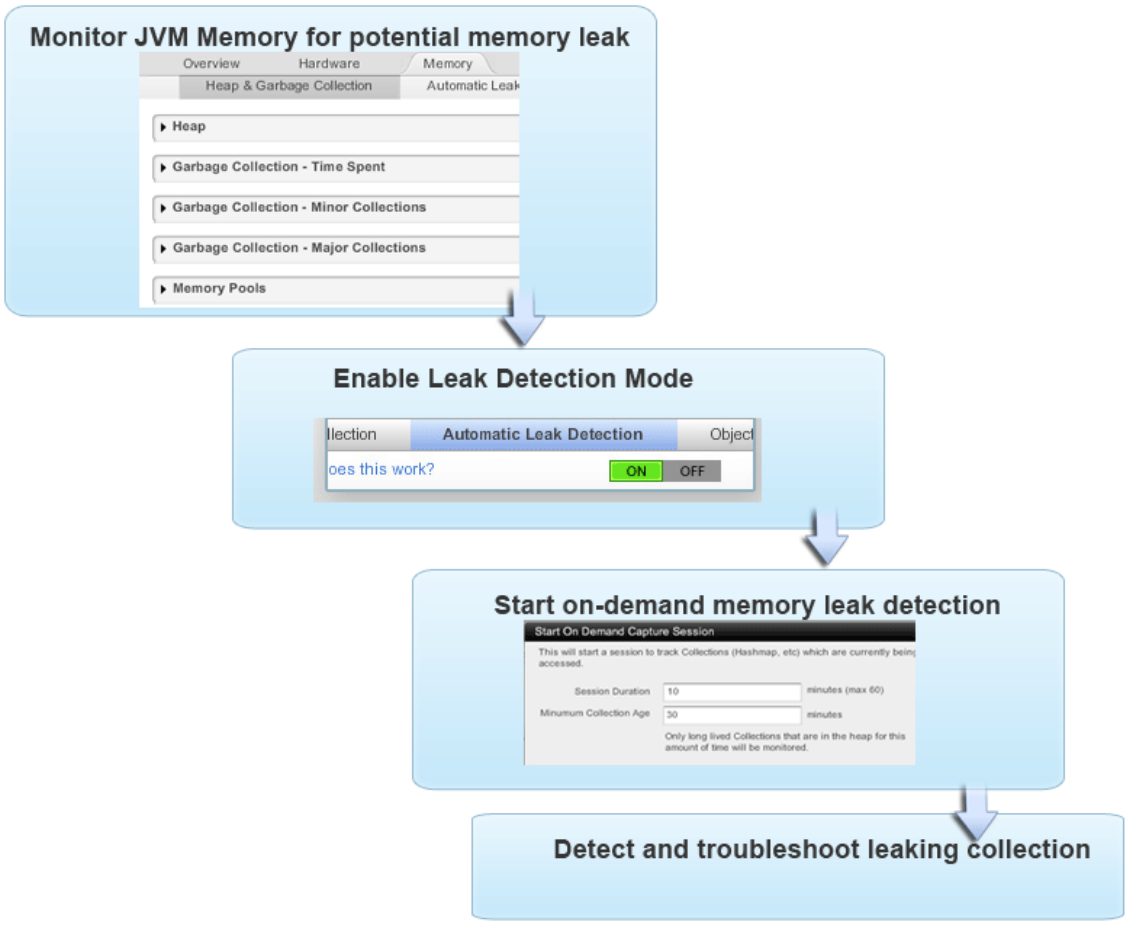
For a Collection object to be identified and monitored, it must meet the following conditions:

- The Collection has been alive for at least *N* minutes. Default is 30 minutes, configurable with the minimum-age-for-evaluation-in-minutes node property.
- The Collection has at least *N* elements. Default is 1000 elements, configurable with the minimum-number-of-elements-in-collection-to-deep-size node property.
- The Collection Deep Size is at least *N* MB. Default is 5 MB, configurable with the minimum-size-for-evaluation-in-node property.
  The Deep Size is calculated by traversing recursive object graphs of all the objects in the Collection.

See App Agent Node Properties and App Agent Node Properties Reference.

Workflow to troubleshoot memory leaks

Use the following workflow to troubleshoot memory leaks on JVMs that have been identified with a potential memory leak problem:



Troubleshooting Memory Leaks

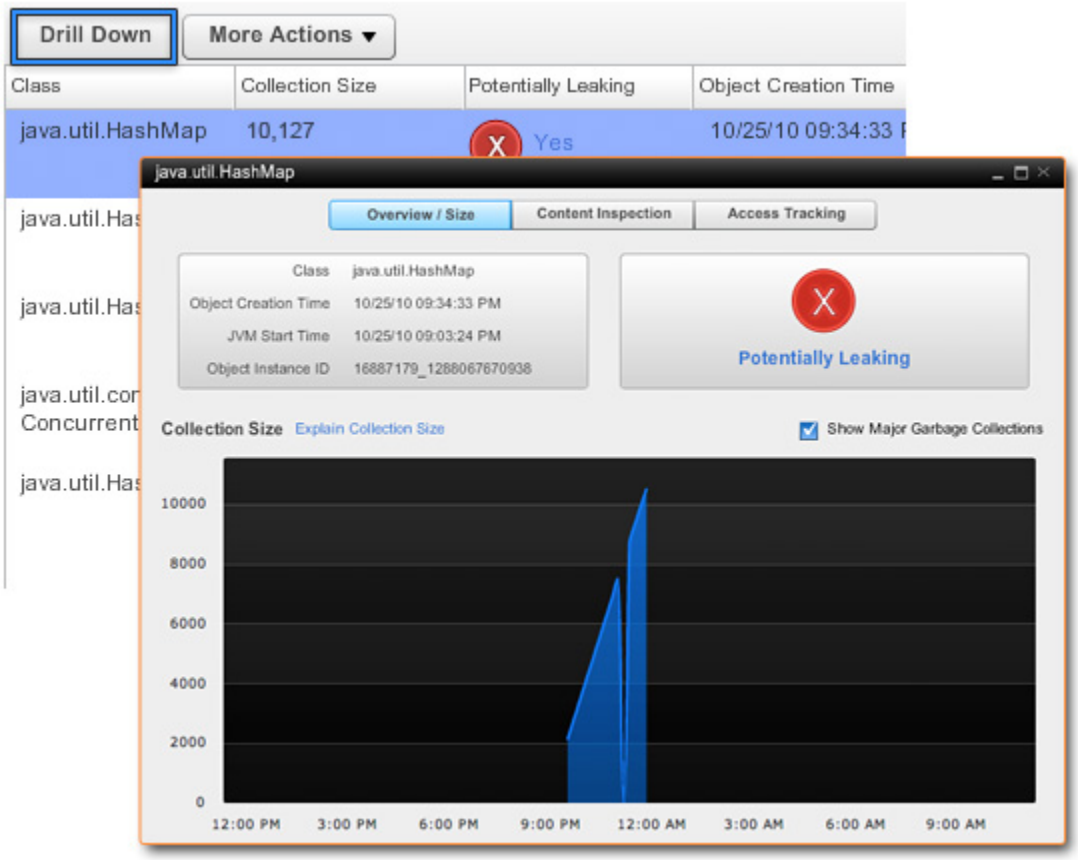
Troubleshooting a memory leak involves performing the following three actions:

- Select the Collection Object that you want to monitor
- Use Content Inspection
- Use Access Tracking

Select the Collection Object that you want to monitor

- Select the class name that you want to monitor.
- Click on the “Drill Down” button provided on the top of the memory leak dashboard.
  Alternatively right click on the class name and select the “Drill Down” option.

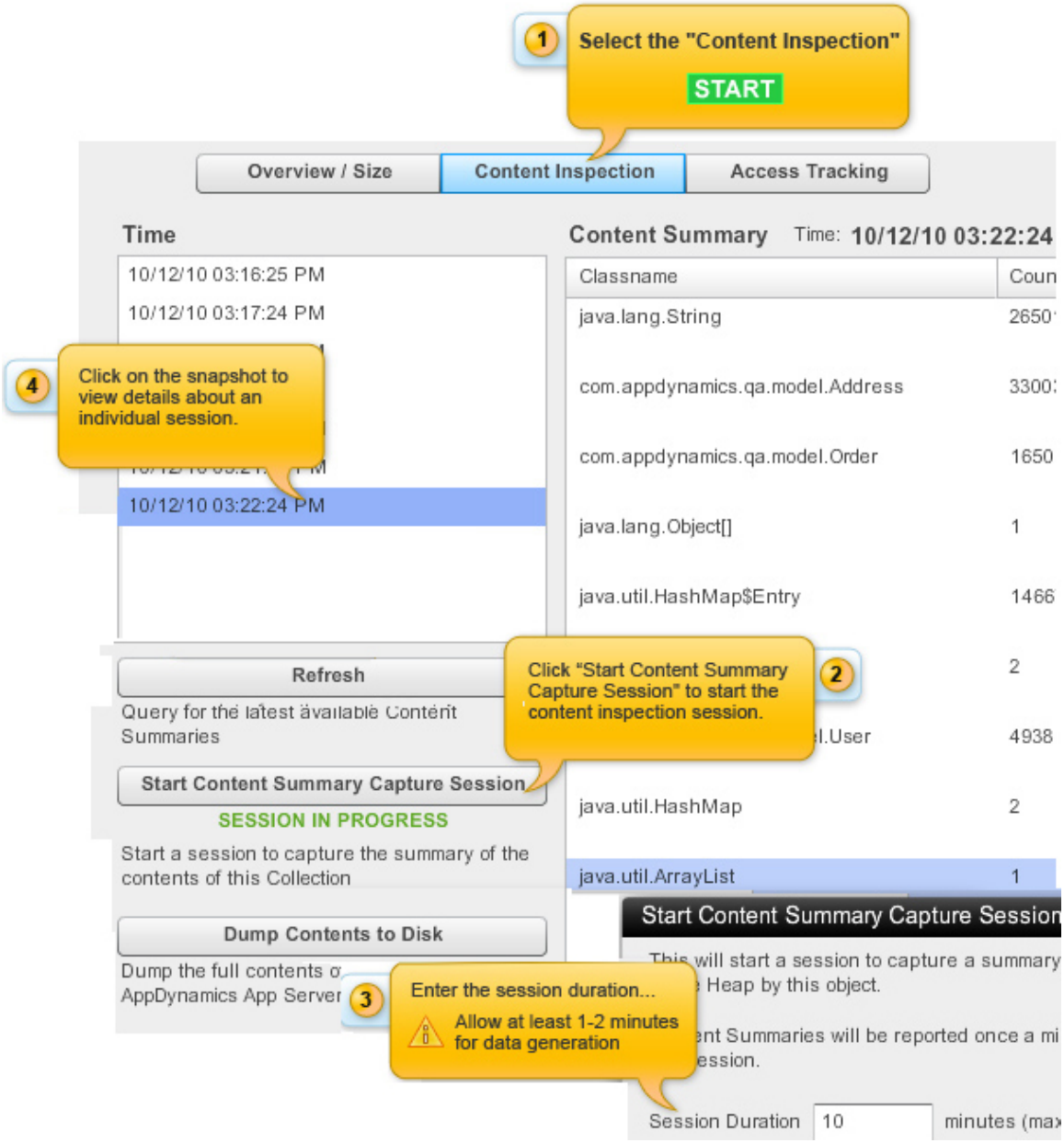
IMPORTANT: Ensure that all the instructions from the "Before you get started" section have been followed. To achieve optimum performance, start the troubleshooting session on a single Collection Object at a time.



