Enabling Service Excellence

Business Transaction Monitoring

Restricted Rights Legend

The information contained in this document is confidential and subject to change without notice. No part of this document may be reproduced or disclosed to others without the prior permission of eG Innovations Inc. eG Innovations Inc. makes no warranty of any kind with regard to the software and documentation, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Trademarks

Microsoft Windows, Windows 2008, Windows 2012, Windows 7, Windows 8, and Windows 10 are either registered trademarks or trademarks of Microsoft Corporation in United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Copyright

Table of Contents

1

Introduction

A business transaction represents a type of user request to a web application. For instance, the following types of requests are considered business transactions for an online retail banking application:

- Logging in
- Balance checking
- Funds transfer
- Bill payments
- Logging out

User experience with a web application not only relies on the successful completion of these user requests/transactions, but also on their rapid execution. This is why, even if a single transaction slows down, stalls, or fails, user dissatisfaction with the web application as a whole grows. This in turn may cause user complaints to increase, support costs to sky rocket, and revenues to dip.

To avoid such disastrous results, web application administrators should monitor every business transaction closely and promptly identify the slow/stalled/failed transactions. Most importantly, administrators will have to determine where and why these transactions under-performed – i.e., identify the root-cause of poor transaction performance - so that the problem can be quickly resolved before users begin doubting the stability of the web application.

Root-cause isolation is often the most challenging! This is because, most web applications these days overlay multi-tier environments characterized by multiple application servers, database servers, remote services, etc. Every business transaction to such web applications travels through multiple nodes, using remote calls to external services, to fulfill its purpose. For example, an online transaction to shop for goods may access a ShopCart web page on a web server. Every time an item is added to a shopping cart, the web server may make an HTTP/S call to a web application server to invoke the business logic. The business logic may then make a database call to run a query for retrieving the total count of goods that that user has shopped for so far. A slowdown in even one node or a delay in processing even a single remote service call can impact the performance of the transaction. To accurately isolate where the actual bottleneck lies, administrators should employ an APM solution that can trace the entire path of every business transaction, measure the total round-trip time of each transaction, identify the synchronous and asynchronous calls made by the transaction at various nodes, and compute the time spent by the transaction at each node, for each call. This can be achieved using the **eG Business Transaction Monitor (BTM)**.

1.1 The eG Business Transaction Monitor (BTM)

The **eG BTM** employs an advanced 'tag-and-follow' technique to trace the complete path of each business transaction to a web application, end-to-end. When doing so, it auto-discovers the application servers the

transaction travels through, and also automatically ascertains what remote service calls were made by the transaction when communicating with the servers. In the process, the eG BTM measures the following:

- The total response time of each transaction;
- The time spent by the transaction on each application server;
- The time spent by the transaction for processing every external service call (including SQL queries);

Using these analytics, the eG BTM precisely pinpoints the slow, stalled, and failed transactions to the web application, enables administrators to accurately isolate where – i.e., on which application server – the transaction was bottlenecked, and helps them figure out exactly what caused the bottleneck – an inefficient or errored query to the database? A slow HTTP/S call to another application server? a time-consuming POJO / JMX method execution? a slow SAP JCO/async call? By quickly leading administrators to the source of transaction failures and delays, the eG BTM facilitates rapid problem resolution, which in turn results in the low downtime of and high user satisfaction with the web application.

1.2 Pre-requisites for Business Transaction Monitoring Using eG

The following are the pre-requisites for performing business transaction monitoring using eG:

- The **eG Business Transaction Monitor (BTM)** can be installed on Java containers only - i.e., Java applications / J2EE-enabled web, application, and messaging servers. The details are as follows:

  Supported JVMs

  - Oracle Hotspot JVM 1.5 to 1.8
  - BEA JRockit 1.5 and 1.6
  - IBM JVM 1.5 to 1.8
  - Open JDK 1.5 to 1.8

  Supported Application Servers

  - WebSphere 7.x, 8.x
  - WebLogic 9.x, 10.x, 12.x
  - JBoss 7.x / EAP / WildFly
  - Apache Tomcat 5.x, 6.x, 7.x
  - GlassFish 3.x and 4.x

Supported Frameworks

- Servlets
- JSPs
- Struts 1.x, 2.x
- Spring MVC

Supported HTTP End Points

- HTTP URL Connection

Supported Web Service End Points

- Axis 1.x, 2.x
- JAX-WS
- JAX-RPC

Supported Databases

- Oracle 8i, 9i, 10g, 11g
- IBM DB2 9.x
- MS SQL Server 2005, 2008, 2012
- Postgres 8.x, 9.x
- MySQL
- HSQLDB

Supported Drivers

- Oracle- Thin
- DB2
- Microsoft SQL Server
- Connector/J
- jTDS - Type4
- JDBC2, JDBC2 EE, JDBC3, JDBC4

- The **eG Business Transaction Monitor (BTM)** can be installed on only those Java containers that use JDK 1.5 or higher

- Do not install the **eG Business Transaction Monitor (BTM)** on a Java container that is already JTM-enabled.
- For complete visibility into the transaction path, make sure that you:
  - BTM-enable each JVM node in the transaction path;
  - Manage each JVM node as a separate component in eG;

1.3 How does the eG BTM Work?

To be able to track the live transactions to a web application, eG Enterprise requires that a special **eG Application Server Agent** be deployed on every JVM node (i.e., web application server instance) through which the transaction travels. The **eG Application Server Agent** is available as a file named **eg_btm.jar** on the eG agent host, which has to be copied to the system hosting the application servers being monitored. You then need to configure the application server with the path to the **eg_btm.jar** file to fully BTM-enable the server. Once this is done, restart the server and then proceed to configure the Java Business Transactions test and Key Java Business Transactions test.

1.3.1 Installing eG BTM on a Generic JVM Node

The steps for deploying an eG BTM on a JVM node will differ based on where the eG agent has been deployed - whether on the JVM node, or on a remote host.

If the eG agent monitoring the JVM node has been deployed on that node itself (which is the agent-based approach), then follow the steps below to BTM-enable that node:

1. Manage the JVM node as a separate component using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple JVM instances are operating on a single node, and you want to BTM-enable al the instances, then you will have to manage each instance as a separate component using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory (on Windows; on Unix, this will be **/opt/egurkha/lib/btm**), you will find the following files:
   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
4. Next, create a new directory under the **<EG_AGENT_INSTALL_DIR>\lib\btm** (on Windows; on Unix, this will be **/opt/egurkha/lib/btm**). Take care to name this directory in the following format: *<Managed_*

Component_ NickName>_ <Managed_ Component_ Port>. For instance, if you have managed the JVM node using the nick name *AppServer1* and the port number *8088*, the new directory under the **btm** directory should be named as *AppServer1_8080*.

5. If you have managed multiple JVM instances running on a single node, then you will have to create multiple sub-directories under the **btm** directory- one each for every instance. Each of these sub-directories should be named after the **Nick name** and **Port number** using which the corresponding instance has been managed in eG.
6. Once the new directory is created, copy the following files from the **btm** directory to the new directory:
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
7. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

8. By default, the**BTMPort** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions** test for that application server, make sure you configure the **BTM** port parameter of the test with this port number.

   Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

9. Finally, save the **btmOther.props** file.
10. Then, proceed to edit the start-up script of the JVM node being monitored, and append the following lines to it:

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
"-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar”
```

For instance, if the .props files had been copied to the **<EG_ AGENT_ INSTALL_ DIR>\lib\btm\AppServer1_8088** directory, the above specification will be:

```
-DEG_PROPS_HOME=<EG_AGENT_INSTALL_DIR>\lib\btm\AppServer1_8088
"-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar”
```

Note:

- The “-javaagent…” entry above should be added as one of the JVM options in the start-up script.
- Note that the above lines will change based on the operating system and the web/web application server being monitored. For example, if the JVM node is operating on Unix, then the above specification will change as follows:

```
-DEG_PROPS_HOME=opt/egurkha/lib/btm/AppServer1_8088
"-javaagent:opt/egurkha/lib/btm/eg_btm.jar”
```

- Also, in Unix environments, when using the agent-based approach, both the agent and the JVM instance will be running using different user privileges. In this situation, by default, the eG BTM logs will not be created. In order to create the same, insert the following entry after the -DEG_PROPS_ HOME specification.

```
-DEG_LOG_HOME=<<Log_File_Path>>
```

For instance, if the .props files had been copied to the **<EG_ AGENT_ INSTALL_ DIR>\lib\btm\AppServer1_8088** directory, and the log files also need to be created in the same directory, the complete specification will be as follows:

```
-DEG_PROPS_HOME=opt/egurkha/lib/btm/AppServer1_8088
```

```
-DEG_LOG_HOME=opt/egurkha/lib/btm/AppServer1_8088
```

```
"-javaagent:opt/egurkha/lib/btm/eg_btm.jar"
```

11. Then, add the **eg_btm.jar** file to the **CLASSPATH** of the JVM node being monitored.
12. Finally, save the file, and restart the JVM node.

If the eG agent has been deployed on a remote host (which is the agentless approach), then follow the steps below to BTM-enable the JVM node:

1. Manage the JVM node as a separate component using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple JVM instances are operating on a single node, and you want to monitor each of those instances, then you will have to manage each instance as a separate component using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_ AGENT_ INSTALL_ DIR> \lib\btm** directory (on Windows; on Unix, this will be **/opt/egurkha/lib/btm**), you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
4. Next, log into the JVM node that is being monitored.
5. Create a new directory named, say **eGBTM**, in any location on that node.
6. Under this directory, create a sub-directory. Take care to name this directory in the following format: *<Managed_ Component_ NickName>_ <Managed_ Component_ Port>* . For instance, if you have managed the JVM node using the nick name *AppServer1* and the port number *8088*, the sub-directory should be named as *AppServer1_8080*.
7. If you have managed multiple instances of that JVM node, then you will have to create multiple sub-directories - one each for every instance. Each of these sub-directories should be named after the *Nick name* and *port number* using which the corresponding instance has been managed in eG.
8. Once the new sub-directory is created, copy all the files from the **btm** directory of the remote agent to the sub-directory on the JVM node:
9. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
# Below property is BTM Server Socket Port, through which eG Agent Communicates
```

```
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

10. By default, the**BTM_Port** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions test** for that application server, make sure you configure the **BTM PORT** parameter of the test with this port number.

    Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

    Note:

    In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

11. Finally, save the **btmOther.props** file.

12. Then, proceed to edit the start-up script of the JVM node being monitored, and append the following lines to it:

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
"-javaagent:<<PATH OF THE LOCAL FOLDER CONTAINING THE eg_btm.jar FILE>>”
```

For instance, if the .props files had been copied to the **C:\eGBTM\AppServer1_8088** directory, the above specification will be:

```
-DEG_PROPS_HOME=C:\AppServer1_8088
```

```
"-javaagent:C:\eGBTM\eg_btm.jar"
```

Note:

- The "-javaagent…" entry above should be added as one of the JVM options in the start-up script.
- Note that the above lines will change based on the operating system and the web/web application server being monitored. For example, if the JVM node is operating on Unix, then the above specification will change as follows:

```
-DEG_PROPS_HOME=opt/eGBTM/AppServer1_8088
```

```
"-javaagent:opt/eGBTM/eg_btm.jar"
```

13. Then, add the **eg_btm.jar** file to the **CLASSPATH** of the JVM node being monitored.
14. Finally, save the file, and restart the JVM node.

1.3.2 Installing eG BTM on an Apache Tomcat Server

The steps for BTM-enabling an Apache Tomcat server will differ based on where the eG agent monitoring that Tomcat server has been deployed - whether on the Tomcat server, or on a remote host.

1.3.2.1 Agent-based Approach to Deploying eG BTM on an Apache Tomcat Server

If an Apache Tomcat Server is running on Windows, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that Tomcat server:

1. Manage the Apache Tomcat server using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple Tomcat server instances are operating on a single host, and you want to BTM-enable all the instances, then you will have to manage each instance as a separate Apache Tomcat server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**

- **btmOther.props**
- **exclude.props**

4. Next, create a new directory under the **<EG_AGENT_INSTALL_DIR>\lib\btm**. Take care to name this directory in the following format: *<Managed_Component_NickName>_<Managed_Component_Port>*. For instance, if you have managed the Tomcat server using the nick name *Tomcat1* and the port number *8080*, the new directory under the **btm** directory should be named as *Tomcat1_8080*.
5. If you have managed multiple Tomcat server instances running on a single host, then you will have to create multiple sub-directories under the **btm** directory- one each for every instance. Each of these sub-directories should be named after the **Nick name** and **Port number** using which the corresponding instance has been managed in eG.
6. Once the new directory is created, copy the following files from the **btm** directory to the new directory. If multiple directories have been created as described by step 5 above, then the files should be copied to each of those directories:
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
7. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

By default, the**BTMPort** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions** test for that application server, make sure you configure the **BTM** port parameter of the test with this port number.

Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

8. Then, you need to configure the Tomcat server with the path to the **eg_btm.jar** and **.props** files. This can be done, in one of the following ways:

   - Through the Tomcat control panel;
   - Through the Tomcat start-up script

9. To use the control panel, do the following:
   - First, open the Tomcat Control Panel.

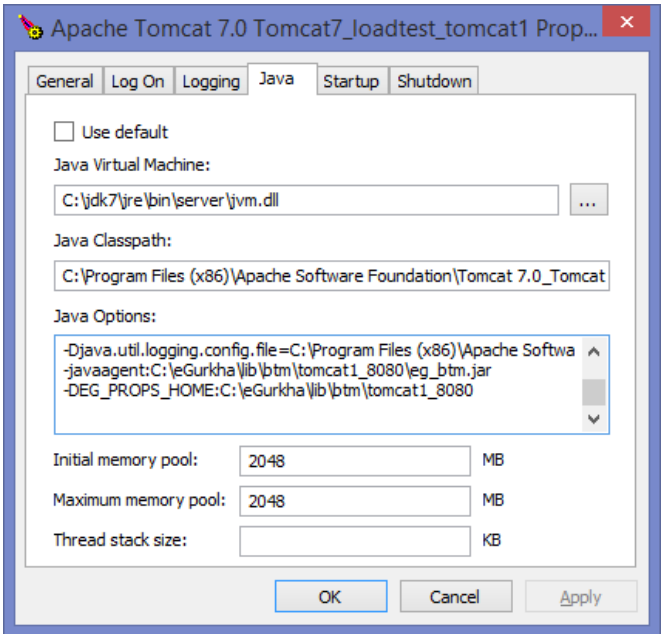


Figure 1.1: BTM-enabling the Tomcat server on Windows

- Select the **Java** tab page in Section **1.3.2** above.
- Add the following entry to the **Java Options** section of 1.3.2:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the .props files had been copied to the **<EG_ AGENT_ INSTALL_ DIR>\lib\btm\tomcat1_8080** directory, the above specification will be:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<EG_AGENT_INSTALL_DIR>\lib\btm\tomcat1_8080
```

- Click the **Apply** and **OK** buttons in 1.3.2.
- Restart the Tomcat service.

10. On the other hand, if you want to configure using the Tomcat start-up script, follow the steps below:
    - Open the **catalina.bat** file from the <TOMCAT_HOME> directory on the Tomcat server.
    - Insert the lines of code indicated by 1.3.2 above to BTM-enable the Tomcat server.

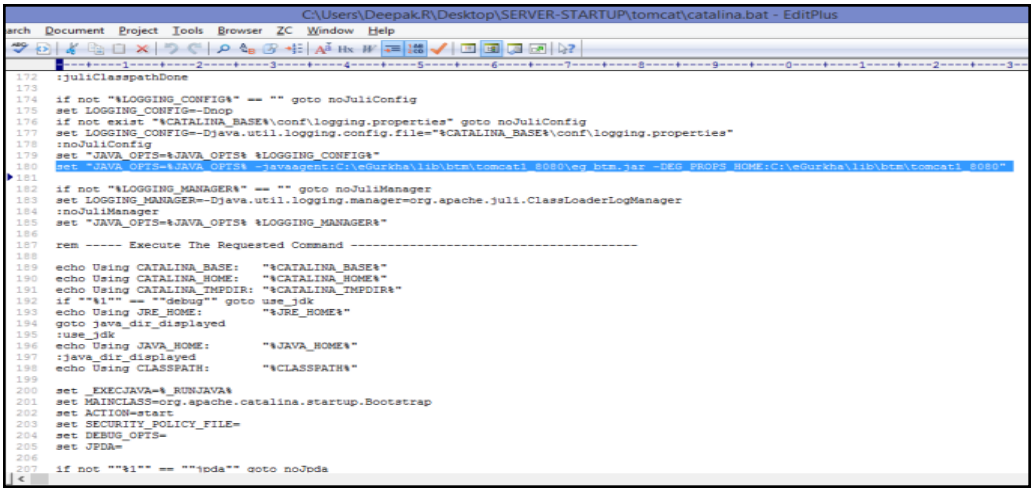


Figure 1.2: Editing the catalina.bat file

- Save the file and restart the Tomcat server.

11. Where multiple Tomcat server instances on a host are to be monitored, repeat 7 to 10 for each of the server instances.

If an Apache Tomcat Server is running on Unix, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that Tomcat server:

1. Manage the Apache Tomcat server using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.

2. If multiple Tomcat server instances are operating on a single host, and you want to BTM-enable all the instances, then you will have to manage each instance as a separate Apache Tomcat server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **/opt/egurkha/lib/btm** directory, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
4. Next, create a new directory under the **/opt/egurkha/lib/btm**. Take care to name this directory in the following format: *<Managed_Component_NickName>_<Managed_Component_Port>*. For instance, if you have managed the Tomcat server using the nick name *Tomcat* and the port number *8080*, the new directory under the **btm** directory should be named as *Tomcat_8080*.
5. If you have managed multiple Tomcat server instances running on a single host, then you will have to create multiple sub-directories under the **btm** directory- one each for every instance. Each of these sub-directories should be named after the **Nick name** and **Port number** using which the corresponding instance has been managed in eG.
6. Once the new directory is created, copy the following files from the **btm** directory to the new directory. If multiple directories have been created as described in step 5 above, then the following files should be copied to all directories:

   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
7. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

   By default, the**BTMPort** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions** test for that application server, make sure you configure the **BTM**

port parameter of the test with this port number.

Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

8. Then, you need to configure the Tomcat server with the path to the **eg_btm.jar** and **.props** files. This can be done by editing the start-up script of the Tomcat server. For that, first open the start-up script.
9. Insert the following lines in the script (as depicted by Figure 1.3) to BTM-enable the server.

```
if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:<<PATH TO THE eg_btm.jar>> -DEG_PROPS_
HOME=<<PATH TO LOCAL FOLDER CONTAINING THE
.PROPS FILES>>
fi
```

For instance, if the .props file had been copied to the **Tomcat_8080** folder within the **/opt/egurkha/lib/btm** folder, then your specification will be as follows:

```
if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:/opt/egurkha/lib/btm/eg_btm.jar -DEG_PROPS_
HOME=/opt/egurkha/lib/btm/Tomcat_8080
fi
```

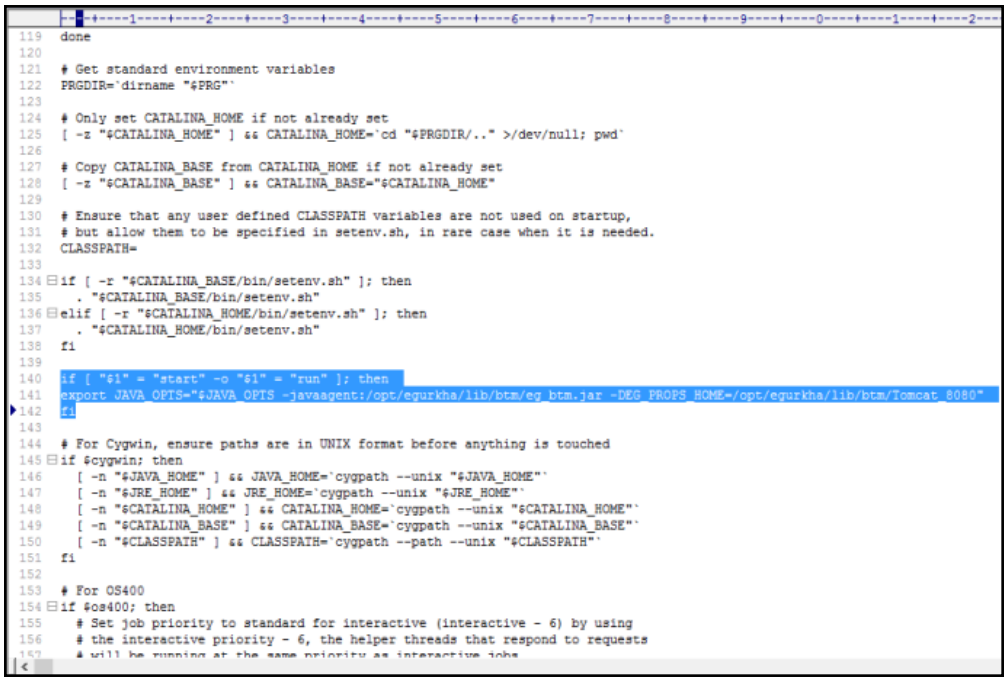


Figure 1.3: Editing the start-up script of a Tomcat server on Linux to BTM-enable the server

10. In Unix environments, if the eG agent is deployed on the same host as the Tomcat server, then both the agent and the server will be running using different user privileges. In this situation, by default, the eG BTM logs will not be created. In order to create the same, insert the following entry after the -DEG_PROPS_HOME specification and before the closing quotes .

```
-DEG_LOG_HOME=<LogFile_Path>
```

For instance, if the .props files have been copied to the **/opt/egurkha/lib/btm/Tomcat_8080** directory, and the BTM log files also need to be created in the same directory, then your complete specification will be as follows:

```
if [ "$1" = "start" -o "$1" = "run" ]; then
```

```
export JAVA_OPTS="$JAVA_OPTS -javaagent:/opt/egurkha/lib/btm/eg_btm.jar -DEG_PROPS_
HOME=/opt/egurkha/lib/btm/Tomcat_8080 -DEG_LOG_HOME=/opt/egurkha/lib/btm/Tomcat_8080
```

```
fi
```

11. Finally, save the file and restart the Tomcat server.
12. Where multiple Tomcat server instances on a host are to be monitored, repeat steps 7 to 11 for each of the server instances.

1.3.2.2 Agentless Approach to Deploying eG BTM on an Apache Tomcat Server

If an Apache Tomcat Server is running on Windows, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that Tomcat server:

1. Manage the Apache Tomcat server as a separate component using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple Tomcat instances are operating on a single node, and you want to monitor each of those instances, then you will have to manage each instance as a separate Apache Tomcat server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory (on Windows; on Unix, this will be the **/opt/egurkha/lib/btm** directory) on the eG agent host, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
4. Next, log into the Tomcat server that is being monitored.
5. Create a new directory named, say **btm**, in any location on that server.
6. Under this directory, create a sub-directory. Take care to name this directory in the following format: *<Managed_ Component_ NickName>_ <Managed_ Component_ Port>* . For instance, if you have managed the Tomcat server using the nick name *tomcat1* and the port number *8080*, the sub-directory should be named as *tomcat1_8080*.
7. If you have managed multiple instances of the Tomcat server, then you will have to create multiple sub-directories - one each for every instance. Each of these sub-directories should be named after the *Nick name* and *port number* using which the corresponding instance has been managed in eG.
8. Once the new sub-directory is created, copy all the files from the **btm** directory of the remote agent to the sub-directory on the Tomcat server. Where multiple sub-directories have been created, you will have to copy the files to each of those directories.
9. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
```

```
#
```

10. By default, the**BTM_Port** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions test** for the Tomcat server, make sure you configure the **BTM PORT** parameter of the test with this port number.

    Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

    Note:

    In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

11. Finally, save the **btmOther.props** file.
12. Then,proceed to configure the Tomcat server with the path to the **eg_btm.jar** and **.props** files. This can be done, in one of the following ways:

    - Through the Tomcat control panel;
    - Through the Tomcat start-up script

13. To use the control panel, do the following:
    - First, open the Tomcat Control Panel.

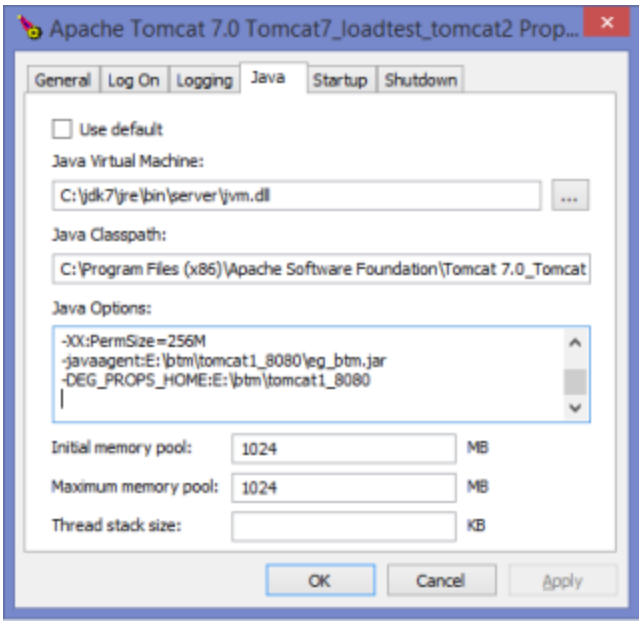


Figure 1.4: BTM-enabling the Tomcat server on Windows in an agentless manner

14. Select the **Java** tab page in Figure 1.4 above.
15. Add the following entry to the **Java Options** section of Figure 1.4:

```
-javaagent:<<PATH OF THE LOCAL FOLDER CONTAINING THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the jar file and .props files had been copied to the **E:\btm\tomcat1_8080** directory, the above specification will be:

```
-javaagent:E:\btm\tomcat1_8080\eg_btm.jar
```

```
-DEG_PROPS_HOME=E:\btm\tomcat1_8080
```

16. Click the **Apply** and **OK** buttons in 1.3.2.
17. Restart the Tomcat service.
18. On the other hand, if you want to configure using the Tomcat start-up script, follow the steps below:
    - Open the **catalina.bat** file from the <TOMCAT_HOME> directory on the Tomcat server.
    - Insert the lines of code indicated by Figure 1.5 above to BTM-enable the Tomcat server.

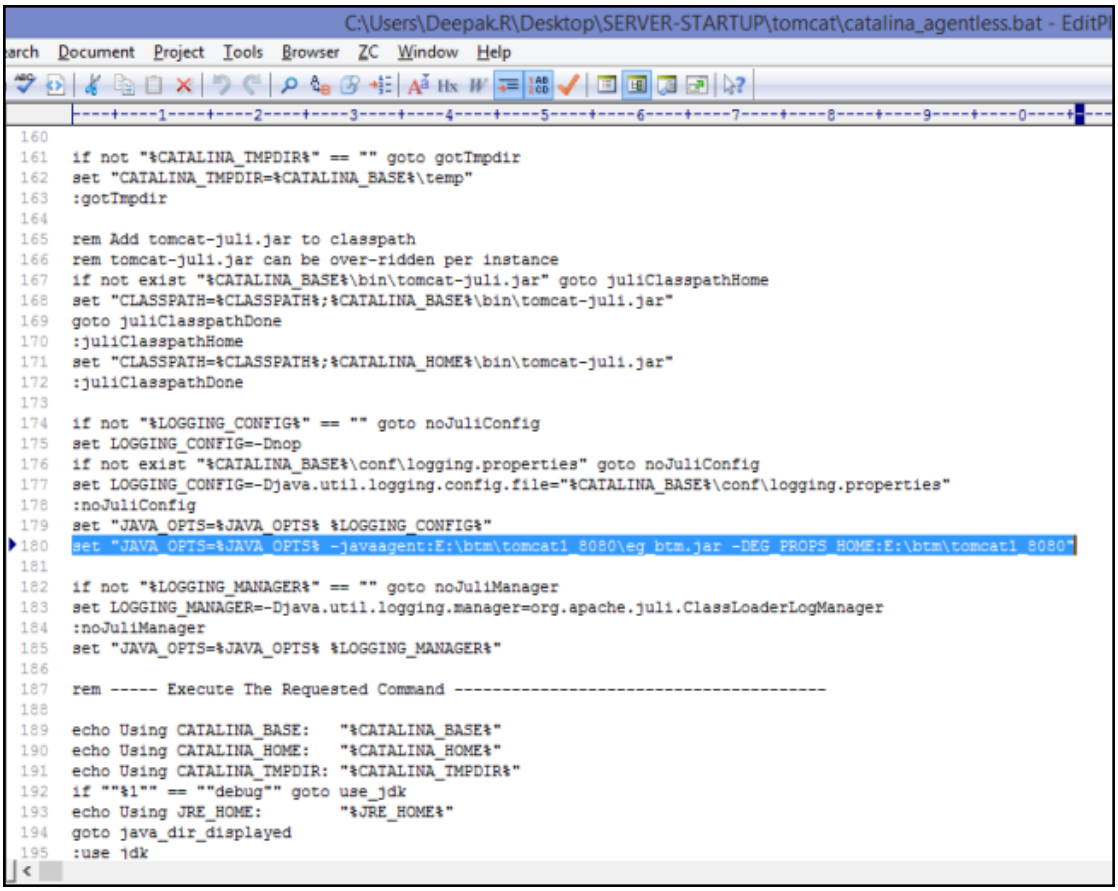


Figure 1.5: Editing the catalina.bat file of a Tomcat server on Windows that is monitored in an agentless manner

- Save the file and restart the Tomcat server.

19. Where multiple Tomcat server instances on a host are to be monitored, repeat steps 9 to 18 for each of the server instances.

If an Apache Tomcat Server is running on Unix, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that Tomcat server:

1. Manage the Apache Tomcat server using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple Tomcat server instances are operating on a single host, and you want to BTM-enable all the instances, then you will have to manage each instance as a separate Apache Tomcat server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory (on Windows; on Unix, this will be the **/opt/egurkha/lib/btm** directory) on the eG agent host, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**

- **btmOther.props**
- **exclude.props**

4. Next, log into the Tomcat server that is being monitored.
5. Create a new directory named, say **btm**, in any location on that server.
6. Under this directory, create a sub-directory. Take care to name this directory in the following format: *<Managed_ Component_ NickName>_ <Managed_ Component_ Port>* . For instance, if you have managed the Tomcat server using the nick name *Tomcat* and the port number *8080*, the sub-directory should be named as *Tomcat_8080*.
7. If you have managed multiple instances of the Tomcat server, then you will have to create multiple sub-directories - one each for every instance. Each of these sub-directories should be named after the *Nick name* and *port number* using which the corresponding instance has been managed in eG.
8. Once the new sub-directory is created, copy all the files from the **btm** directory of the remote agent to the sub-directory on the Tomcat server. Where multiple sub-directories have been created, you will have to copy the files to each of those directories.
9. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

By default, the**BTM_Port** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions test** for the Tomcat server, make sure you configure the **BTM PORT** parameter of the test with this port number.

10. Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
```

```
Designated_Agent=
#
```

Note:

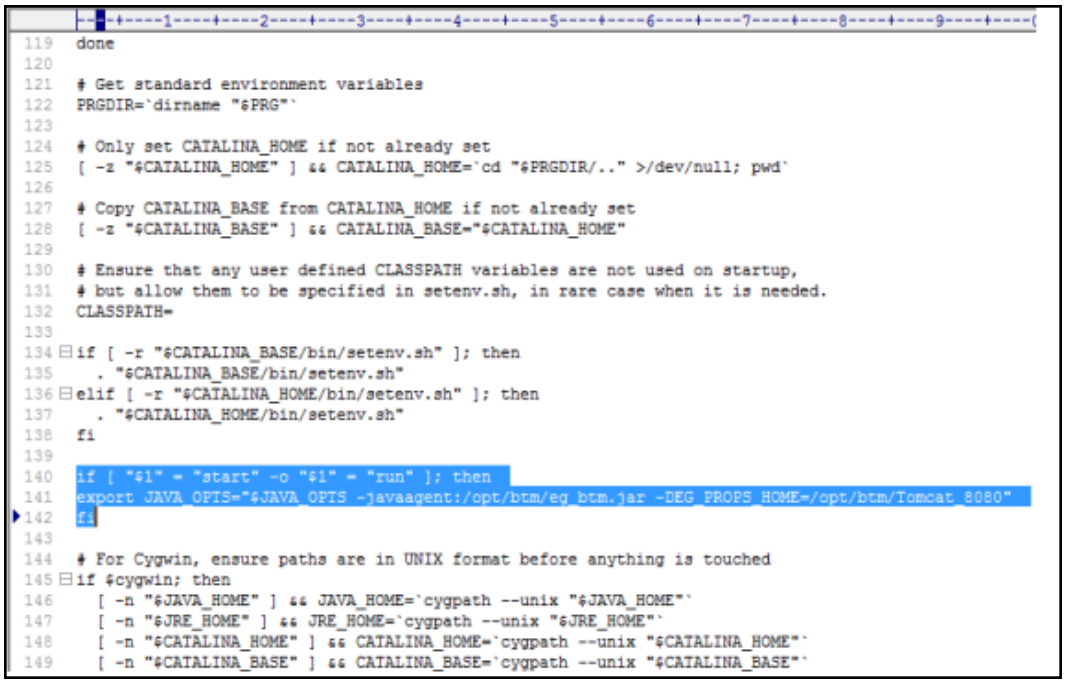
In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

11. Finally, save the **btmOther.props** file.
12. Then, proceed to configure the Tomcat server with the path to the **eg_btm.jar** and **.props** files. For this, you need to edit the start-up script of Tomcat. The first step to achieving that is to open the start-up script file.
13. Insert the following lines in the file, as depicted by Figure 1.6.

```
if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:<<PATH TO THE eg_btm.jar>> -DEG_PROPS_
HOME=<<PATH TO THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>"
fi
```

For instance, if the eg_btm.jar and .props files were copied to the **/opt/btm/Tomcat_8080** directory on the Tomcat server, then your specification will be as follows:

```
if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:/opt/btm/Tomcat_8080/eg_btm.jar -DEG_PROPS_
HOME=/opt/btm/Tomcat_8080"
fi
```



```
done

# Get standard environment variables
PRGDIR=`dirname "$PRG"`

# Only set CATALINA_HOME if not already set
[ -z "$CATALINA_HOME" ] && CATALINA_HOME=`cd "$PRGDIR/.." >/dev/null; pwd`

# Copy CATALINA_BASE from CATALINA_HOME if not already set
[ -z "$CATALINA_BASE" ] && CATALINA_BASE="$CATALINA_HOME"

# Ensure that any user defined CLASSPATH variables are not used on startup,
# but allow them to be specified in setenv.sh, in rare case when it is needed.
CLASSPATH=

if [ -r "$CATALINA_BASE/bin/setenv.sh" ]; then
  . "$CATALINA_BASE/bin/setenv.sh"
elif [ -r "$CATALINA_HOME/bin/setenv.sh" ]; then
  . "$CATALINA_HOME/bin/setenv.sh"
fi

if [ "$1" = "start" -o "$1" = "run" ]; then
export JAVA_OPTS="$JAVA_OPTS -javaagent:/opt/btm/eg_btm.jar -DEG_PROPS_HOME=/opt/btm/Tomcat_8080"
fi

# For Cygwin, ensure paths are in UNIX format before anything is touched
if $cygwin; then
  [ -n "$JAVA_HOME" ] && JAVA_HOME=`cygpath --unix "$JAVA_HOME"`
  [ -n "$JRE_HOME" ] && JRE_HOME=`cygpath --unix "$JRE_HOME"`
  [ -n "$CATALINA_HOME" ] && CATALINA_HOME=`cygpath --unix "$CATALINA_HOME"`
  [ -n "$CATALINA_BASE" ] && CATALINA_BASE=`cygpath --unix "$CATALINA_BASE"`
```

Figure 1.6: Editing the start-up script to BTM-enable a Tomcat server on Linux in an agentless manner

14. Finally, save the file and restart the Tomcat server.
15. Where multiple Tomcat server instances on a host are to be monitored, repeat steps 9 to 14 for each of the server instances.

1.3.3 Installing eG BTM on an IBM WebSphere

The steps for BTM-enabling an IBM WebSphere server will differ based on where the eG agent monitoring that WebSphere server has been deployed - whether on the WebSphere server, or on a remote host.

1.3.3.1 Agent-based Approach to BTM-Enabling IBM WebSphere

If an IBM WebSphere server is running on Windows, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that WebSphere server:

1. Manage the WebSphere server using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple WebSphere server instances are operating on a single host, and you want to BTM-enable all the instances, then you will have to manage each instance as a separate WebSphere server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory, you will find the following files:

- **eg_btm.jar**
- **btmLogging.props**
- **btmOther.props**
- **exclude.props**

4. Next, create a new directory under the **<EG_AGENT_INSTALL_DIR>\lib\btm**. Take care to name this directory in the following format: *<Managed_Component_NickName>_<Managed_Component_Port>*. For instance, if you have managed the WebSphere server using the nick name *Websphere1* and the port number *9080*, the new directory under the **btm** directory should be named as *Websphere1_9080*.
5. If you have managed multiple WebSphere server instances running on a single host, then you will have to create multiple sub-directories under the **btm** directory- one each for every instance. Each of these sub-directories should be named after the **Nick name** and **Port number** using which the corresponding instance has been managed in eG.
6. Once the new directory is created, copy the following files from the **btm** directory to the new directory. If multiple directories have been created as described in step 5 above, then the following files should be copied to all directories:
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
7. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

By default, the**BTMPort** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions** test for the WebSphere server, make sure you configure the **BTM** port parameter of the test with this port number.

Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

8. Then, you need to configure the WebSphere server with the path to the **eg_btm.jar** and **.props** files. For this, first login to the WebSphere administration console. When Figure 1.7 appears, click on the **WebSphere Application Server** link in the right panel.

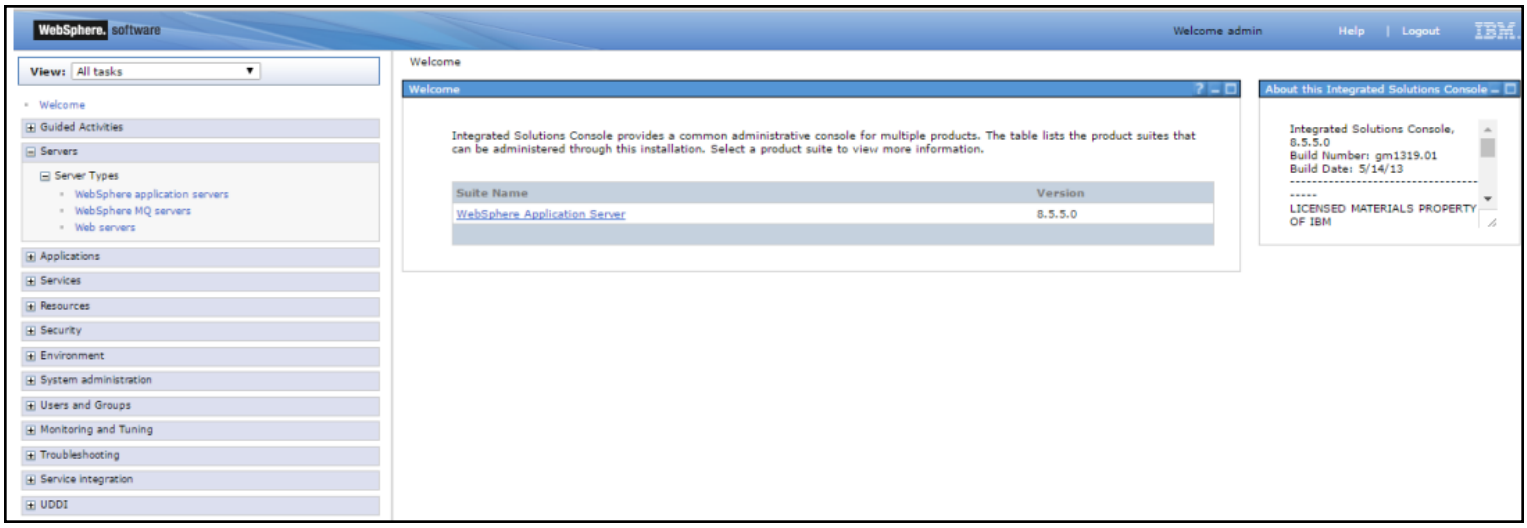


Figure 1.7: The WebSphere Administration console

9. This will invoke Figure 1.8. In the right panel of Figure 1.8, click on the link representing the WebSphere server instance that you want to BTM-enable.

Figure 1.8: Clicking on the WebSphere server instance to be BTM-enabled

10. Figure 1.9 will then appear.

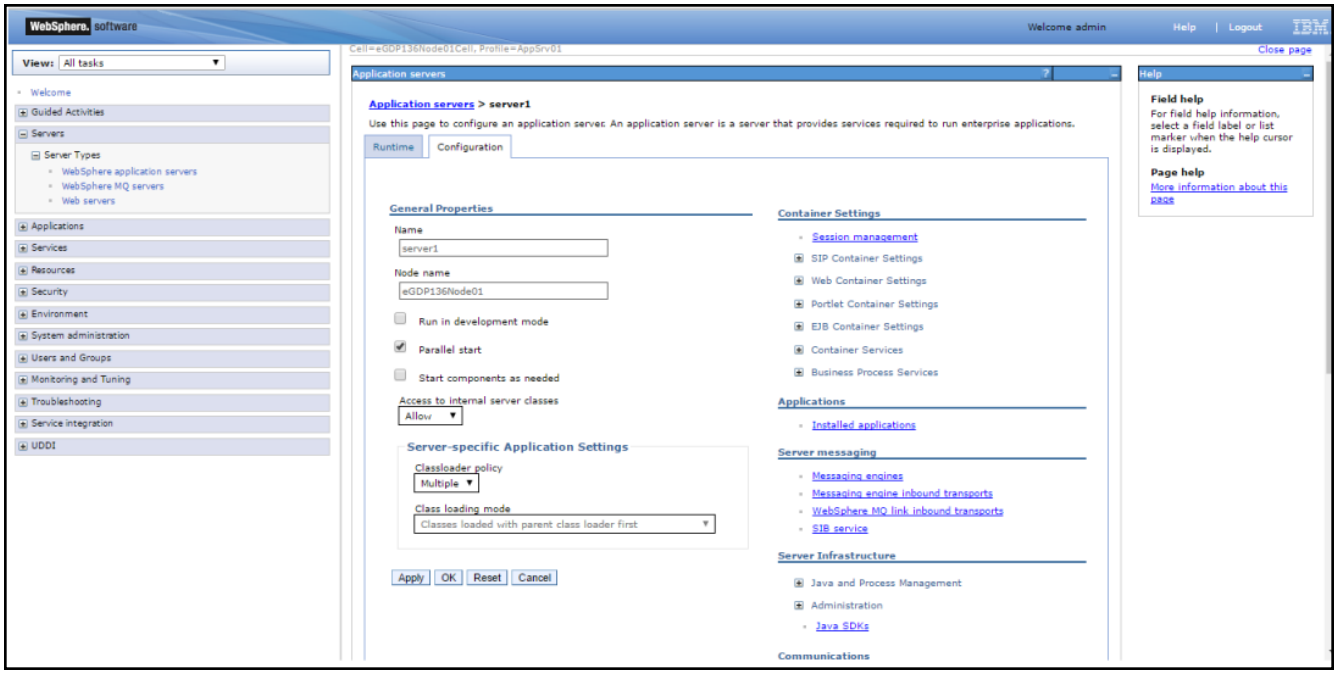


Figure 1.9: The Configuration tab page of the WebSphere server instance to be BTM-enabled

11. Keep scrolling down the right panel of Figure 1.10 until you find the **Server Infrastructure** section. Expand the **Java and Process Management** node in that section, and click on the **Process definition** link within.