Use Content Inspection

Use Content Inspection to identify which part of the application the Collection belongs to so that you can start troubleshooting. It allows monitoring histograms of all the elements in a particular Collection.
Start diagnostic session on the object and then follow the steps listed below:

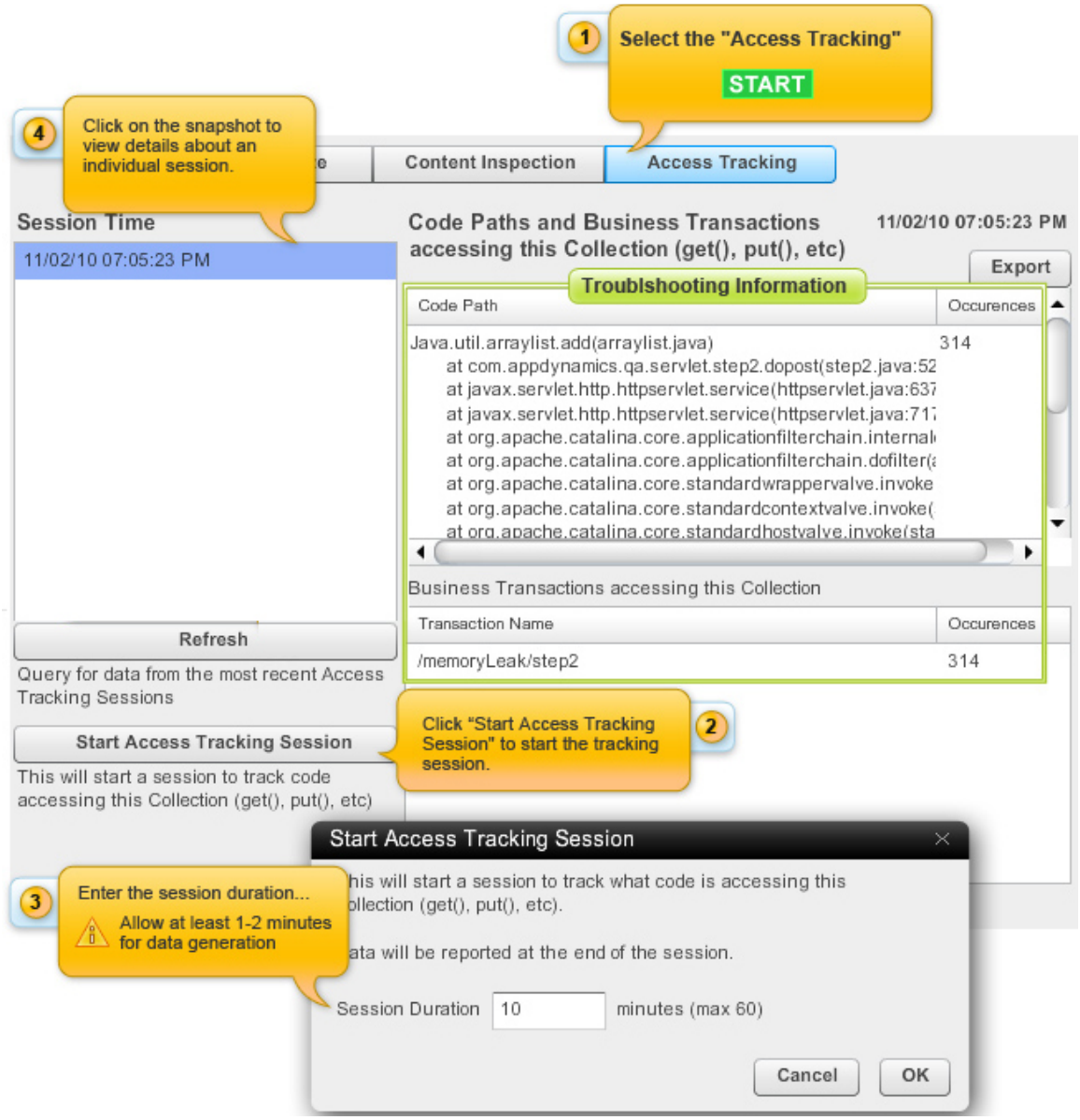
1. Select the **Content Inspection** tab.
2. Click **Start Content Summary Capture Session** to start the content inspection session.
3. Enter the session duration. Allow at least 1-2 minutes for data generation.
4. Click the refresh button to retrieve the session data.
5. Click on the snapshot to view details about an individual session.



Use Access Tracking

Use Access Tracking to view the actual code paths and business transactions accessing the Collections object. Start diagnostic session on the object and then follow the steps listed below:

1. Select the **Access Tracking** tab
2. Click **Start Access Tracking Session** to start the tracking session.
3. Enter the session duration. Allow at least 1-2 minutes for data generation.
4. Click the refresh button to retrieve session data.
5. Click on the snapshot to view details about an individual session.
   Note: You can also export the troubleshooting information into excel files using the "Export" button on the right side.



Learn More

- App Agent Node Properties

Troubleshoot Java Memory Thrash

- Memory Thrash and Object Instance Tracking
- Workflow for Detecting and Troubleshooting Memory Thrash
- Isolating and Analyzing Memory Thrash
    - To analyze memory thrash problems
    - To verify memory thrash
- Using Allocation Tracking
    - To use allocation tracking:

Memory Thrash and Object Instance Tracking

Memory thrash is caused when a large number of temporary objects are created in very short intervals. Although these objects are

temporary and are eventually cleaned up, the garbage collection mechanism struggles to keep up with the rate of object creation. Use object instance tracking to isolate the root cause of the memory thrash. To configure and enable object instance tracking, see Configure Object Instance Tracking (Java).

AppDynamics automatically tracks the following classes:

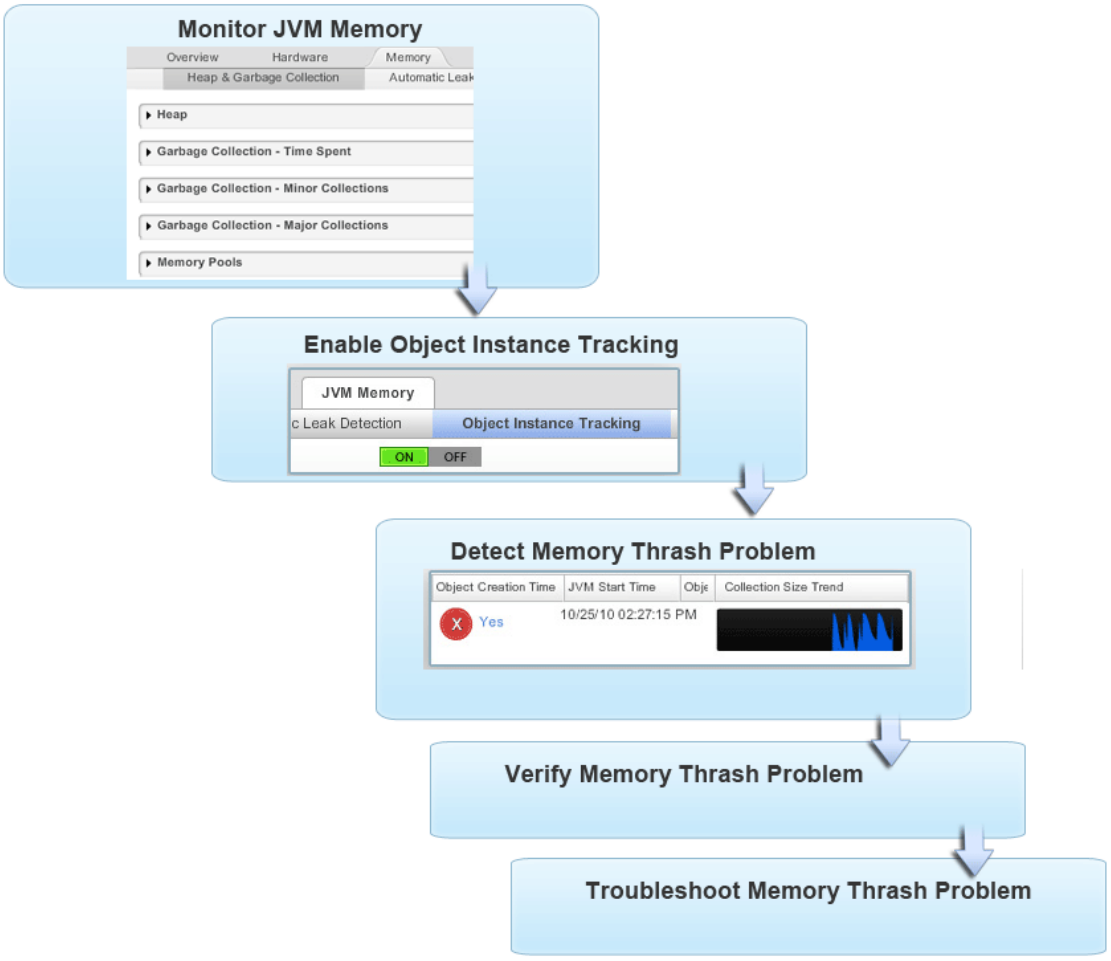
- Application Classes
- System Classes

Object instance tracking maps a histogram of every object in the JVM. The instance dashboard not only provides the number of instances for a particular class but also provides the shallow memory size (the memory footprint of the object and the primitives it contains) used by all the instances.

Workflow for Detecting and Troubleshooting Memory Thrash

The following diagram outlines the workflow for monitoring and troubleshooting memory thrash problems in a production environment.

Use the **JVM** tab in node dashboards to monitor leaks. See Troubleshoot Java Memory Leaks.



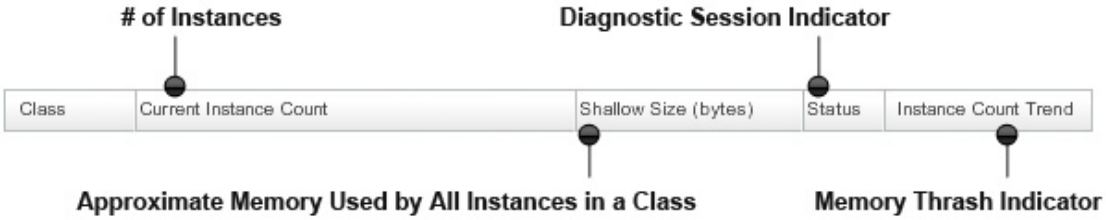
Isolating and Analyzing Memory Thrash

The Instance dashboard indicates possible cases of memory thrash. Prime indicators of memory thrash problems are:

- **Current Instance Count:** A high number indicates possible allocation of large number of temporary objects.
- **Shallow Size:** A large number for shallow size signals potential memory thrash.

- **Instance Count Trend:** A saw wave is an instant indication of memory thrash.

Once a memory thrash problem is identified in a particular Collection, start the diagnostic session.



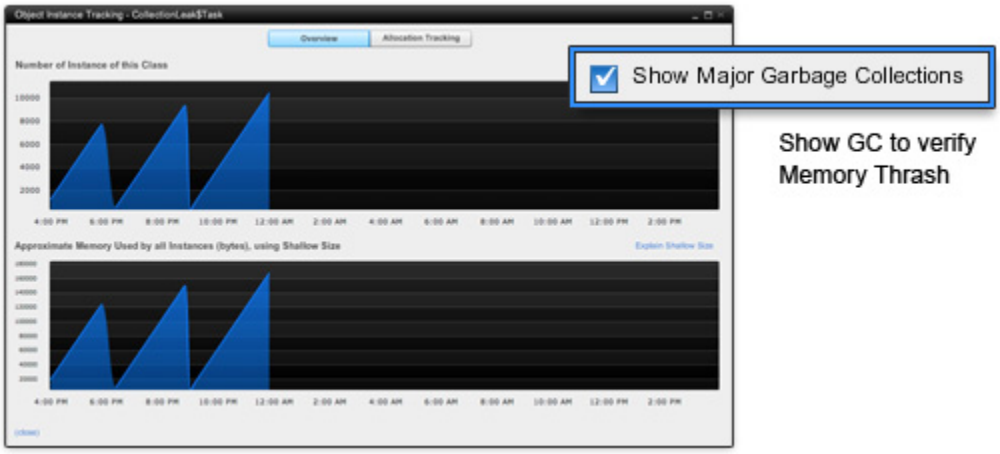
To analyze memory thrash problems

1. Select the class name to monitor and click **Drill Down** at the top of the Object Instance Dashboard.
    - Or right click on the class name and select the “Drill Down” option.
      Note: To achieve optimum performance, trigger drill down action on a single instance /class name at a time.

After the drill down action is triggered, data collection for object instances is performed every minute.

To verify memory thrash

1. Enable “Show Major Garbage Collections”.
2. If the instance count doesn’t vary with the GC cycle, it is an indication of potential leak (see Troubleshoot Java Memory Leaks).



Using Allocation Tracking

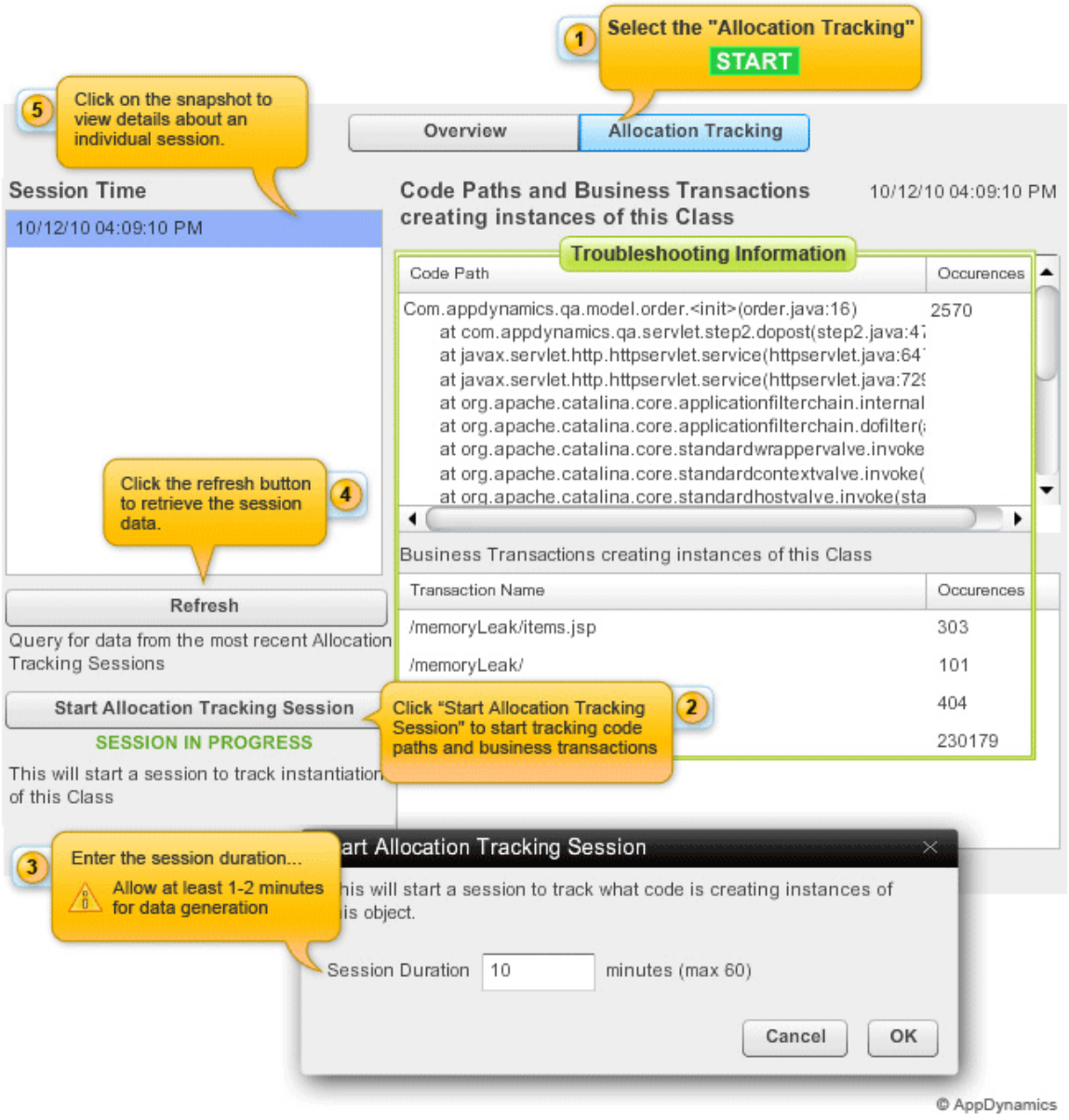
Allocation Tracking tracks all the code paths and those business transactions that are allocating instances of a particular class.

Allocation tracking detects those code path/business transactions that are creating and throwing away instances.

To use allocation tracking:

1. Trigger Drill Down action.
2. Select "Allocation Tracking" from the pop-up window.
3. Choose “Start Allocation Tracking Session “ to start tracking code paths and business transactions.
4. Enter the session duration and allow at least 1-2 minutes for data generation.
5. Click the refresh button to retrieve the session data.

6. Click on the particular snapshot to view the details about an individual session.



Monitor Java Applications

Monitor JVMs

- Infrastructure Monitoring in a Java Environment
- JVM Key Performance Indicators
  - Memory Usage and Garbage Collection
    - To view heap usage, garbage collection, and memory pools
    - Heap Usage

        - Garbage Collection
        - Memory Pools
    - Classes, Threads, and Process CPU Usage
        - To view class, thread, and CPU usage
- Alerting for JVM Health
- Monitoring JVM Configuration Changes
- Detecting Memory Leaks
    - Automatic Leak Detection
        - To enable automatic leak detection
- Detecting Memory Thrash
    - Object Instance Tracking
        - To monitor Java object instances
- Monitoring Long-lived Collections
        - To view or configure custom memory structures
- Learn More

AppMan Advice

JVM/container configuration can often be a root cause for slow performance because not enough resources are available to the application.

Infrastructure Monitoring in a Java Environment

A Java application environment has multiple functional subsystems. These are usually instrumented using JMX (Java Management Extensions) or IBM Performance Monitoring Infrastructure (PMI). AppDynamics automatically discovers JMX and PMI attributes.

JMX uses objects called MBeans (Managed Beans) to expose data and resources from your application. In a typical application environment, there are three main layers that use JMX:

- JVMs provide built-in JMX instrumentation, or platform-level MBeans that supply important metrics about the JVM.
- Application servers provide server or container-level MBeans that reveal metrics about the server.
- Applications often define custom MBeans that monitor application-level activity.

MBeans are typically grouped into domains to indicate where resources belong. Usually in a JVM there are multiple domains. For example, for an application running on Apache Tomcat there are “Catalina” and “Java.lang” domains. “Catalina” represents resources and MBeans relating to the Tomcat container, and “Java.lang” represents the same for the JVM Hotspot runtime. The application may have its own custom domains.

For more information about JMX see the JMX overview and tutorial. To learn about PMI see Writing PMI Applications Using the JMX Interface.

JVM Key Performance Indicators

There are often thousands of attributes, however, you may not need to know about all of them. By default, AppDynamics monitors the attributes that most clearly represent key performance indicators and provide useful information about short and long term trends. The preconfigured JVM metrics include:

- Total classes loaded and how many are currently loaded
- Thread usage
- Percent CPU process usage
- On a per-node basis:
    - Heap usage
    - Garbage collection
    - Memory pools and caching
    - Java object instances

You can configure additional monitoring for:

- Automatic leak detection
- Custom memory structures

Memory Usage and Garbage Collection

Monitoring garbage collection and memory usage can help you identify memory leaks or memory thrash that can have a negative impact application performance.

To view heap usage, garbage collection, and memory pools

1. In the left navigation pane, click **Servers - > App Servers -> <tier> -> <node>**. The Node Dashboard opens.

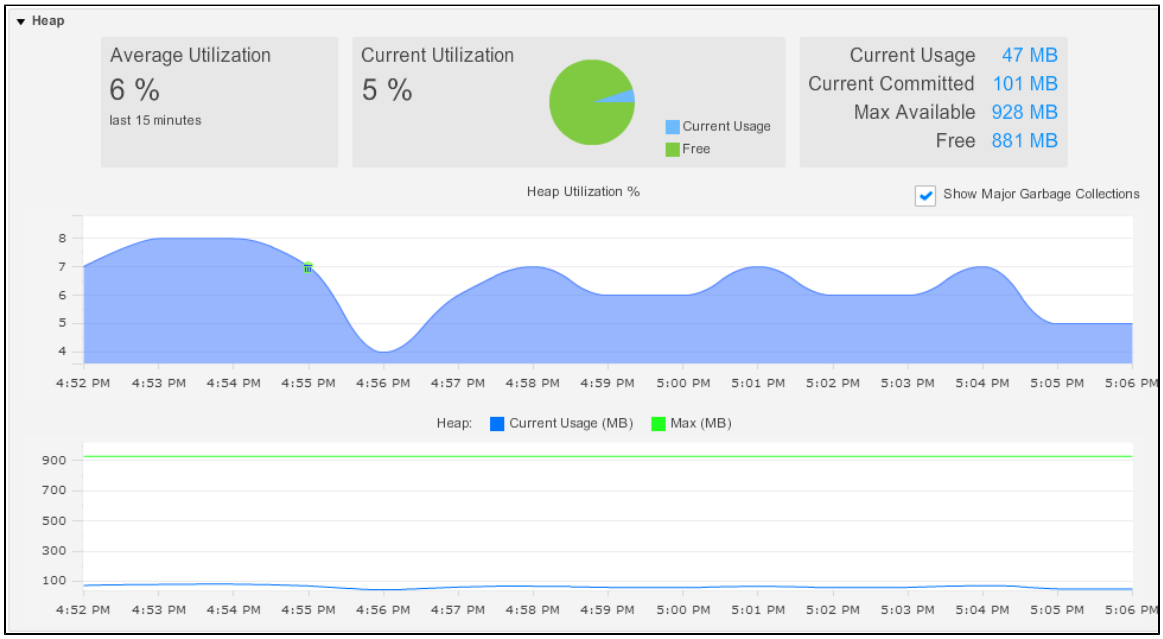
2. In the Node Dashboard, click the **Memory** tab.