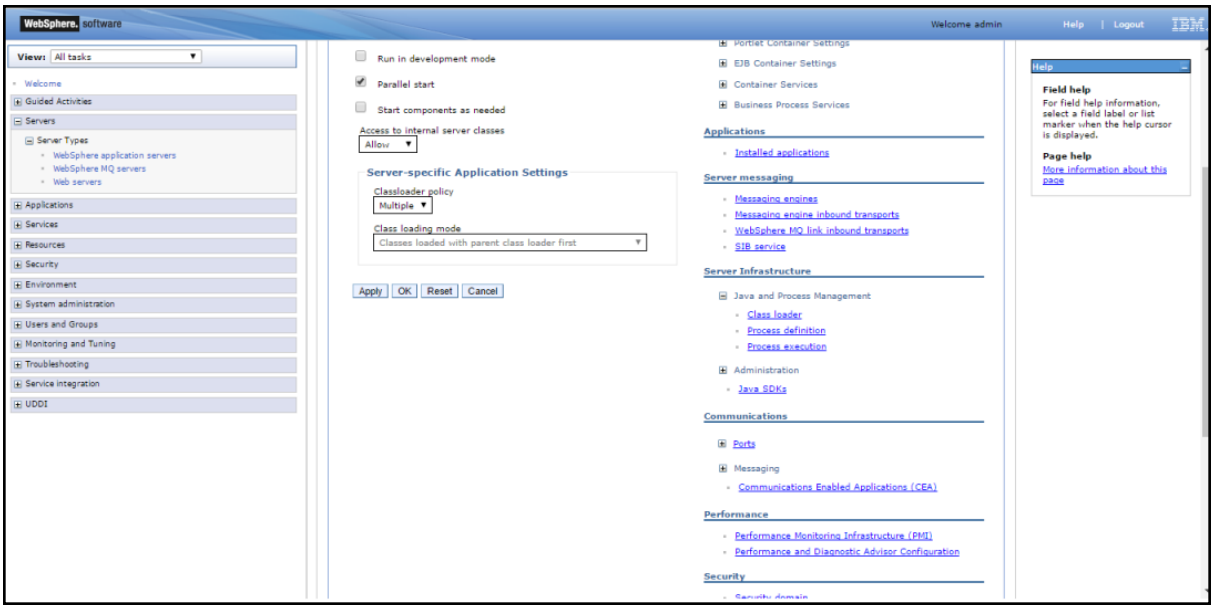


Figure 1.10: Selecting the Process definition option from Java and Process Management tree

12. Figure 1.11 will then appear. From the **Additional Properties** section, select **Java Virtual Machines**.

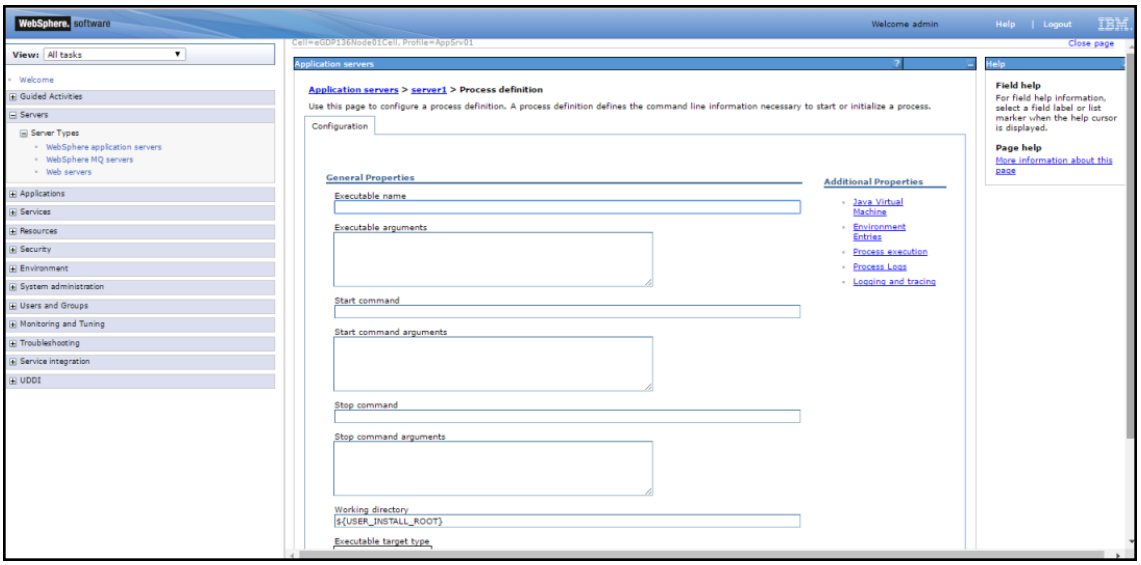


Figure 1.11: Configuring the Process definition

13. When Figure 1.12 appears, scroll down its right panel until the **Generic JVM Arguments** text box comes into view.

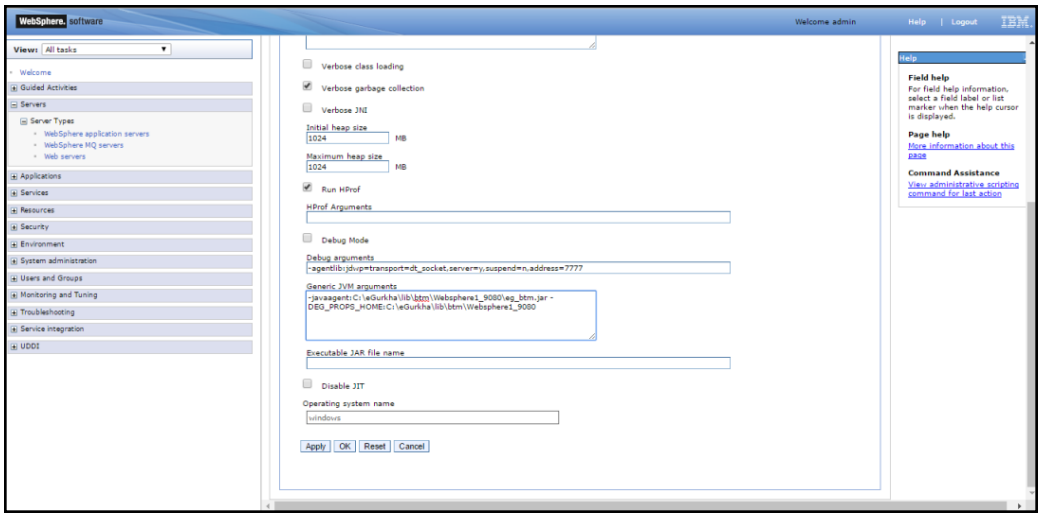


Figure 1.12: Configuring the JVM arguments

14. Here, specify the following:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the .props files had been copied to the **<EG_ AGENT_ INSTALL_ DIR>\lib\btm\Websphere1_9080** directory, the above specification will be:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<EG_AGENT_INSTALL_DIR>\lib\btm\Websphere1_9080
```

15. Save the changes and restart the WebSphere server.

If an IBM WebSphere server is running on Unix, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that WebSphere server:

1. Follow the steps 1-13 above. When doing so, note that the **eg_btm.jar** and the .props files will be available in the **/opt/egurkha/lib/btm** directory on the Unix host.
2. In the **Generic JVM Arguments** text box mentioned in step 13 (see Figure 1.12), specify the following:

```
-javaagent:<<PATH TO THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the .props files had been copied to the **opt/egurkha/lib/btm/Websphere1_9080** directory, the above specification will be:

```
-javaagent:opt/egurkha/lib/btm/eg_btm.jar
```

```
-DEG_PROPS_HOME=opt/egurkha/lib/btm/WebSphere1_9080
```

3. In Unix environments, if the eG agent is deployed on the same host as the WebSphere server, then both the agent and the server will be running using different user privileges. In this situation, by default, the eG BTM logs will not be created. In order to create the same, insert the following entry after the -DEG_PROPS_HOME specification .

```
-DEG_LOG_HOME=<LogFile_Path>
```

For instance, if the .props files have been copied to the **/opt/egurkha/lib/btm/Websphere1_9080** directory, and the BTM log files also need to be created in the same directory, then your complete **Generic JVM Arguments** specification will be as follows:

```
-javaagent:opt/egurkha/lib/btm/eg_btm.jar
-DEG_PROPS_HOME=opt/egurkha/lib/btm/WebSphere1_9080
-DEG_LOG_HOME=opt/egurkha/lib/btm/WebSphere1_9080
```

4. Save the file and restart the WebSphere server.

1.3.3.2 Agentless Approach to BTM-Enabling an IBM WebSphere server

If an IBM WebSphere server is running on Windows, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that WebSphere server:

1. Manage the WebSphere server as a separate component using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple WebSphere server instances are operating on a single node, and you want to monitor each of those instances, then you will have to manage each instance as a separate WebSphere server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory (on Windows; on Unix, this will be **/opt/egurkha/lib/btm**) of the eG agent host, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**

4. Next, log into the WebSphere server that is being monitored.
5. Create a new directory named, say **btm**, in any location on that server.

6. Under this directory, create a sub-directory. Take care to name this directory in the following format: <*Managed_ Component_ NickName*>_ <*Managed_ Component_ Port*>. For instance, if you have managed the WebSphere server using the nick name *Websphere1* and the port number *9080*, the sub-directory should be named as *Websphere1_9080*.
7. If you have managed multiple instances of the WebSphere server, then you will have to create multiple sub-directories - one each for every instance. Each of these sub-directories should be named after the *Nick name* and *port number* using which the corresponding instance has been managed in eG.
8. Once the new sub-directory is created, copy all the files from the **btm** directory of the remote agent to the sub-directory on the Websphere server. Where multiple sub-directories have been created, you will have to copy the files to each of those directories.
9. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

10. By default, the**BTM_Port** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions test** for the WebSphere server, make sure you configure the **BTM PORT** parameter of the test with this port number.

    Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

    Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

11. Finally, save the **btmOther.props** file.
12. Then, proceed to configure the WebSphere server with the path to the **eg_btm.jar** and **.props** files. To achieve this, follow steps 8 - 13 detailed in Section **1.0.1.1** above. This will lead you to the **Generic JVM Arguments** text box of Figure 1.12.

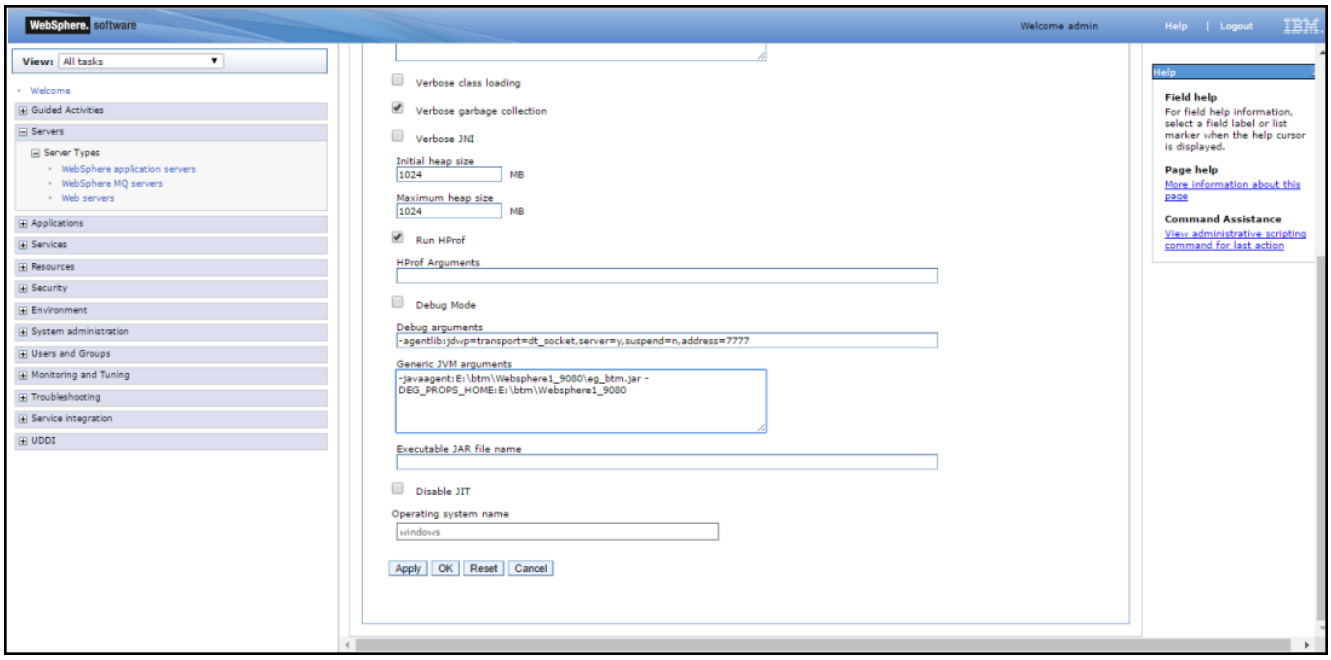


Figure 1.13: Configuring the JVM arguments

13. Here, specify the following:

```
-javaagent:<<PATH OF THE LOCAL FOLDER CONTAINING THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the jar file and .props files had been copied to the **E:\btmWebsphere1_9080** directory, the above specification will be:

```
-javaagent:E:\btm\Websphere1_9080\eg_btm.jar
```

```
-DEG_PROPS_HOME=E:\btm\Websphere1_9080
```

14. Finally, save the changes and restart the WebSphere server.

If an IBM WebSphere server is running on Unix, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that WebSphere server:

1. Follow the steps 1-12 above.
2. In the **Generic JVM Arguments** text box mentioned in step 12 , specify the following:

```
-javaagent:<<PATH TO THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the jar and .props files had been copied to the **opt/btm/WebSphere1_9080** directory, the above specification will be:

```
-javaagent:opt/btm/WebSphere1_9080/eg_btm.jar
```

```
-DEG_PROPS_HOME=opt/btm/WebSphere1_9080
```

3. Save the changes and restart the WebSphere server.

1.3.4 Installing eG BTM on an Oracle WebLogic Server

The steps for BTM-enabling an Oracle WebLogic server will differ based on where the eG agent monitoring that server has been deployed - whether on the WebLogic server, or on a remote host.

1.3.4.1 Agent-based Approach to BTM-Enabling Oracle WebLogic Server

If an Oracle WebLogic server is running on Windows, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that WebLogic server:

1. Manage the WebLogic server using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple WebLogic server instances are operating on a single host, and you want to BTM-enable all the instances, then you will have to manage each instance as a separate WebLogic server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**

4. Next, create a new directory under the **<EG_AGENT_INSTALL_DIR>\lib\btm**. Take care to name this directory in the following format: *<Managed_Component_NickName>_<Managed_Component_Port>*.

For instance, if you have managed the WebLogic server using the nick name *WebLogic1* and the port number *9080*, the new directory under the **btm** directory should be named as *WebLogic1_9080*.

5. If you have managed multiple WebLogic server instances running on a single host, then you will have to create multiple sub-directories under the **btm** directory- one each for every instance. Each of these sub-directories should be named after the **Nick name** and **Port number** using which the corresponding instance has been managed in eG.
6. Once the new directory is created, copy the following files from the **btm** directory to the new directory. If multiple directories have been created as described in step 5 above, then the following files should be copied to all directories:
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
7. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

By default, the**BTMPort** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions** test for the WebLogic server, make sure you configure the **BTM** port parameter of the test with this port number.

Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
```

```
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

8. Then, you need to configure the WebLogic server with the path to the **eg_btm.jar** and **.props** files. To achieve this, you can use one of the following two ways:
   - If you want to BTM-enable a single WebLogic server instance, then use the WebLogic Administration console for this purpose.
   - If you want to BTM-enable the Admin server of a WebLogic cluster, then use the start-up script of the Admin server for this purpose
9. To use the WebLogic Administration console, first login to the console. Then, follow the steps detailed below:
   - When Figure 1.14 appears, click on the **Servers** link in the right panel.

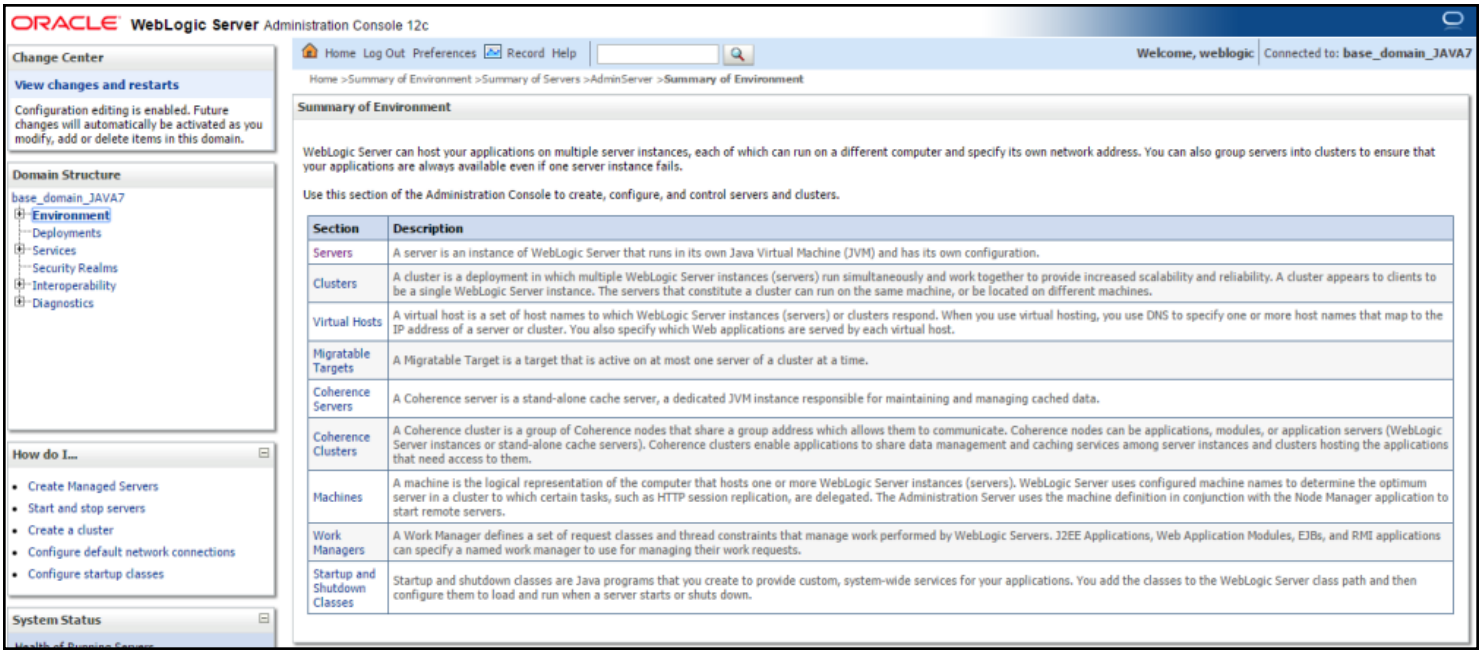


Figure 1.14: Clicking on the Servers link

- Figure 1.15 will then appear.

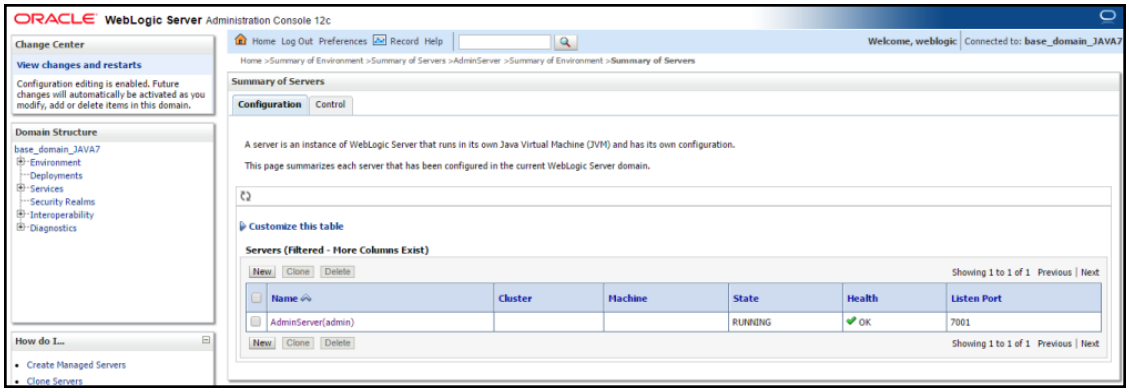


Figure 1.15: Clicking on the server instance to be BTM-enabled

- Figure 1.16 will then appear.

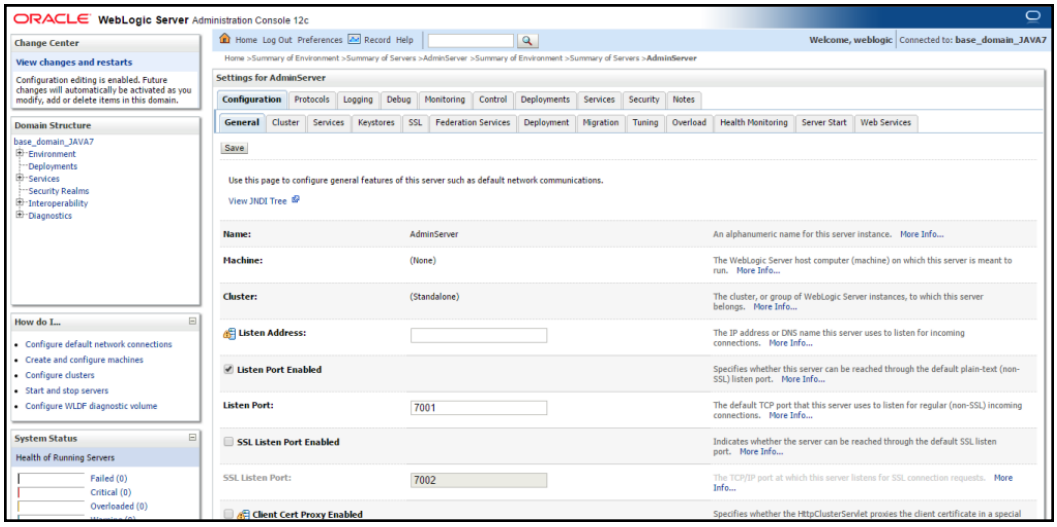


Figure 1.16: Viewing the configuration of the chosen server instance

- Keep scrolling down the right panel of Figure 1.16 until the Arguments text box comes into view (see Figure 1.17).

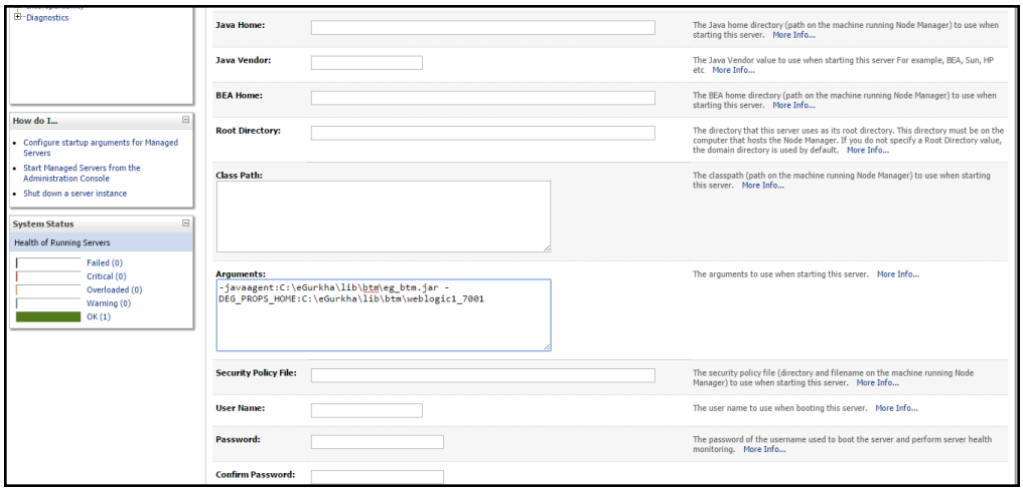


Figure 1.17: Configuring the JVM arguments

- In the **Arguments** text box, specify the following lines:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the .props files had been copied to the **<EG_ AGENT_ INSTALL_ DIR>\lib\btm\weblogic1_7001** directory, the above specification will be:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<EG_AGENT_INSTALL_DIR>\lib\btm\weblogic_7001
```

- Finally, save the changes and restart the WebLogic server.

10. To edit the start-up script of the Admin server of the WebLogic cluster, then follow the steps below:

- Login to the Admin server, open the start-up script, and insert the following lines in it:

```
set EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_
btm.jar -DEG_PROPS_HOME=<<PATH TO THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>"
```

```
if "%SERVER_NAME%"=="AdminServer" (
```

```
set EG_JAVA_OPTIONS=%EG_JAVA_OPTIONS_ADMIN_SERVER%
```

```
)
```

```
set JAVA_OPTIONS=%JAVA_OPTIONS% %JAVA_PROPERTIES% -
Dwlw.iterativeDev=%iterativeDevFlag% -Dwlw.testConsole=%testConsoleFlag% -
Dwlw.logErrorsToConsole=%logErrorsToConsoleFlag% %EG_JAVA_OPTIONS%
```

For instance, if the .props files had been copied to the **C:\eGurkha\lib\btm\WebLogic_7001** directory, the above specification will be:

```
set EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:c:\eGurkha\lib\btm\eg_btm.jar -DEG_PROPS_HOME=c:\eGurkha\lib\btm\WebLogic_7001"

if "%SERVER_NAME%"=="AdminServer" (

set EG_JAVA_OPTIONS=%EG_JAVA_OPTIONS_ADMIN_SERVER%

)

set JAVA_OPTIONS=%JAVA_OPTIONS% %JAVA_PROPERTIES% -Dwlw.iterativeDev=%iterativeDevFlag% -Dwlw.testConsole=%testConsoleFlag% -Dwlw.logErrorsToConsole=%logErrorsToConsoleFlag% %EG_JAVA_OPTIONS%
```

```
294        ) else (
295              set POST_CLASSPATH=%ARDIR%\xqrl.jar
296        )
297  )
298
299  @REM PROFILING SUPPORT
300
301  set JAVA_PROFILE=
302
303  set SERVER_CLASS=weblogic.Server
304
305  set JAVA_PROPERTIES=%JAVA_PROPERTIES% %WLP_JAVA_PROPERTIES%
306
307
308
309  set EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:c:\egurkha\lib\btm\eg_btm.jar -DEG_PROPS_HOME=c:\egurkha\lib\btm\WebLogic_7001"
310
311  if "%SERVER_NAME%"=="AdminServer" (
312        set EG_JAVA_OPTIONS=%EG_JAVA_OPTIONS_ADMIN_SERVER%
313  )
314  set JAVA_OPTIONS=%JAVA_OPTIONS% %JAVA_PROPERTIES% -Dwlw.iterativeDev=%iterativeDevFlag% -Dwlw.testConsole=%testConsoleFlag%
     -Dwlw.logErrorsToConsole=%logErrorsToConsoleFlag% %EG_JAVA_OPTIONS%
315
316
317  if "%PRODUCTION_MODE%"=="true" (
318        set JAVA_OPTIONS= -Dweblogic.ProductionModeEnabled=true %JAVA_OPTIONS%
319  )
320
321  @REM -- Setup properties so that we can save stdout and stderr to files
322
323  if NOT "%WLS_STDOUT_LOG%"=="" (
324        echo Logging WLS stdout to %WLS_STDOUT_LOG%
325        set JAVA_OPTIONS=%JAVA_OPTIONS% -Dweblogic.Stdout=%WLS_STDOUT_LOG%
326  )
327
328  if NOT "%WLS_STDERR_LOG%"=="" (
329        echo Logging WLS stderr to %WLS_STDERR_LOG%
330        set JAVA_OPTIONS=%JAVA_OPTIONS% -Dweblogic.Stderr=%WLS_STDERR_LOG%
331  )
332
```

Figure 1.18: Editing the start-up script of the WebLogic Admin server on Windows that is monitored in an agent-based manner

- Finally, save the file and restart the Admin server.

If an Oracle WebLogic server is running on Unix, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that WebSphere server:

1. If you want to BTM-enable a single WebLogic server instance, then first, follow the steps 1-9 above, until the **Arguments** text box comes into view. When doing so, note that the **eg_btm.jar** and the .props files will be available in the **/opt/egurkha/lib/btm** directory on the Unix host.
2. In the **Arguments** text box mentioned in step 9 (see Figure 1.17), specify the following:

```
-javaagent:<<PATH TO THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the .props files had been copied to the **opt/egurkha/lib/btm/WebLogic_7001** directory, the above specification will be:

```
-javaagent:opt/egurkha/lib/btm/eg_btm.jar
```

```
-DEG_PROPS_HOME=opt/egurkha/lib/btm/WebLogic_7001
```

3. In Unix environments, if the eG agent is deployed on the same host as the WebLogic server, then both the agent and the server will be running using different user privileges. In this situation, by default, the eG BTM logs will not be created. In order to create the same, insert the following entry after the -DEG_PROPS_HOME specification .

```
-DEG_LOG_HOME=<LogFile_Path>
```

For instance, if the .props files have been copied to the **/opt/egurkha/lib/btm/WebLogic1_9080** directory, and the BTM log files also need to be created in the same directory, then your complete **Arguments** specification will be as follows:

```
-DEG_PROPS_HOME=opt/egurkha/lib/btm/WebLogic_7001
```

```
-javaagent:opt/egurkha/lib/btm/eg_btm.jar
```

```
-DEG_LOG_HOME=opt/egurkha/lib/btm/WebLogic_7001
```

4. Save the changes and then restart the WebLogic server instance.
5. On the other hand, if you want to BTM-enable an Admin server (of a WebLogic cluster) on Unix, then follow steps 1-7 above. Then, jump to step 10. As instructed by step 10, edit the start-up script of the Admin server, and insert the following lines in it:

```
EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:/opt/egurkha/lib/btm/eg_btm.jar -DEG_PROPS_
HOME=<<PATH TO THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>"
```

```
if [ "${SERVER_NAME}" = "AdminServer" ] ; then
```

```
EG_JAVA_OPTIONS="${EG_JAVA_OPTIONS_ADMIN_SERVER}"
```

```
fi
```

```
SAVE_JAVA_OPTIONS="${JAVA_OPTIONS} ${EG_JAVA_OPTIONS}"
```