3. In the Memory panel, click the **Heap & Garbage Collection** tab. The panels show data about the current usage.

See Node Dashboard for a complete description of the this dashboard.

Heap Usage

The Heap panel shows data about the current usage.

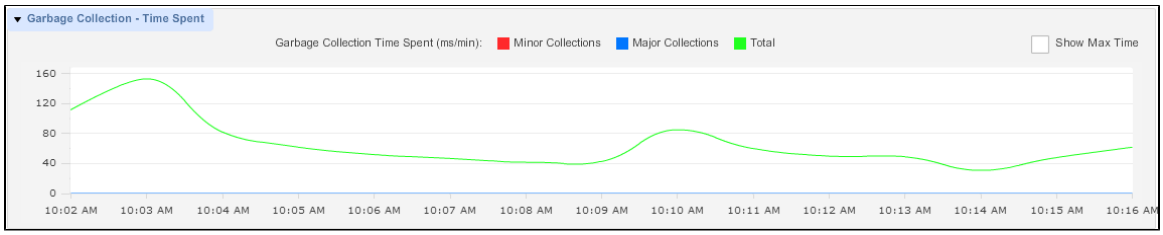


Garbage Collection

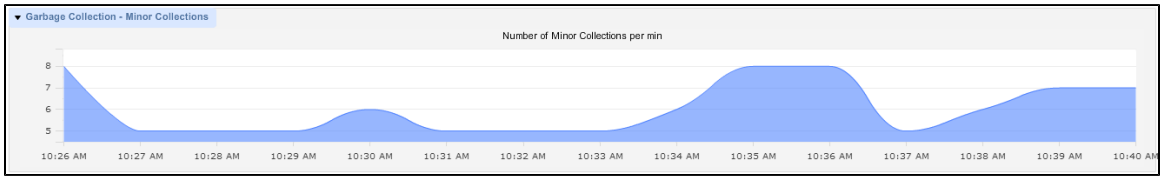
Java garbage collection refers to how the JVM monitors the objects in memory to find any objects which are no longer being referenced by the running application. Unused objects are deleted from memory to make room for new objects. For details see the Java documentation for Tuning Garbage Collection.

Garbage collection is a well-known mechanism provided by Java Virtual Machine to reclaim heap space from objects that are eligible for Garbage collection. The process of scanning and deleting objects can cause pauses in the application. Because this can be an issue for applications with large amounts of data, multiple threads, and high transaction rates, AppDynamics captures performance data about the duration of the pauses for garbage collection.

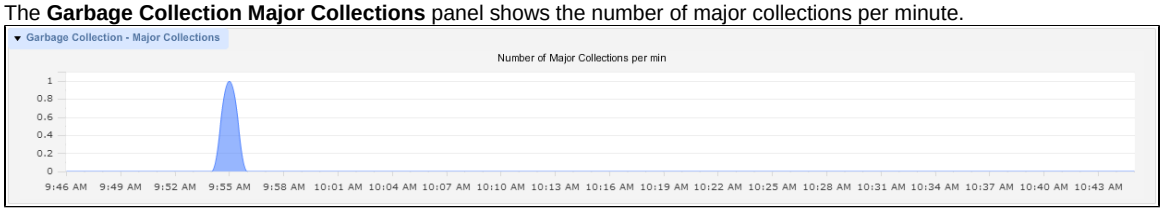
Below the **Heap** panel, the **Garbage Collection Time Spent** panel shows how much time, in milliseconds, it takes to complete both minor and major collections.



The **Garbage Collection Minor Collections** panel shows the number of minor collections per minute. The effectiveness of minor collections indicates better performance for your application.

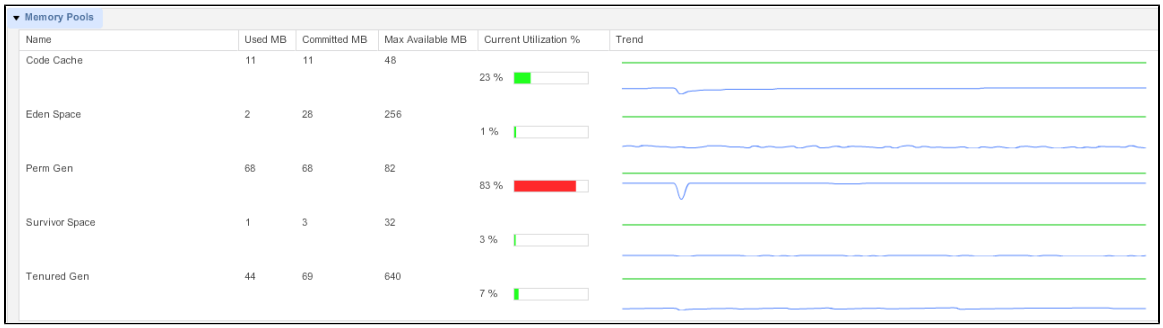


The **Garbage Collection Major Collections** panel shows the number of major collections per minute.



Memory Pools

The **Memory Pools** panel shows usage and trends about the Java memory pools.

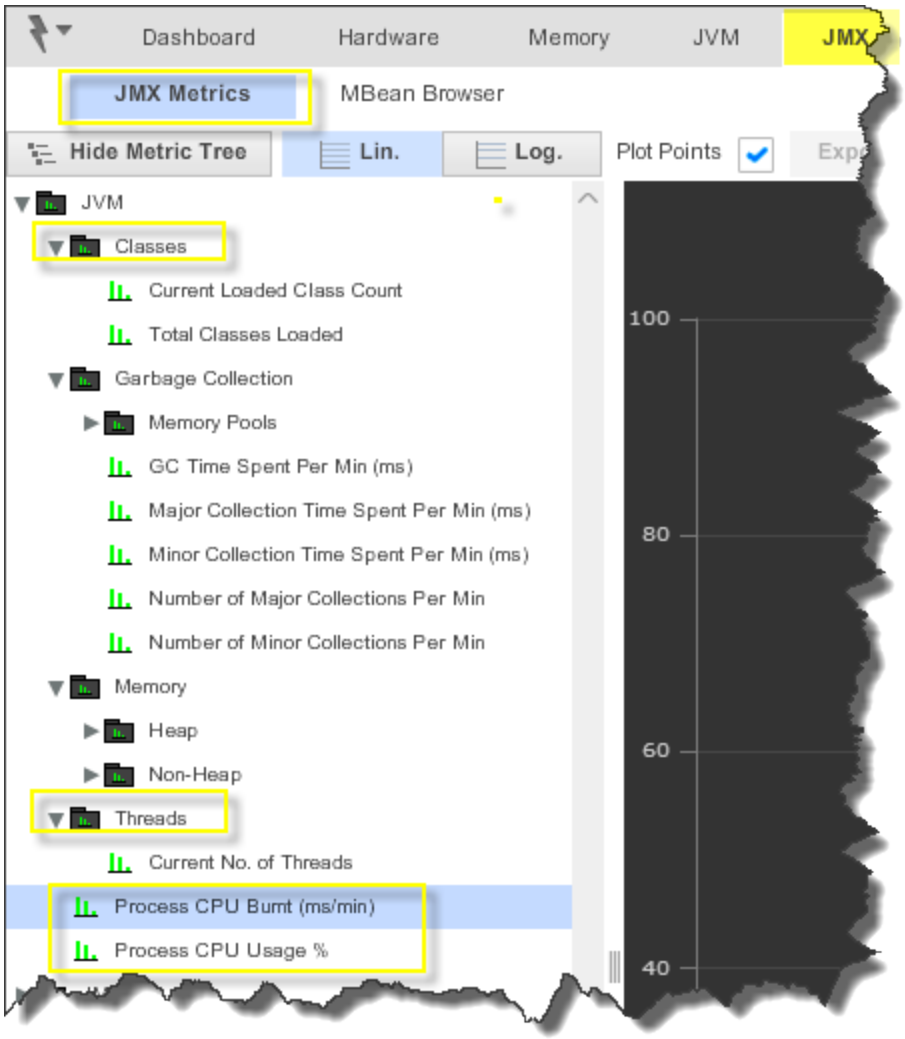


Classes, Threads, and Process CPU Usage

Information on JVM classes, threads and process CPU usage is available on the **JMX** tab of the Node Dashboard.

To view class, thread, and CPU usage

1. In the left navigation pane, click **Servers - > App Servers -> <tier> -> <node>**. The Node Dashboard opens.
2. In the Node Dashboard, click the **JMX** tab.
3. In the **JMX Metrics** panel, click **JVM**. The panels show data about the current usage.



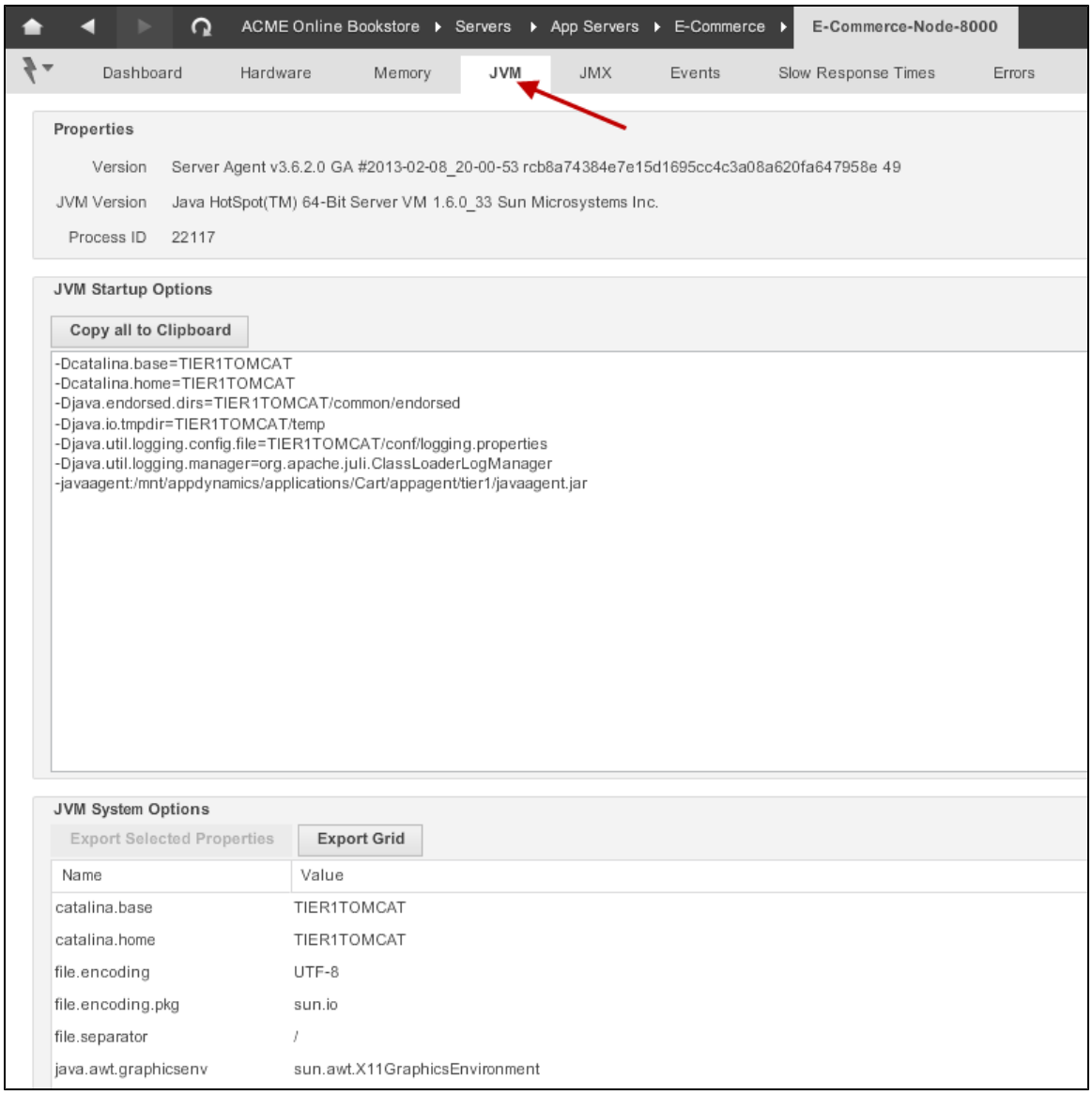
Alerting for JVM Health

You can set up health rules based on JVM/JMX metrics. Once you have a health rule, you can create specific policies based on health rule violations. One type of response to a health rule violation is an alert. See Alert and Respond for a discussion of how health rules, alerts, and policies can be used.