For instance, if the .props files have been copied to the **/opt/egurkha/lib/btm/WebLogic_7001** directory on the Unix host, then your specification will be:

```
EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:/opt/egurkha/lib/btm/eg_btm.jar -DEG_PROPS_
HOME=/opt/egurkha/lib/btm/WebLogic_7001"
if [ "${SERVER_NAME}" = "AdminServer" ] ; then
EG_JAVA_OPTIONS="${EG_JAVA_OPTIONS_ADMIN_SERVER}"
fi
SAVE_JAVA_OPTIONS="${JAVA_OPTIONS} ${EG_JAVA_OPTIONS}"
```

6. Here again, to create the log files, insert the following entry after the -DEG_PROPS_HOME specification and before the closing quotes.

```
-DEG_LOG_HOME=<LogFile_Path>
```

For instance, if the .props files have been copied to the **/opt/egurkha/lib/btm/WebLogic_7001** directory on the Admin server host and the log files also need to be created in the same directory, then your specification will be:

```
EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:/opt/egurkha/lib/btm/eg_btm.jar -DEG_PROPS_
HOME=/opt/egurkha/lib/btm/WebLogic_7001 -DEG_LOG_HOME=/opt/egurkha/lib/btm/WebLogic_
7001"
if [ "${SERVER_NAME}" = "AdminServer" ] ; then
EG_JAVA_OPTIONS="${EG_JAVA_OPTIONS_ADMIN_SERVER}"
fi
SAVE_JAVA_OPTIONS="${JAVA_OPTIONS} ${EG_JAVA_OPTIONS}"
```

7. Finally, save the file and restart the WebLogic Admin server.

1.3.4.2 Agentless Approach to BTM-Enabling an Oracle WebLogic server

If an Oracle WebLogic server is running on Windows, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that WebLogic server:

1. Manage the WebLogic server as a separate component using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple WebLogic server instances are operating on a single node, and you want to monitor each of those instances, then you will have to manage each instance as a separate WebLogic server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory (on Windows; on Unix, this will be **/opt/egurkha/lib/btm**) of the eG agent host, you will find the following files:

- **eg_btm.jar**
- **btmLogging.props**
- **btmOther.props**
- **exclude.props**

4. Next, log into the WebLogic server that is being monitored.
5. Create a new directory named, say **btm**, in any location on that server.
6. Under this directory, create a sub-directory. Take care to name this directory in the following format: *<Managed_ Component_ NickName>_ <Managed_ Component_ Port>* . For instance, if you have managed the WebLogic server using the nick name *weblogic1* and the port number *7001*, the sub-directory should be named as *weblogic_7001*.
7. If you have managed multiple instances of the WebLogic server, then you will have to create multiple sub-directories - one each for every instance. Each of these sub-directories should be named after the *Nick name* and *port number* using which the corresponding instance has been managed in eG.
8. Once the new sub-directory is created, copy all the files from the **btm** directory of the remote agent to the sub-directory on the WebLogic server. Where multiple sub-directories have been created, you will have to copy the files to each of those directories.
9. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

10. By default, the**BTM_Port** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions test** for the WebLogic server, make sure you configure the **BTM PORT** parameter of the test with this port number.

    Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

11. Finally, save the **btmOther.props** file.
12. Then, proceed to configure the WebLogic server with the path to the **eg_btm.jar** and **.props** files. To achieve this, you can use one of the following two ways:
    - If you want to BTM-enable a single WebLogic server instance, then use the WebLogic Administration console for this purpose.
    - If you want to BTM-enable the Admin server of a WebLogic cluster, then use the start-up script of the Admin server for this purpose
13. To use the WebLogic Administration console, first login to the console. Then, follow the steps detailed in step 9 of the previous section, until you get to the step where the **Arguments** text box comes into view. Here, provide the entry depicted by Figure 1.19 below.

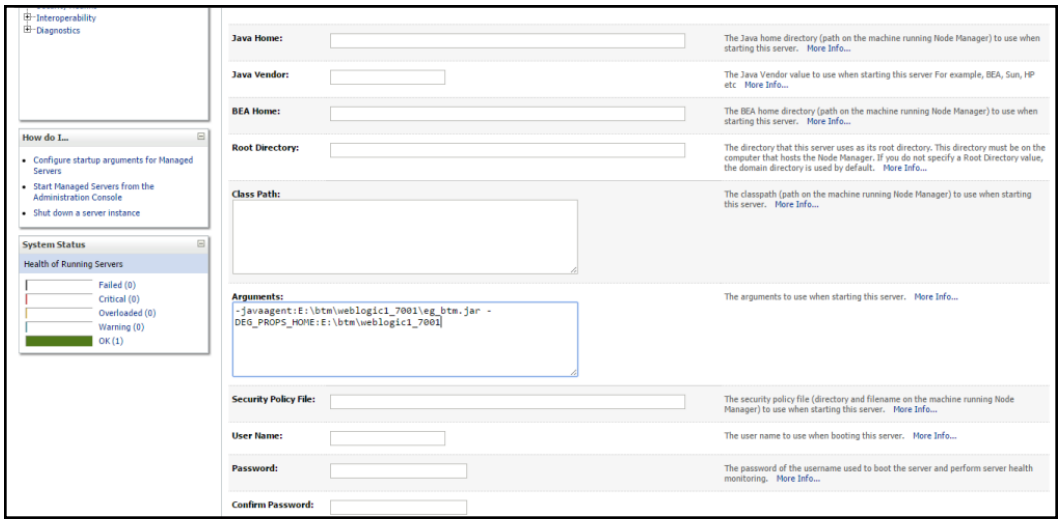


Figure 1.19: Configuring the JVM arguments

Here, specify the following:

```
-javaagent:<<PATH OF THE LOCAL FOLDER CONTAINING THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the jar file and .props files had been copied to the **E:\btm\weblogic_7001** directory, the above specification will be:

```
-javaagent:E:\btm\weblogic_7001\eg_btm.jar
```

```
-DEG_PROPS_HOME=E:\btm\weblogic_7001
```

14. Finally, save the changes and restart the WebLogic server.
15. To BTM-enable the Admin server of a WebLogic cluster, edit the start-up script of the Admin server. For that, follow the steps below:
    - Open the start-up script and insert the following lines in it:

```
set EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:<<PATH TO THE eg_btm.jar FILE>> -DEG_
PROPS_HOME=<<PATH TO THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>"
```

```
if "%SERVER_NAME%"=="AdminServer" (
```

```
set EG_JAVA_OPTIONS=%EG_JAVA_OPTIONS_ADMIN_SERVER%
```

```
)
```

```
set JAVA_OPTIONS=%JAVA_OPTIONS% %JAVA_PROPERTIES% -
Dwlw.iterativeDev=%iterativeDevFlag% -Dwlw.testConsole=%testConsoleFlag% -
Dwlw.logErrorsToConsole=%logErrorsToConsoleFlag% %EG_JAVA_OPTIONS%
```

For instance, if the jar file and the .props file had been copied to the **C:\btm\WebLogic_7001** directory, the above specification will be:

```
set EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:c:\btm\WebLogic_7001\eg_btm.jar -DEG_
PROPS_HOME=c:\btm\WebLogic_7001"
```

```
if "%SERVER_NAME%"=="AdminServer" (
```

```
set EG_JAVA_OPTIONS=%EG_JAVA_OPTIONS_ADMIN_SERVER%
```

```
)
```

```
set JAVA_OPTIONS=%JAVA_OPTIONS% %JAVA_PROPERTIES% -
Dwlw.iterativeDev=%iterativeDevFlag% -Dwlw.testConsole=%testConsoleFlag% -
Dwlw.logErrorsToConsole=%logErrorsToConsoleFlag% %EG_JAVA_OPTIONS%
```

```
        ) else (
                set POST_CLASSPATH=%ARDIR%\xqrl.jar
        )
)

@REM PROFILING SUPPORT

set JAVA_PROFILE=

set SERVER_CLASS=weblogic.Server

set JAVA_PROPERTIES=%JAVA_PROPERTIES% %WLP_JAVA_PROPERTIES%


set EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:-javaagent:c:\btm\WebLogic_7001\eg_btm.jar -DEG_PROPS_HOME=c:\btm\WebLogic_7001"

if "%SERVER_NAME%"=="AdminServer" (
        set EG_JAVA_OPTIONS=%EG_JAVA_OPTIONS_ADMIN_SERVER%
)
set JAVA_OPTIONS=%JAVA_OPTIONS% %JAVA_PROPERTIES% -Dwlw.iterativeDev=%iterativeDevFlag% -Dwlw.testConsole=%testConsoleFlag%
-Dwlw.logErrorsToConsole=%logErrorsToConsoleFlag% %EG_JAVA_OPTIONS%


if "%PRODUCTION_MODE%"=="true" (
        set JAVA_OPTIONS= -Dweblogic.ProductionModeEnabled=true %JAVA_OPTIONS%
)

@REM -- Setup properties so that we can save stdout and stderr to files

if NOT "%WLS_STDOUT_LOG%"=="" (
        echo Logging WLS stdout to %WLS_STDOUT_LOG%
        set JAVA_OPTIONS=%JAVA_OPTIONS% -Dweblogic.Stdout=%WLS_STDOUT_LOG%
)

if NOT "%WLS_STDERR_LOG%"=="" (
        echo Logging WLS stderr to %WLS_STDERR_LOG%
        set JAVA_OPTIONS=%JAVA_OPTIONS% -Dweblogic.Stderr=%WLS_STDERR_LOG%
)
```

Figure 1.20: Editing the start-up script of the WebLogic Admin server on Windows that is monitored in an agentless manner

- Finally, save the file and restart the Admin server.

If an Oracle WebLogic server is running on Unix, and the eG agent monitoring the server has been deployed on a remote host, then follow the steps below to BTM-enable that WebLogic server:

1. To BTM-enable an individual WebLogic server instance, follow the steps 1-13 above, until the **Arguments** text box comes into view.
2. In the **Arguments** text box mentioned in step 13, specify the following:

```
-javaagent:<<PATH TO THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the jar file and the .props files had been copied to the **opt/btm/WebLogic_7001** directory, the above specification will be:

```
-javaagent:opt/btm/eg_btm.jar
```

```
-DEG_PROPS_HOME=opt/btm/WebLogic_7001
```

3. Save the changes and then restart the WebLogic server instance.
4. On the other hand, if you want to BTM-enable an Admin server (of a WebLogic cluster) on Unix, then follow steps 1-12 above. Then, jump to step 15. As instructed by step 15, edit the start-up script of the Admin server, and insert the following lines in it:

```
EG_JAVA_OPTIONS_ADMIN_SERVER="-javaagent:<<PATH TO THE eg_btm.jar FILE>> -DEG_PROPS_
HOME=<<PATH TO THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>"
if [ "${SERVER_NAME}" = "AdminServer" ] ; then
EG_JAVA_OPTIONS="${EG_JAVA_OPTIONS_ADMIN_SERVER}"
fi
SAVE_JAVA_OPTIONS="${JAVA_OPTIONS} ${EG_JAVA_OPTIONS}"
```

For instance, if the jar file and .props files have been copied to the /opt//btm/WebLogic_7001 directory on the Unix host, then your specification will be:

```
EG_ JAVA_ OPTIONS_ ADMIN_ SERVER="- javaagent:/opt/btm/WebLogic_ 7001/eg_ btm.jar  - DEG_
PROPS_HOME=/opt/btm/WebLogic_7001"
if [ "${SERVER_NAME}" = "AdminServer" ] ; then
EG_JAVA_OPTIONS="${EG_JAVA_OPTIONS_ADMIN_SERVER}"
fi
SAVE_JAVA_OPTIONS="${JAVA_OPTIONS} ${EG_JAVA_OPTIONS}"
```

5. Finally, save the file and restart the WebLogic Admin server.

1.3.5 Installing eG BTM on GlassFish

The steps for BTM-enabling GlassFish server will differ based on where the eG agent monitoring that server has been deployed - whether on the GlassFish server, or on a remote host.

1.3.5.1 Agent-based Approach to BTM-Enabling a GlassFish Server

If a GlassFish server is running on Windows, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that GlassFish server:

1. Manage the GlassFish server using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple GlassFish server instances are operating on a single host, and you want to BTM-enable all the instances, then you will have to manage each instance as a separate GlassFish server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory, you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**

- **btmOther.props**
- **exclude.props**

4. Next, create a new directory under the **<EG_AGENT_INSTALL_DIR>\lib\btm**. Take care to name this directory in the following format: *<Managed_Component_NickName>_<Managed_Component_Port>*. For instance, if you have managed the GlassFish server using the nick name *GlassFish1* and the port number *8080*, the new directory under the **btm** directory should be named as *GlassFish1_8080*.
5. If you have managed multiple GlassFish server instances running on a single host, then you will have to create multiple sub-directories under the **btm** directory- one each for every instance. Each of these sub-directories should be named after the **Nick name** and **Port number** using which the corresponding instance has been managed in eG.
6. Once the new directory is created, copy the following files from the **btm** directory to the new directory. If multiple directories have been created as described in step 5 above, then the following files should be copied to all directories:
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**
7. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

By default, the**BTMPort** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions** test for the GlassFish server, make sure you configure the **BTM** port parameter of the test with this port number.

Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

Note:

In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

8. Then, you need to configure the GlassFish server with the path to the **eg_btm.jar** and **.props** files. To achieve this, you can use one of the following two ways:
   - Through the GlassFish Administration console
   - By editing the start-up script of the GlassFish server instance
9. If you choose to use the GlassFish Administration console, then first, login to the console. Then, follow the steps detailed below:
   - When Figure 1.21 appears, click on the **server-config** node in the tree-structure in the left panel.

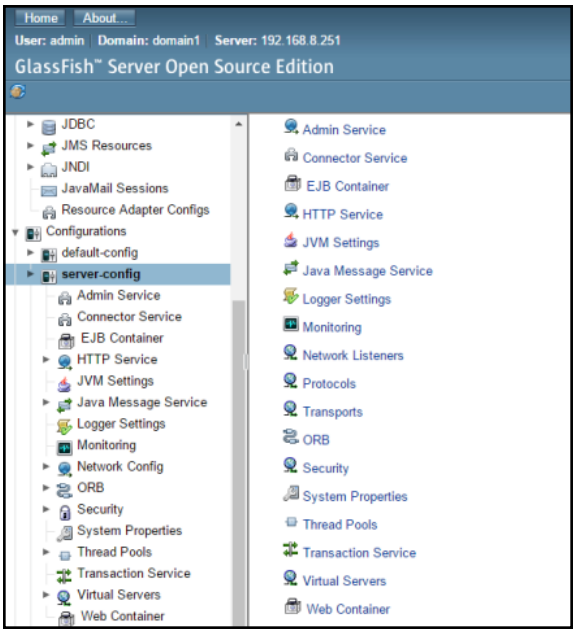


Figure 1.21: Clicking on the server-config node

- From the options listed in the right panel of Figure 1.21, select the **JVM Settings** option. Figure 1.22 will then appear. Select the **JVM Options** tab page in Figure 1.22.