You can also create additional persistent JMX metrics from MBean attributes. See Configure JMX Metrics from MBeans.

Monitoring JVM Configuration Changes

The **JVM** tab of the Node Dashboard displays the JVM version, startup options, system options and environment properties for the node.



Changes to the application configuration generate events that can be viewed in the Events list.

For more information, see Monitor Application Change Events.

Detecting Memory Leaks

By monitoring JVM heap utilization and memory pool usage you can identify potential memory leaks. Consistently increasing heap valleys may indicate either an improper heap configuration or a memory leak. You might identify potential memory leaks by analyzing the usage pattern of either the survivor space or the old generation. To troubleshoot memory leaks see Troubleshoot Java Memory Leaks.

Automatic Leak Detection

AppDynamics supports automatic leak detection for some JVMs as listed at the Compatibility Matrix for Memory Monitoring. By default this functionality is not enabled, because using this mode results in higher overhead on the JVM. AppDynamics recommends that you enable leak detection mode only when you suspect a memory leak problem and that you turn it off once the leak is identified and remedied.

Memory leaks occur when an unused object's references are never freed. These are the most common occurrences in Collections classes, such as HashMap. This is caused when an application code puts objects in Collections but does not remove them even when

they are not being actively used. In production environments with high workloads, a frequently accessed Collection with a memory leak can cause the application to crash.

AppDynamics automatically tracks every Java Collection (HashMap, ArrayList, and so on) that has been alive in the heap for more than 30 minutes. The Collection size is tracked and a linear regression model identifies if the Collection is potentially leaking. You can then identify the root cause of the leak by tracking frequent accesses of the Collection over a period of time.

View this data on the Node Dashboard on the **Automatic Leak Detection** panel of the **Memory** tab. See Node Dashboard for a complete description of this dashboard and its tabs.

To enable automatic leak detection

To enable automatic leak detection follow the instructions at Troubleshoot Java Memory Leaks.

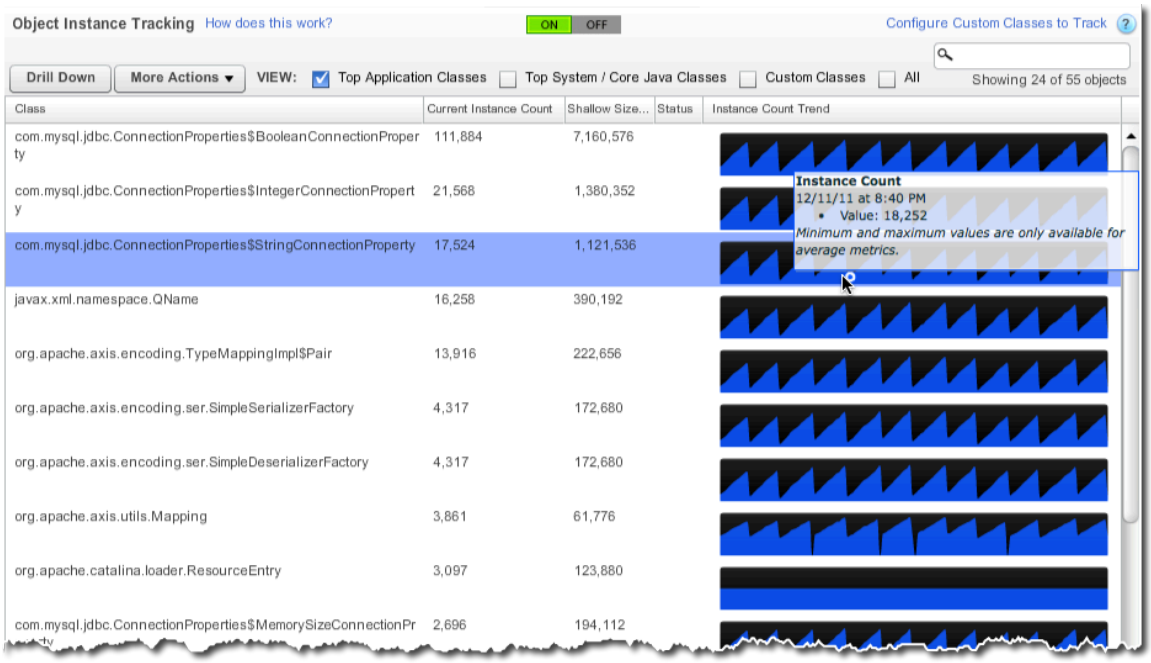
Detecting Memory Thrash

Memory thrash is caused when a large number of temporary objects are created in very short intervals. Although these objects are temporary and are eventually cleaned up, the garbage collection mechanism may struggle to keep up with the rate of object creation. This may cause application performance problems. Monitoring the time spent in garbage collection can provide insight into performance issues, including memory thrash. For example, an increase in the number of spikes for Major Collections either slows down a JVM or indicates potential memory thrash. To troubleshoot memory thrash, see Troubleshoot Java Memory Thrash.

Object Instance Tracking

The **Object Instance Tracking** panel helps you isolate the root cause of possible memory thrash. By default,AppDynamics tracks the object instances for the top 20 core Java classes and the top 20 application classes. For the list of the supported JVMs see the Compatibility Matrix for Memory Monitoring.

The **Object Instance Tracking** panel provides the number of instances for a particular class and graphs the count trend of those object in the JVM. It provides the shallow memory size (the memory footprint of the object and the primitives it contains) used by all the instances.



To monitor Java object instances

1. In the Node Dashboard, click the **Memory** tab.

3. In the Memory panel, click the **Object Instance Tracking** tab.

For details see Configure Object Instance Tracking (Java).

Monitoring Long-lived Collections

AppDynamics automatically tracks long lived Java Collections (HashMap, ArrayList, and so on) with Automatic Leak Detection. You can also configure tracking of specific classes using the Custom Memory Structures capability. You can use this capability to monitor a custom cache or other structure that is not a Java Collection. Custom memory structures are used as caching solutions. For example, you may have a custom cache or a third party cache such as Ehcache. In a distributed environment, caching can easily become a prime source of memory leaks. In addition, custom memory structures may or may not contain Collections objects that would be tracked using automatic leak detection. It is therefore important to manage and track these memory structures.

AppDynamics provides visibility into:

- Cache access for slow, very slow, and stalled business transactions
- Usage statistics (rolled up to Business Transaction level)
- Keys being accessed
- Deep size of internal cache structures

To view or configure custom memory structures

1. In the Node Dashboard, click the **Memory** tab.

3. In the **Memory** panel, click **Custom Memory Structures**.

For details see Configure Custom Memory Structures (Java).

Learn More

- Configure Policies
- Supported Environments and Versions
- Infrastructure Metrics
- Monitor Events

Monitor Java App Servers

- Infrastructure Monitoring in a Java Environment
- App Server Key Performance Indicators
- Alerting for App Server Health
- Learn More

AppMan Advice

JVM/container configuration can often be a root cause for slow performance because not enough resources are available to the application.

Infrastructure Monitoring in a Java Environment

A Java application environment has multiple functional subsystems. These are usually instrumented using JMX (Java Management Extensions) or IBM Performance Monitoring Infrastructure (PMI). AppDynamics automatically discovers JMX and PMI attributes.

JMX uses objects called MBeans (Managed Beans) to expose data and resources from your application. In a typical application environment, there are three main layers that use JMX:

- JVMs provide built-in JMX instrumentation, or platform-level MBeans that supply important metrics about the JVM.
- Application servers provide server or container-level MBeans that reveal metrics about the server.
- Applications often define custom MBeans that monitor application-level activity.

MBeans are typically grouped into domains to indicate where resources belong. Usually in a JVM there are multiple domains. For example, for an application running on Apache Tomcat there are “Catalina” and “Java.lang” domains. “Catalina” represents resources and MBeans relating to the Tomcat container, and “Java.lang” represents the same for the JVM Hotspot runtime. The application may have its own custom domains.

For more information about JMX see the JMX overview and tutorial. To learn about PMI see Writing PMI Applications Using the JMX Interface.

App Server Key Performance Indicators

AppDynamics creates long-term metrics of the key MBean attributes that represent the health of the Java container. Depending on your application configuration, metrics may include:

- Session information such as the number of active and expired sessions, maximum active sessions, processing time, average and maximum alive times, and a session counter.
- Web container runtime metrics that represent the thread pool that services user requests. The metrics include pending requests and number of current threads servicing requests. These metrics are related to Business Transaction metrics such as response time.
- Messaging metrics related to JMS destinations, including the number of current consumers and the number of current messages.
- JDBC connection pool metrics including current pool size and maximum pool size.

To see the JMX metrics discovered in a node, see the JMX tab on the Node Dashboard.

To learn how to customize additional MBean attributes for long-term monitoring, see Configure JMX Metrics from MBeans.

Alerting for App Server Health

AppDynamics discovers metrics for most Java platforms and applications. Some environments however are not instrumented by default, yet they have MBeans. For those situations you can enable monitoring using the MBean Browser. For details see Monitor JMX MBeans.

In addition to the preconfigured metrics, you may be interested in additional JVM or Java container metrics. You can add custom metrics using JMX MBean attributes in the Metric Browser. To customize which MBean attributes are monitored, see Configure JMX Metrics from MBeans.

Once you add a custom metric you can create a custom health rule for it and receive alerts if conditions indicate problems. For details see Alert and Respond.

AppDynamics also provides the Application Server Agent API (Agent API) for access to metrics that are not supported by default or by MBeans. You can use the Agent API to:

- Inject custom events and report on them
- Create and report on new metrics
- Correlate distributed transactions when using protocols that AppDynamics does not support

Learn More

- Configure JMX Metrics from MBeans
- Monitor JMX MBeans
- Configure Health Rules
- Supported Environments and Versions

Monitor JMX MBeans

- JMX MBeans and Monitoring Application Infrastructure
  - Prerequisites for JMX Monitoring
  - Preconfigured JMX Metrics
    - To view the configuration of the preconfigured JMX metrics
- Using AppDynamics for JMX Monitoring
    - To view JMX metrics in the Metrics Browser

- Trending MBeans Using Live Graphs
  - To monitor the real-time trend of an MBean
- Configuring New JMX Metrics
- Reusing JMX Metric Configurations
- Learn More

This topic discusses how to provide visibility into the JMX metrics for your JVM and application server.