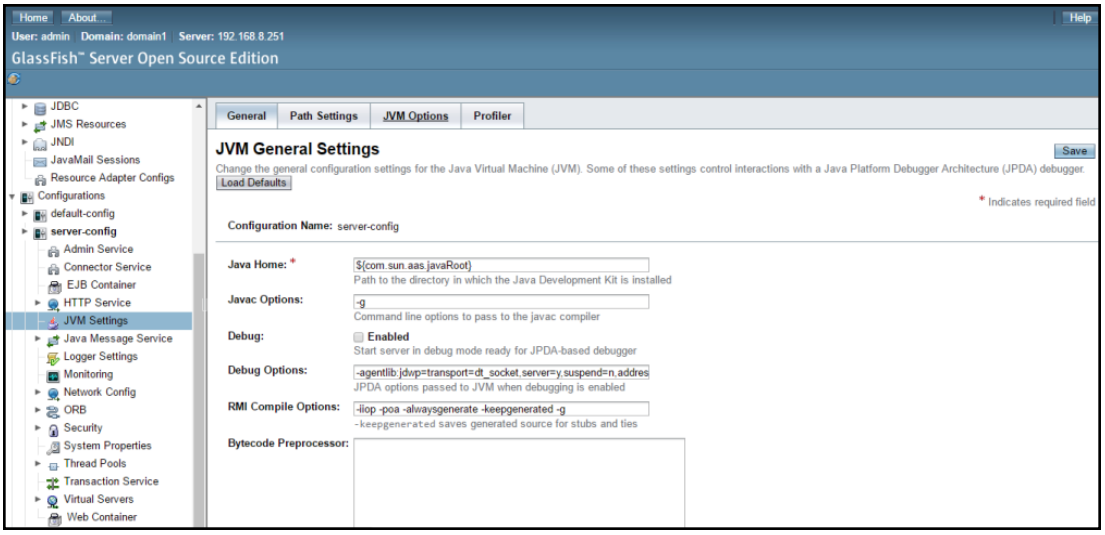


Figure 1.22: Clicking on the JVM Options tab page

- Figure 1.23 will then appear. You now need to add two new JVM options. For this, click on the **Add JVM Option** button in Figure 1.23, twice.

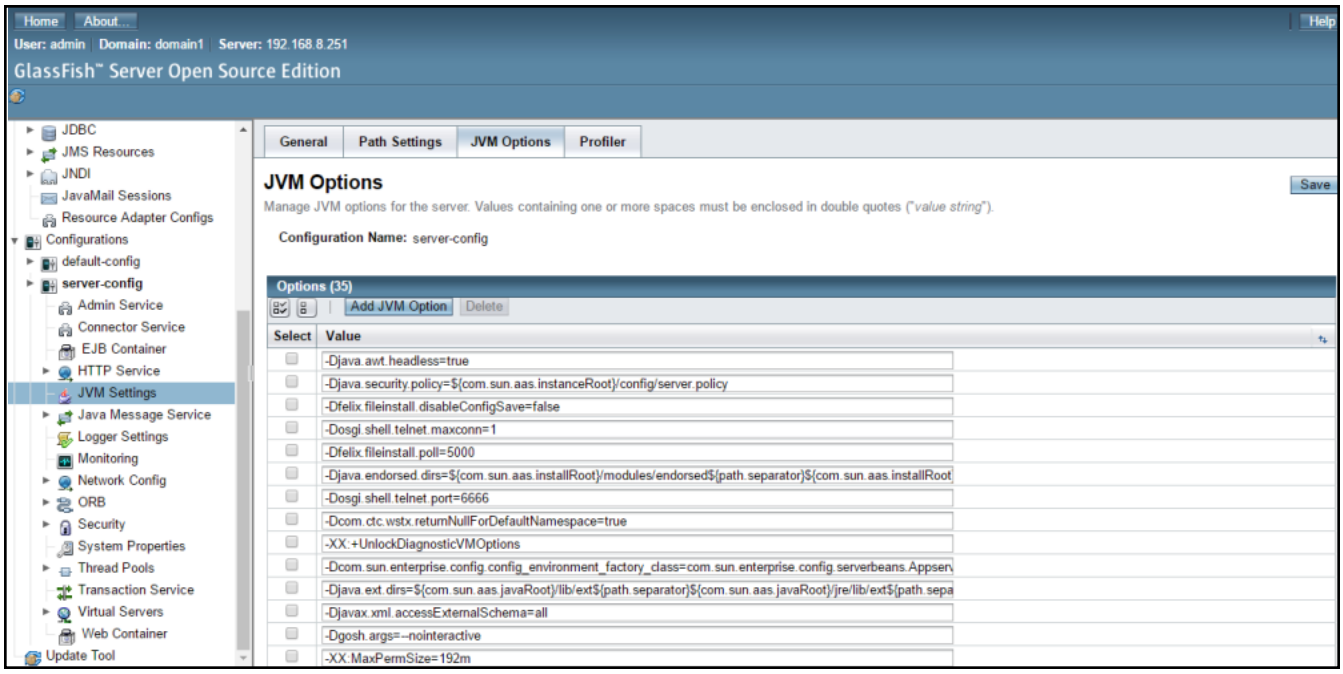


Figure 1.23: Clicking on the ADD JVM Option button

- Two empty rows will then be inserted, as depicted by Figure 1.24.

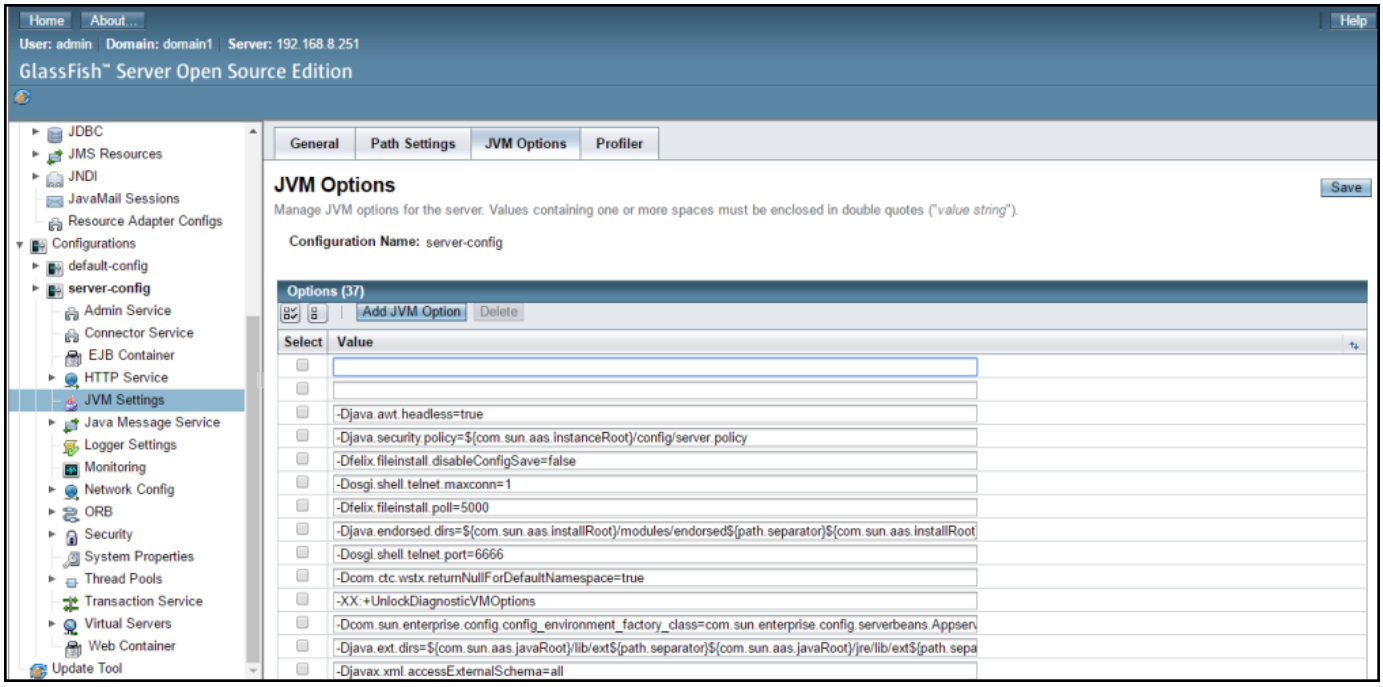


Figure 1.24: Two empty rows inserted in the JVM Options tab page

- Specify each of the following lines in each of the empty rows, as indicated by Figure 1.25:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the .props files had been copied to the **<EG_AGENT_INSTALL_DIR>\lib\btm\GlassFish1_8080** directory, the above specification will be:

```
-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar
```

```
-DEG_PROPS_HOME=<EG_AGENT_INSTALL_DIR>\lib\btm\GlassFish1_8080
```

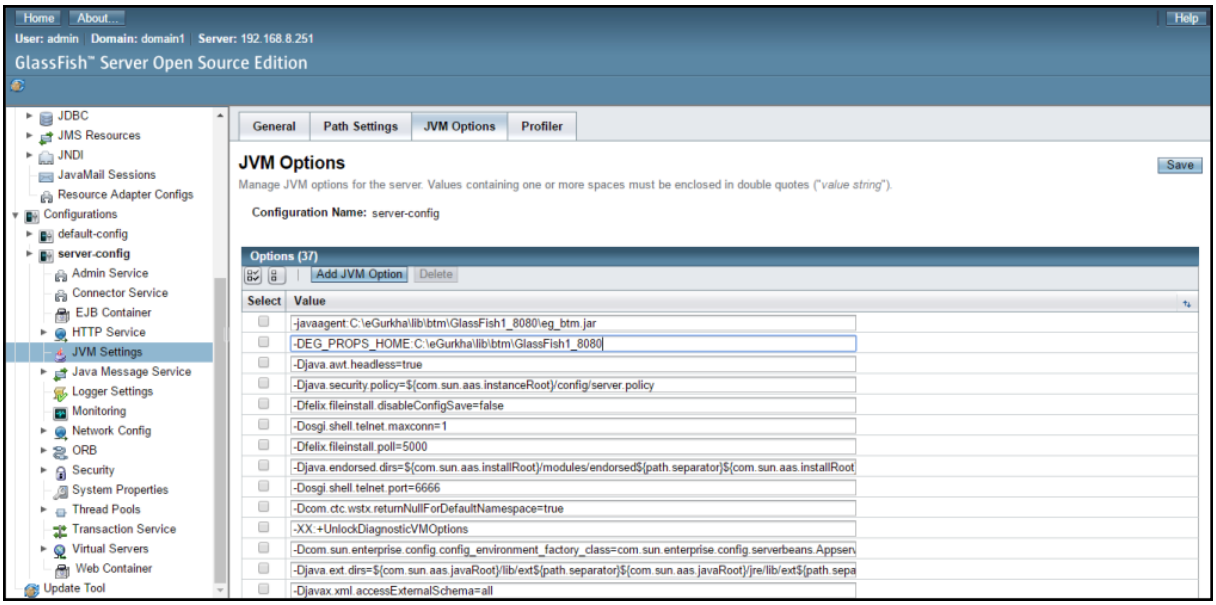


Figure 1.25: Specifying the Java arguments for BTM-enabling the GlassFish server

- Finally, save the changes and restart the GlassFish server.

10. On the other hand, if you want to BTM-enable the GlassFish server by editing the start-up script of the GlassFish server instance, then follow the steps below:

    - Open the start-up script and enter the following lines in it, as depicted by Figure 1.26.

```
<jvm-options>-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar</jvm-options>
```

```
<jvm-options>-DEG_ROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>></jvm-options>
```

For instance, if the .props files had been copied to the **<EG_AGENT_INSTALL_DIR>\lib\btm\GlassFish1_8080** directory, the above specification will be:

```
<jvm-options>-javaagent:<EG_AGENT_INSTALL_DIR>\lib\btm\eg_btm.jar</jvm-options>
```

```
<jvm-options>-DEG_PROPS_HOME=<EG_AGENT_INSTALL_DIR>\lib\btm\GlassFish1_8080</jvm-options>
```

Figure 1.26: Editing the start-up script of the GlassFish server instance to BTM-enable the instance

- Finally, save the file and restart the GlassFish server instance.

If a GlassFish server is running on Unix, and the eG agent monitoring the server has been deployed on that server itself, then follow the steps below to BTM-enable that GlassFish server:

1. Follow step 1 - 7 above. While doing so, note that the jar and .props files will be available in the **/opt/egurkha/lib/btm** directory on the eG agent host.
2. Then, proceed to configure the GlassFish server with the path to the .jar and .props files. For this, you need to edit the start-up script of the GlassFish server.
3. The first step towards this end is to open the start-up script. Then, insert the following lines in it:

```
<jvm-options>-javaagent:opt/egurkha/lib/btm/eg_btm.jar </jvm-options>
```

```
<jvm- options>- DEG_ PROPS_ HOME= <<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>></jvm-options>
```

For instance, if the .props files had been copied to the **/opt/egurkha/lib/btm/GlassFish1_ 8080** directory, the above specification will be:

```
<jvm-options>-javaagent:/opt/egurkha/lib/btm/eg_btm.jar</jvm-options>
```

```
<jvm-options>-DEG_PROPS_HOME=opt/egurkha/lib/btm/GlassFish1_8080</jvm-options>
```

4. In Unix environments, if the eG agent is deployed on the same host as the GlassFish server, then both the agent and the server will be running using different user privileges. In this situation, by default, the eG BTM logs will not be created. In order to create the same, insert the following entry after the -DEG_ PROPS_HOME specification .

```
<jvm-options>-DEG_LOG_HOME=<<LogFile_Path>></jvm-options>
```

For instance, if the .props files have been copied to the **/opt/egurkha/lib/btm/GlassFish1_8080** directory, and the BTM log files also need to be created in the same directory, then your complete specification will be as follows:

```
<jvm-options>-javaagent:/opt/egurkha/lib/btm/eg_btm.jar</jvm-options>
<jvm-options>-DEG_PROPS_HOME=opt/egurkha/lib/btm/GlassFish1_8080</jvm-options>
<jvm-options>-DEG_LOG_HOME=opt/egurkha/lib/btm/GlassFish1_8080</jvm-options>
```

5. Finally, save the file and restart the GlassFish server.

1.3.5.2 Agentless Approach to BTM-Enabling an GlassFish server

If a GlassFish server is running on Windows, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that GlassFish server:

1. Manage the GlassFish server as a separate component using the eG administrative interface. When managing, make a note of the **Nick name** and **Port number** that you provide.
2. If multiple GlassFish server instances are operating on a single node, and you want to monitor each of those instances, then you will have to manage each instance as a separate GlassFish server using the eG administrative interface. When doing so, make a note of the **Nick name** and **Port number** using which you managed each instance.
3. In the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory (on Windows; on Unix, this will be **/opt/egurkha/lib/btm**), you will find the following files:

   - **eg_btm.jar**
   - **btmLogging.props**
   - **btmOther.props**
   - **exclude.props**

4. Next, log into the GlassFish server that is being monitored.
5. Create a new directory named, say **btm**, in any location on that server.
6. Under this directory, create a sub-directory. Take care to name this directory in the following format: *<Managed_Component_NickName>_<Managed_Component_Port>*. For instance, if you have managed the GlassFish server using the nick name *GlassFish1* and the port number *8080*, the sub-directory should be named as *GlassFish1_8080*.
7. If you have managed multiple instances of the GlassFish server, then you will have to create multiple sub-directories - one each for every instance. Each of these sub-directories should be named after the *Nick name* and *port number* using which the corresponding instance has been managed in eG.