JMX MBeans and Monitoring Application Infrastructure

As discussed at Monitor JVMs and Monitor Java App Servers, AppDynamics uses JMX (Java Management Extensions) to monitor Java applications.

JMX uses objects called MBeans (Managed Beans) to expose data and resources from your application. You can use one or more MBean attributes to create persistent JMX metrics in AppDynamics. In addition, you can import and export JMX metric configurations from one version or instance of AppDynamics to another.

Prerequisites for JMX Monitoring

AppDynamics can capture MBean data, when these conditions are met:

- The monitored system must be running on Java 1.5 or later.
- Each monitored Java process must enable JMX. See the JMX documentation.

Additional MBean data may be available when a monitored business application exposes Managed Beans (MBeans) using standard JMX. See the MBean documentation.

Preconfigured JMX Metrics

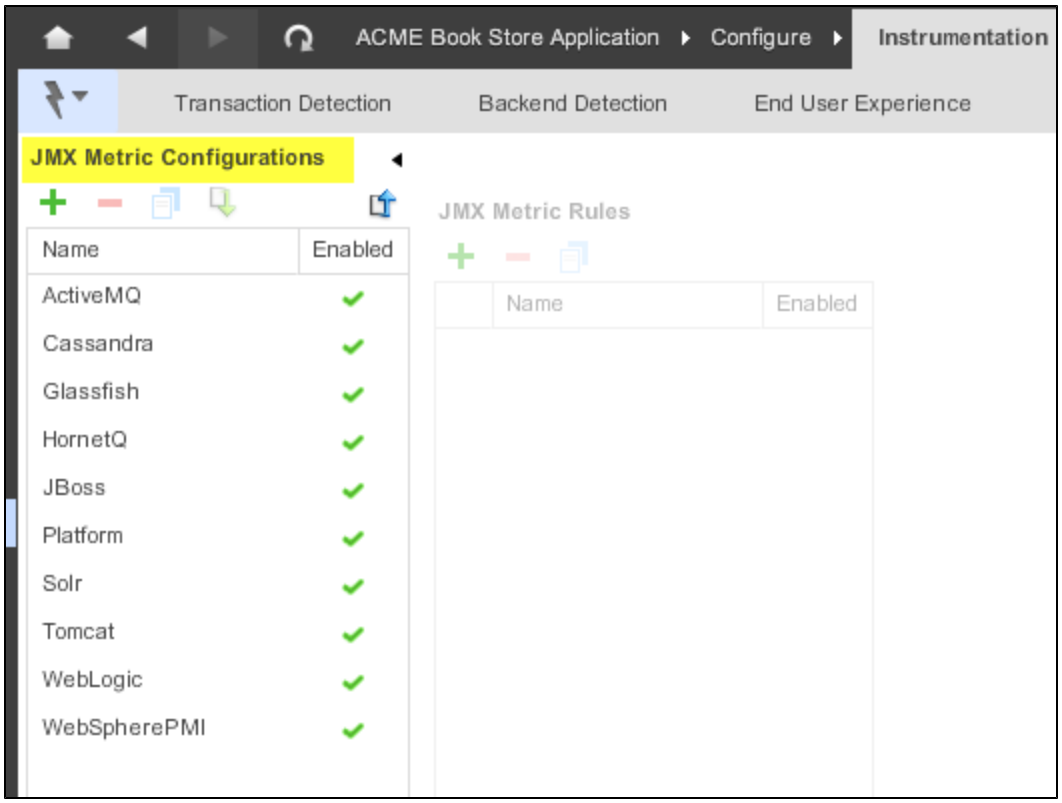
AppDynamics provides preconfigured JMX metrics for several common app server environments:

- Apache ActiveMQ
- Cassandra
- GlassFish
- HornetQ
- JBoss
- Apache Solr
- Apache Tomcat
- Oracle WebLogic Server
- WebSphere PMI

For app server environments that are not instrumented out-of-the-box, you can configure a new JMX metrics configuration. You can also add new metric rules to the existing set of configurations. For example, Glassfish JDBC connection pools can be manually configured using MBean attributes and custom JMX metrics.

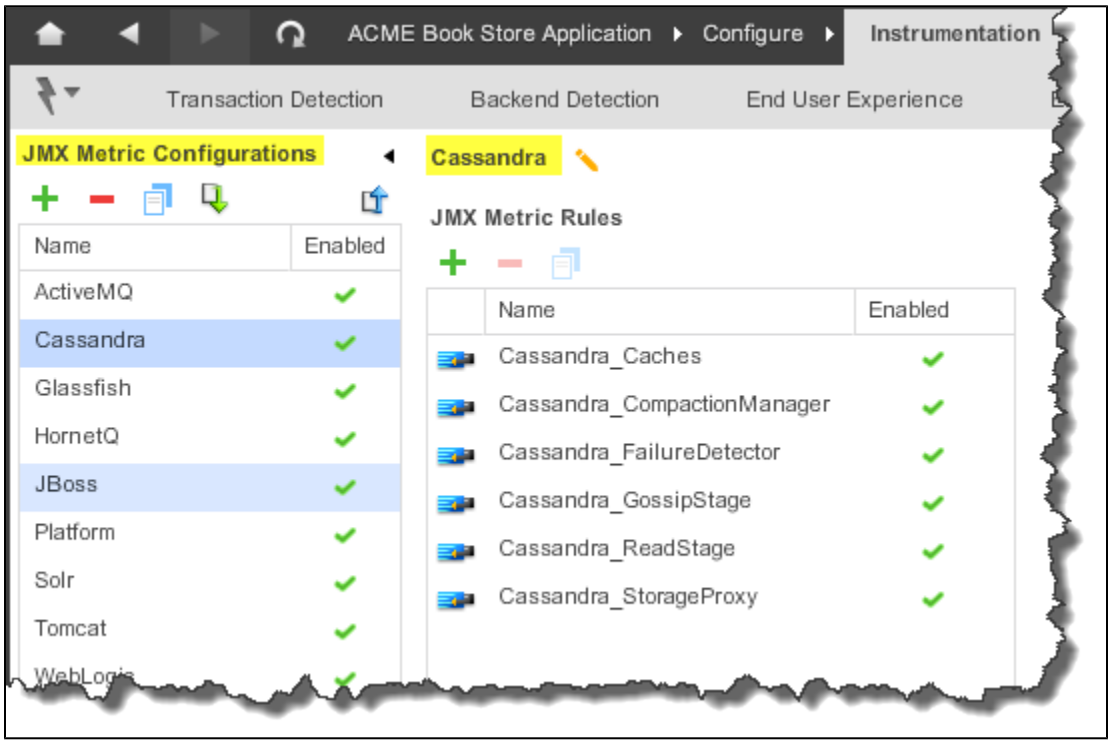
To view the configuration of the preconfigured JMX metrics

1. In the left navigation pane, click **Configure -> Instrumentation** and select the **JMX** tab.
The list of JMX Metric Configurations appears.



2. Click a metric configuration to view the preconfigured JMX metrics for that app server.

For example, selecting Cassandra shows the preconfigured JMX Metric Rules for Apache Cassandra. Double-click a metric rule to see configuration details such as the MBeans matching criteria and the MBean attributes being used to define the metric.



You can view, delete, and edit the existing JMX metric rules. You can add new JMX metric rules. See [37BKUP:Configure JMX Metrics from MBeans].

Using AppDynamics for JMX Monitoring

You can view MBean-based metrics using the Node Dashboard and the Metric Browser. In addition, the MBean Browser enables you to view all the MBeans defined in the system.

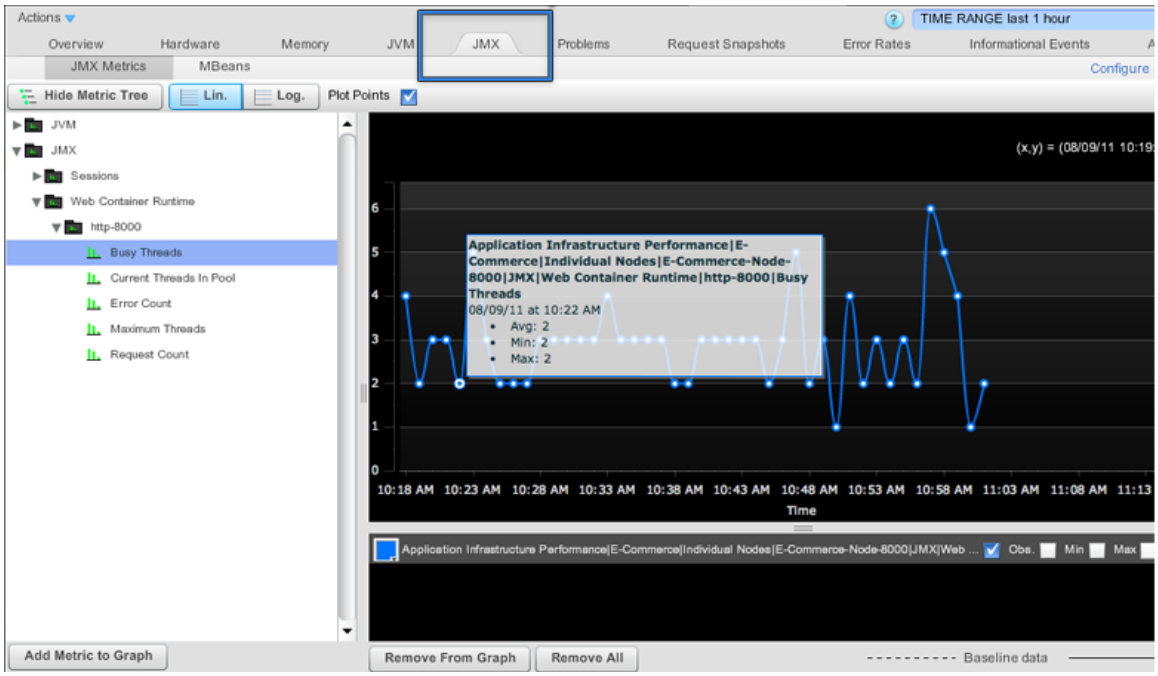
To view JMX metrics in the Metrics Browser

1. In the left navigation pane, click **Servers->App Servers-><Tier>><Node>**. The Node Dashboard opens.

2. Click the **JMX** tab. The JMX Metric Browser opens and displays the MBeans in a Metric Tree.

3. Browse the default JMX metrics.

4. To monitor a particular metric, double-click or drag and drop the metric onto the graph panel.



5. You can perform all the operations that are provided by the Metric Browser such as:

- Drill-down
- Analyze the transaction snapshot for a selected time duration
- Set the selected time range as a global time range

Trending MBeans Using Live Graphs

You can monitor the trend of a particular MBean attribute over time using the **Live Graph**.

To monitor the real-time trend of an MBean

1. In the left navigation pane, click **Servers->App Servers->>**<Tier>><Node>. The Node Dashboard opens.

2. Click the **JMX** tab.

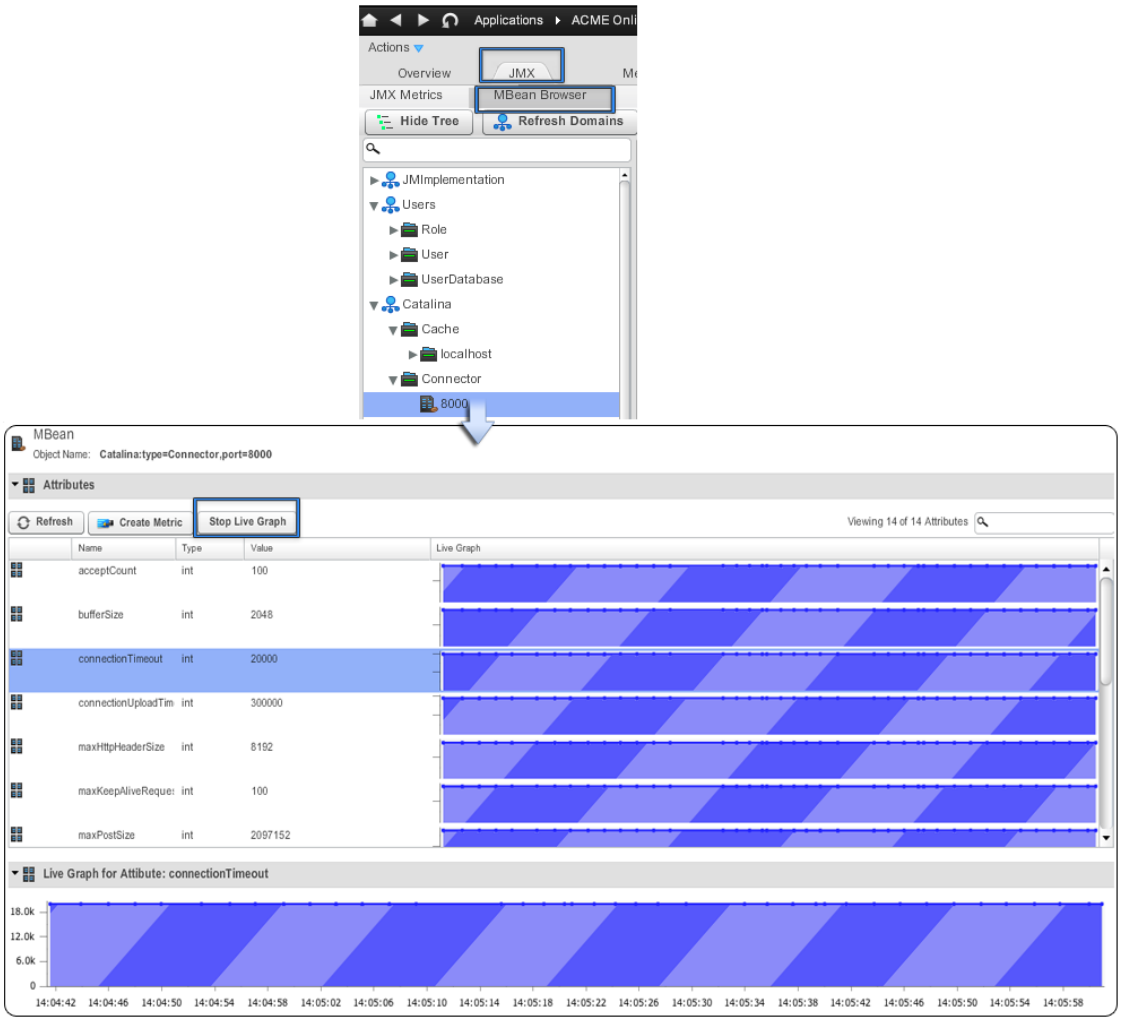
3. Click the **MBean Browser** sub-tab.

4. Select the domain for which you want to monitor MBeans. For a description of domains see Monitor JVMs.

5. Select the MBean that is of interest to you.

6. Click **Start Live Graph**. You can see the runtime values.

7. Select an attribute and click **Live Graph for Attribute** to see a larger view of a particular graph.



Configuring New JMX Metrics

In addition to the preconfigured metrics, you can define a new persistent metric using a JMX Metric Rule that maps a set of attributes from one or more MBeans.

You can create a JMX metric from any MBean attribute or set of attributes. Once you create a persistent JMX metric, you can:

- View it in the Metric Browser
- Add it to a Custom Dashboard
- Create a health rule for it so that you can receive alerts

The JMX Metrics Configuration panel is the central configuration interface for all of the JMX metrics that AppDynamics reports. You can use the MBean Browser to view MBeans exposed in your environment. From there, you can access the JMX Metrics Configuration panel by selecting an MBean attribute and clicking **Create Metric**.

For details, see Configure JMX Metrics from MBeans.

Reusing JMX Metric Configurations

Once you create a custom JMX metric, you can keep the configuration for upgrade or other purposes. The JMX metric information is stored in an XML file that you can export and then import to another AppDynamics system. For instructions see Import or Export JMX Metric Configurations.

Learn More

- Configure JMX Metrics from MBeans
- Monitor JVMs
- Import or Export JMX Metric Configurations
- Configure JMX Without Transaction Monitoring

Trace Multi-Threaded Transactions (Java)

- Thread Visibility
  - Asynchronous Calls in Dashboards
  - Threads and Thread Tasks in the Metric Browser
- Threads in Call Graphs
  - To Drill Down into Downstream Calls on a Thread
- Thread Metrics in Health Rules
- Learn More

Multithreaded programming techniques are common in applications that require asynchronous processing. Threads are first class entities that you can monitor.

Although each thread has its own call stack, multiple threads can access shared data. This creates two potential problems: visibility and access.

- A visibility problem occurs if thread A reads shared data which is later changed by thread B, and thread A is not aware of the change.
- An access problem occurs if several threads are trying to access and change the same shared data at the same time.

Visibility and access problems can lead to:

- Liveness failure: Application performance becomes sluggish or stops processing also known as a deadlock.
- Safety failure: Race condition that results in difficult to discover programming errors.

Thread contention can occur when multiple threads attempt to access a synchronized method or block at the same time. If a thread remains in the synchronized method or blocks for a long time, the other threads must wait for access to shared resources. This situation has an adverse effect on application performance. Call graphs for multi-threaded transactions enable you to trace thread creation in a business transaction and provide an aggregated view of the overall processing for transactions that spawn threads for concurrent processing.

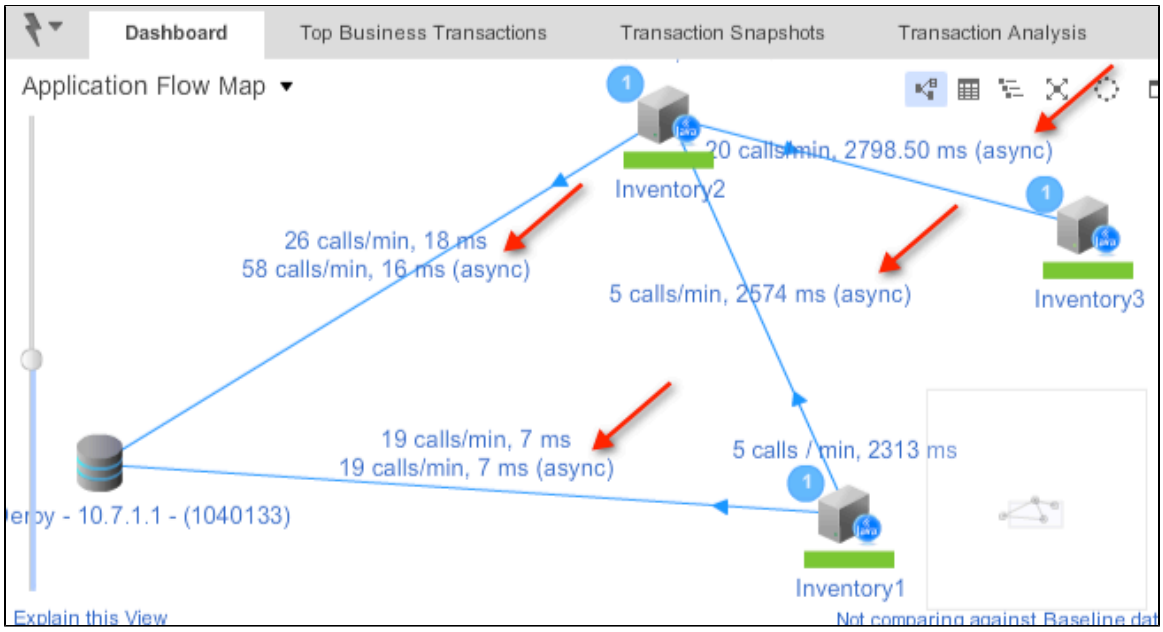
Thread Visibility

Application often spawn threads to perform concurrent tasks. You can monitor each thread as a separate entity, including exit calls and policies linked to a specific thread. All Runnables, Callables and Threads are instrumented by default except those that are explicitly excluded. In some environments, this could lead to too many classes being instrumented. In this case you can create custom rules to exclude them. If you do not want to monitor any threads, you can completely disable Asynchronous monitoring, which requires an agent restart. See Configure Multi-Threaded Transactions (Java).

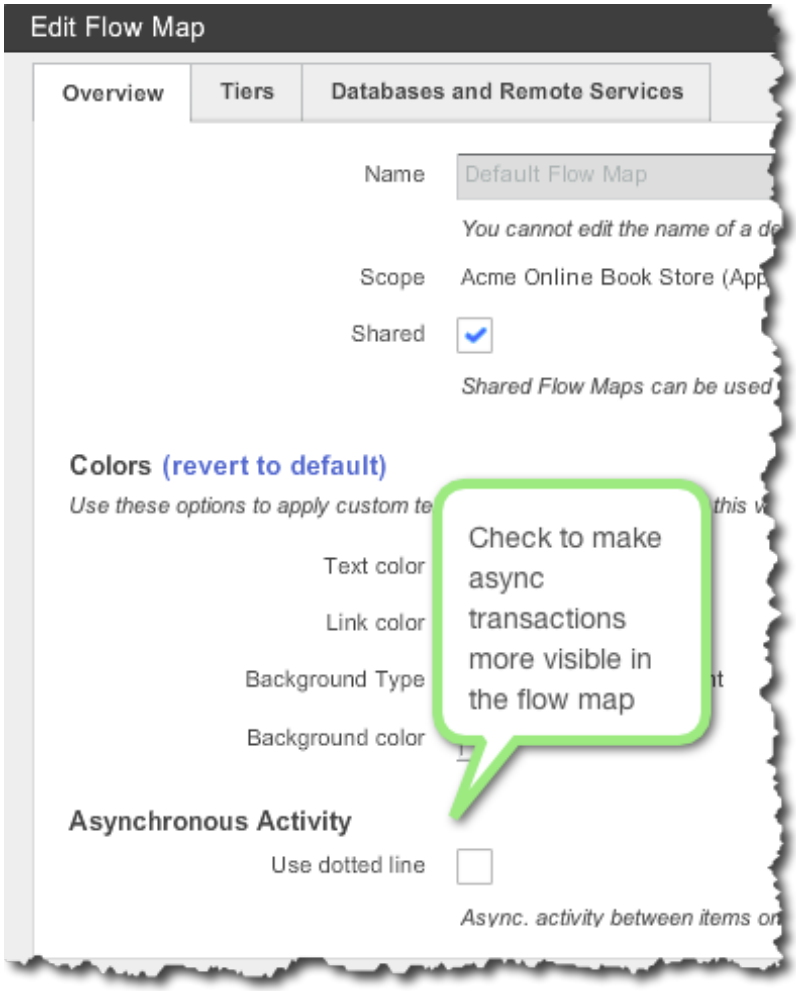
AppDynamics provides thread visibility in dashboards, the metric browser and call graphs.

Asynchronous Calls in Dashboards

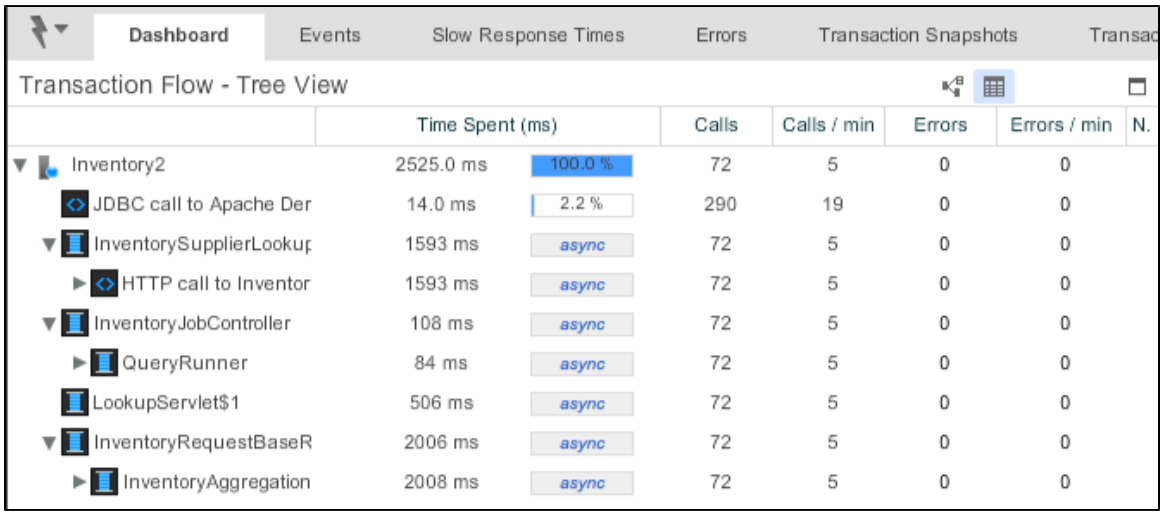
AppDynamics detects asynchronous calls in an application and labels them as "async" in the dashboards that display the asynchronous activity.



You can set the flow map lines to render as dotted lines for easier visibility. Click **Application Flow Map -> Edit Current Flow Map** and check **Use dotted line**.

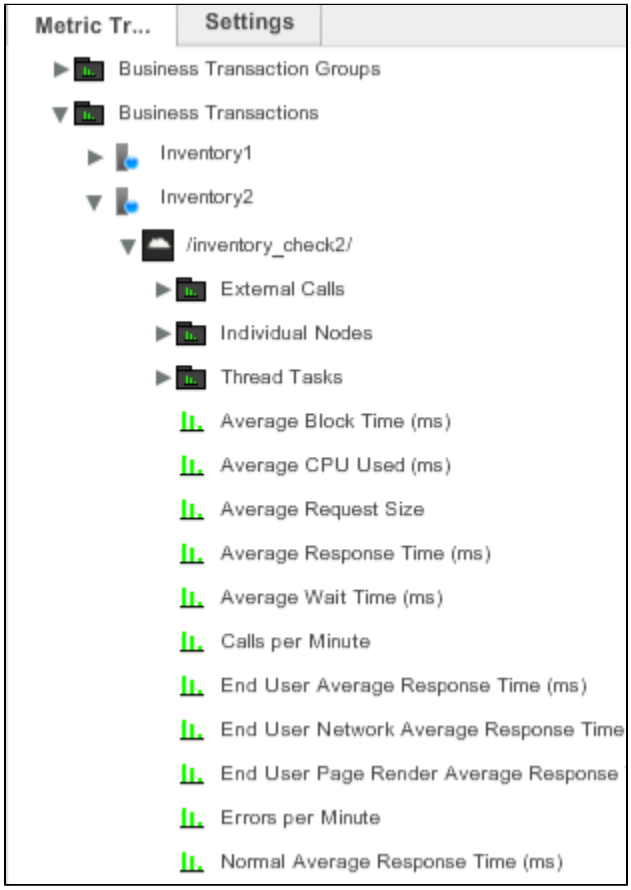


The tree view of a multi-threaded transaction dashboard shows the errors and time spent in asynchronous calls. You may need to expand the class to see all the threads.

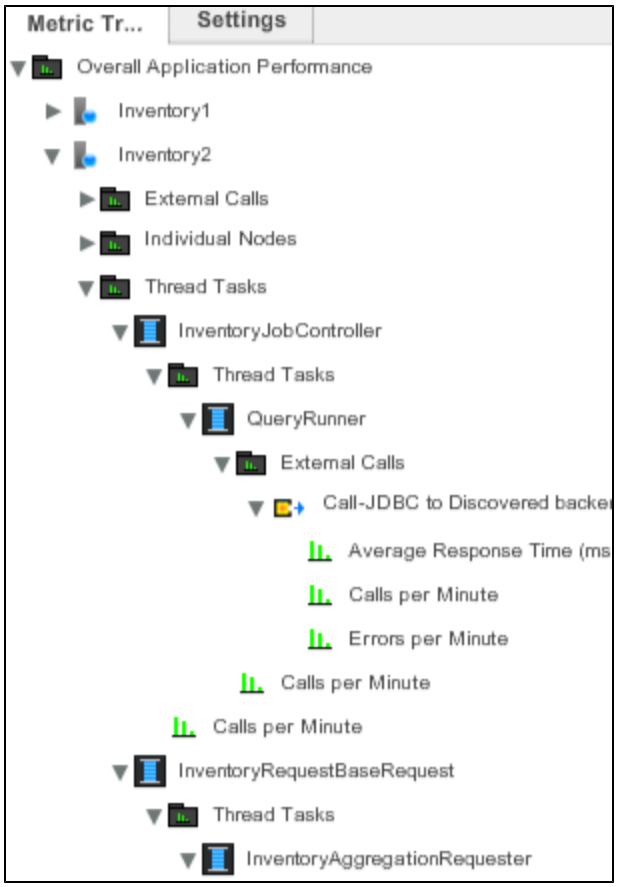


Threads and Thread Tasks in the Metric Browser

In a multi-threaded transaction, AppDynamics reports key performance metrics for the individual threads in Thread Tasks branch of the tier in the Metric Browser. The Thread Tasks branch is created only for multi-threaded transactions.



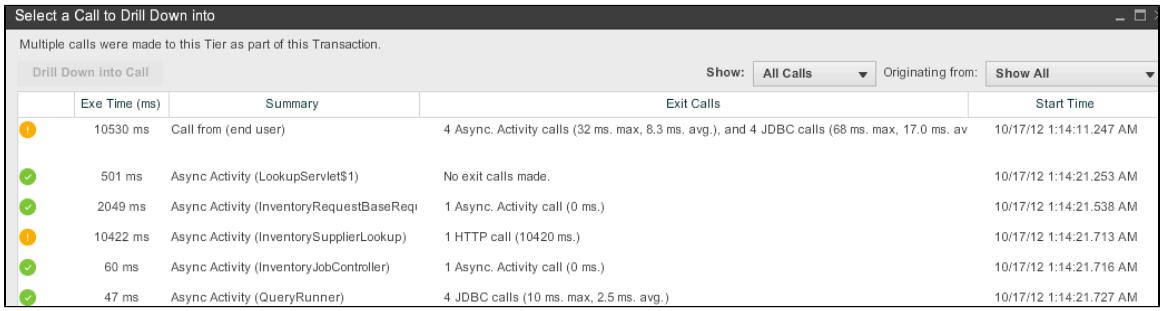
Thread Tasks are also reported in tiers under Overall Application Performance, where you can get metrics on specific calls made by each thread in a node or in a tier.



Under Application Infrastructure in the JVM section for a node or tier, you can get the number of threads spawned in the Current No. of Threads metric. This is a generic metric not necessarily limited to the thread tasks that AppDynamics tracks.

Threads in Call Graphs

When you drill down in a transaction snapshot with multiple calls, AppDynamics displays the list of calls that you can drill down into.

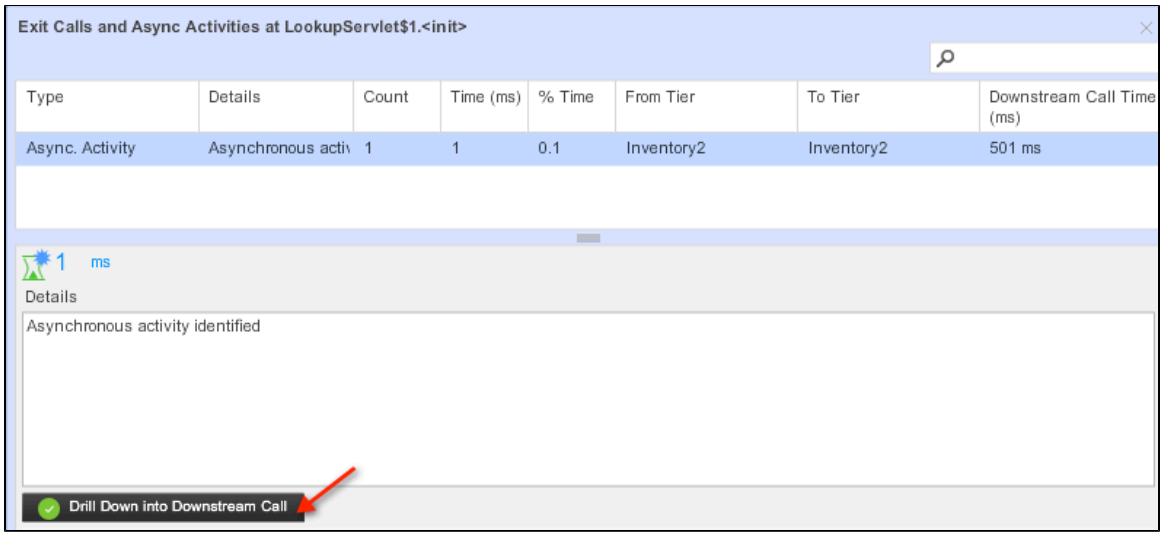


Select a call from the list and double-click or click **Drill Down into Call** to access the call graph for the thread.

To Drill Down into Downstream Calls on a Thread

If the call graph indicates Async Activity in the Exit Call/Threads column, you can drill down further into the downstream call on the thread:

1. Click **Async Activity** in the Exit Calls/Threads Column for the call that you want to drill down from.
2. In the Exit Calls and Async Activities window, click **Drill Down into Downstream Call**.



A call graph for the downstream call opens.

Thread Metrics in Health Rules

You can create a health rule of the custom health rule type based on performance metrics for a thread task.

When you click the metric icon in the Health Rule Wizard, the embedded metric browser includes the Thread Tasks if the entity for which you are configuring the health rule spawns multiple threads.

See Configure Health Rules.

Learn More

- Configure Multi-Threaded Transactions (Java only)
- Metric Browser
- Call Graphs
- Health Rules
- Configure Health Rules