8. Once the new sub-directory is created, copy all the files from the **btm** directory of the remote agent to the sub-directory on the GlassFish server. Where multiple sub-directories have been created, you will have to copy the files to each of those directories.
9. Next, edit the **btmOther.props** file. You will find the following lines in the file:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Below property is BTM Server Socket Port, through which eG Agent Communicates
# Restart is required, if any changes in this property
# Default port is "13931"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
BTM_Port=13931
#
```

10. By default, the**BTM_Port** parameter is set to 13931. If you want to enable eG BTM on a different port, then specify the same here. In this case, when configuring the **Java Business Transactions** test or the **Key Java Business Transactions test** for the GlassFish server, make sure you configure the **BTM PORT** parameter of the test with this port number.

    Also, against the **Designated_Agent** parameter, specify the IP address of the eG agent which will poll the eG BTM for metrics. If no IP address is provided here, then the eG BTM will treat the host from which the very first 'measure request' comes in as the **Designated_Agent**.

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
Below property is used to specify IP address of eG Agent which collectes BTM Data.
# Default is None
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
Designated_Agent=
#
```

    Note:

    In case a specific **Designated_Agent** is not provided, and the eG BTM treats the host from which the very first 'measure request' comes in as the **Designated_Agent**, then if such a **Designated_Agent** is stopped or uninstalled for any reason, the eG BTM will wait for a maximum of 10 measure periods for that 'deemed' **Designated_Agent** to request for metrics. If no requests come in for 10 consecutive measure periods, then the eG BTM will begin responding to 'measure requests' coming in from any other eG agent.

11. Finally, save the **btmOther.props** file.
12. Then, proceed to configure the GlassFish server with the path to the **eg_btm.jar** and **.props** files. To achieve this, you can use one of the following two ways:
    - Through the GlassFish Administration console
    - By editing the start-up script of the GlassFish server instance
13. If you choose to use the GlassFish Administration console, then first, login to the console. Then, follow the steps detailed in step 9 of the previous section, until you get to the step where you add two empty rows in the **JVM Options** page.

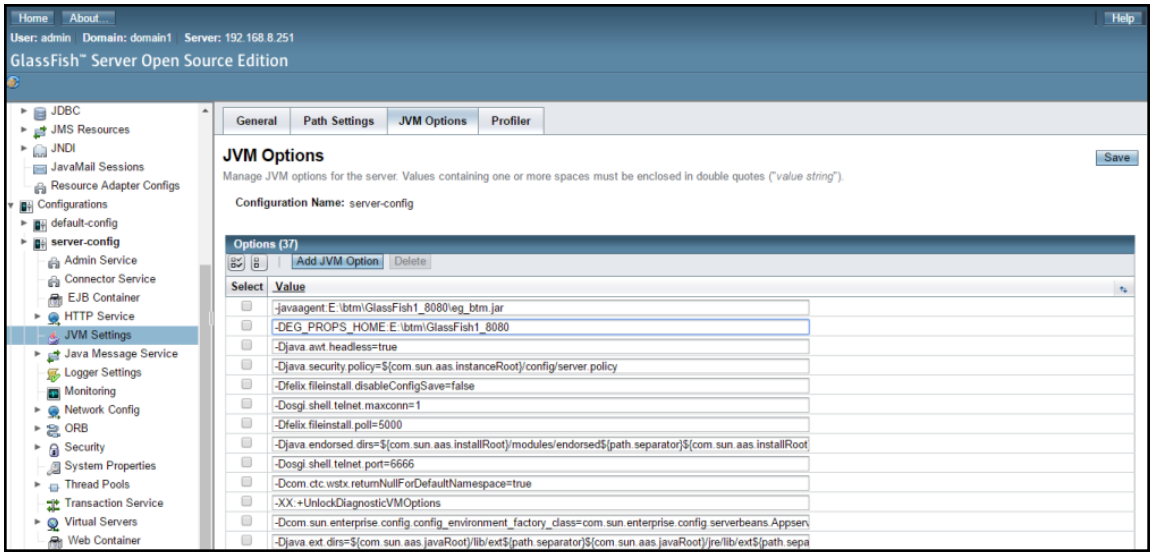


Figure 1.27: Configuring the JVM arguments

In these rows, provide the following entries, as depicted by Figure 1.27.

```
-javaagent:<<PATH OF THE LOCAL FOLDER CONTAINING THE eg_btm.jar FILE>>
```

```
-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>>
```

For instance, if the jar file and .props files had been copied to the **E:\btm\GlassFish1_8080** directory, the above specification will be:

```
-javaagent:E:\btm\GlassFish1_8080\eg_btm.jar
```

```
-DEG_PROPS_HOME=E:\btm\GlassFish1_8080
```

14. Finally, save the changes and restart the GlassFish server.
15. On the other hand, if you want to BTM-enable the GlassFish server by editing the start-up script of the GlassFish server instance, then follow the steps below:
    - Open the start-up script and insert the following lines in it:

```
<jvm-options>-javaagent:<<PATH TO THE eg_btm.jar ON THE LOCAL FOLDER>></jvm-options>
```

```
<jvm-options>-DEG_PROPS_HOME=<<PATH OF THE LOCAL FOLDER CONTAINING THE .PROPS FILES>></jvm-options>
```

For instance, if the jar file and the .props file had been copied to the **E:\btm\weblogic_7001** directory, the above specification will be:

```
<jvm-options>-javaagent:E:\btm\GlassFish1_8080\eg_btm.jar</jvm-options>
```

```
<jvm-options>-DEG_PROPS_HOME=E:\btm\GlassFish1_8080</jvm-options>
```

Figure 1.28: Editing the start-up script of the GlassFish server instance to BTM-enable the instance

- Finally, save the file and restart the GlassFish server instance.

If a GlassFish server is running on Unix, and the eG agent monitoring the server has been deployed on a remote host in the environment, then follow the steps below to BTM-enable that GlassFish server:

1. Follow steps 1 - 11 above.
2. Then, proceed to configure the GlassFish server with the path to the .jar and .props files. For this, you need to edit the start-up script of the GlassFish server.
3. The first step towards this end is to open the start-up script. Then, insert the following lines in it:

```
<jvm-options>-javaagent:<<PATH TO THE eg_btm.jar FILE>></jvm-options>
```

```
<jvm-options>-DEG_PROPS_HOME=<<PATH TO THE LOCAL FOLDER CONTAINING THE .PROPS FILES>></jvm-options>
```

For instance, if the jar file and .props files have been copied to the **/opt/btm/GlassFish1_8080** directory, the above specification will be:

```
<jvm-options>-javaagent:/opt/btm/GlassFish1_8080/eg_btm.jar</jvm-options>
```

```
<jvm-options>-DEG_PROPS_HOME=opt/btm/GlassFish1_8080</jvm-options>
```

4. Finally, save the file and restart the GlassFish server

1.4 Java Business Transactions Test

The responsiveness of a transaction is the key determinant of user experience with that transaction; if response time increases, user experience deteriorates. To make users happy, a Java business transaction should be rapidly processed by each of the JVM nodes in its path. Processing bottlenecks on a single JVM node can slowdown/stall an entire business transaction or can cause serious transaction errors. This in turn can badly scar the experience of users. To avoid this, administrators should promptly identify slow/stalled/errored transactions, isolate the JVM node on which the slowness/error occurred, and uncover what caused the aberration on that node – is it owing to SQL queries executed by the node? Or is it because of external calls – eg., async calls, SAP JCO calls, HTTP calls, etc. - made by that node? The **Java Business Transactions** test helps with this!

This test runs on a BTM-enabled JVM in an IT infrastructure, tracks all the transaction requests received by that JVM, and groups requests based on user-configured pattern specifications. For each transaction pattern, the test then computes and reports the average time taken by that JVM node to respond to the transaction requests of that pattern. In the process, the test identifies the slow/stalled transactions of that pattern, and reports the count of such transactions and their responsiveness. Detailed diagnostics provided by the test accurately pinpoint the exact transaction URLs that are slow/stalled, the total round-trip time of each transaction, and also indicate when such transaction requests were received by that node. The slowest transaction in the group can thus be identified.

Moreover, to enable administrators to figure out if the slowness can be attributed to a bottleneck in SQL query processing, the test also reports the average time the transactions of each pattern took to execute SQL queries. If a majority of the queries are slow, then the test will instantly capture the same and notify administrators.

Additionally, the test promptly alerts administrators to error transactions of each pattern. To know which are the error transactions, the detailed diagnosis capability of the test can be used.

This way, the test effortlessly measures the performance of each transaction to a JVM node, highlights transactions that are under-performing, and takes administrators close to the root-cause of poor transaction performance.

Target of the Test: A BTM-enabled JVM

Agent deploying the test : An internal/remote agent

Output of the test: One set of results for each grouped URL

Test parameters:

1. **TEST PERIOD** - How often should the test be executed
2. **HOST** - The host for which the test is to be configured
3. **BTM PORT** -Specify the port number specified as **BTM_Port** in the **btmOther.props** file on the JVM node being monitored. If the JVM is being monitored in an agent-based manner, then the **btmOther.props** file will be in the **<EG_AGENT_INSTALL_DIR>\lib\btm** directory.
4. **MAX URL SEGMENTS** - This test groups transaction URLs based on the URL segments count configured for monitoring and reports aggregated response time metrics for every group. Using this parameter, you can specify the number of URL segments based on which the transactions are to be grouped.

   URL segments are the parts of a URL (after the base URL) or path delimited by slashes. So if you had the URL: *http://www.eazykart.com/web/shopping/sportsgear/login.jsp*, then *http://www.eazykart.com* will be the base URL or domain,*/web* will be the first URL segment, */shopping*will be the second URL segment, and /sportsgear will be the third URL segment, and*/login.jsp* will be the fourth URL segment. By default, this parameter is set to 3. This default setting, when applied to the sample URL provided above, implies that the eG agent will aggregate response time metrics to all transaction URLs under /web/shopping/sportsgear. Note that the base URL or domain will not be considered when counting URL segments. This in turn means that, if the JVM node receives transaction requests for the URLs such as http://www.eazykart.com/web/shopping/sportsgear/login.jsp, http://www.eazykart.com/web/shopping/sportsgear/jerseys.jsp, http://www.eazykart.com/web/shopping/sportsgear/shoes.jsp, http://www.eazykart.com/web/shopping/sportsgear/gloves.jsp, etc., then the eG agent will track the requests and responses for all these URLs, aggregate the results, and present the aggregated metrics for the descriptor /web/shopping/sportsgear. This way, the test will create different transaction groups based on each of the third-level URL segments – eg. /web/shopping/weddings, /web/shopping/holiday, /web/shopping/gifts etc. – and will report aggregated metrics for each group so created.

   If you want, you can override the default setting by providing a different URL segment number here. For instance, your specification can be just 2 . In this case, for the URL*http://www.eazykart.com/web/shopping/login.jsp*, the test will report metrics for the descriptor *web/shopping*.

5. **EXCLUDED PATTERNS** - By default, this test does not track requests to the following URL patterns:

   .ttf, *.otf, *.woff, *.woff2, *.eot, *.cff, *.afm, *.lwfn, *.ffil, *.fon, *.pfm, *.pfb, *.std, *.pro, *.xsf, *.jpg, *.jpeg, *.jpe, *.jif, *.jfif, *.jfi, *.jp2, *.j2k, *.jpf, *.jpx, *.jpm, *.jxr, *.hdp, *.wdp, *.mj2, *.webp, *.gif, *.png, *.apng, *.mng, *.tiff, *.tif, *.xbm, *.bmp, *.dib, *.svg, *.svgz, *.mpg, *.mpeg, *.mpeg2, *.avi, *.wmv, *.mov, *.rm, *.ram, *.swf, *.flv, *.ogg, *.webm, *.mp4, *.ts, *.mid, *.midi, *.rm, *.ram, *.wma, *.aac, *.wav, *.ogg, *.mp3, *.mp4, *.css, *.js, *.ico|/egurkha

   If required, you can remove one/more patterns from this default list, so that such patterns are monitored, or can append more patterns to this list in order to exclude them from monitoring.

6. **MONITORING MODE** – **Profiler** is the default operational mode for the eG agent that performs business transaction monitoring. In this default mode, the agent collects deep diagnostics of all external calls made by the target JVM node when processing a transaction. This includes POJO calls, which are usually large in number. Since an agent operating in the **Profiler** mode will report response time metrics per POJO call, a marginal increase in the processing overheads of the transaction can be expected in this mode.

   To ensure that the agent balances transaction visibility with low transaction overhead, **Troubleshooting** can be chosen as the **MONITORING MODE**. This mode optimizes agent performance for your live environment. This is why, if you choose this mode, then whenever you attempt to perform transaction execution analysis, eG will reveal the details of all external calls made by the target JVM node for that transaction, except POJO method calls.

7. **METHOD EXEC CUTOFF (MS)**– From the detailed diagnosis of slow/stalled/error transactions, you can drill down and perform deep execution analysis of a particular transaction. In this drill-down, the methods invoked by that slow/stalled/error transaction are listed in the order in which the transaction calls the methods. By configuring a **METHOD EXECUTION CUTOFF**, you can make sure that methods that have been executing for a duration greater the specified cutoff are alone listed when performing execution analysis. For instance, if you specify *5* here, then the **Execution Analysis** window for a slow/stalled/error transaction will list only those methods that have been executing for over 5 milliseconds. This way, you get to focus on only those methods that could have caused the slowness, without being distracted by inconsequential methods. By default, the value of this parameter is set to 250 ms.

8. **SQL EXECUTION CUTOFF (MS)** – Typically, from the detailed diagnosis of a slow/stalled/error transaction on a JVM node, you can drill down to view the SQL queries (if any) executed by that transaction from that node and the execution time of each query. By configuring a **SQL EXECUTION CUTOFF**, you can make sure that queries that have been executing for a duration greater the specified cutoff are alone listed when performing query analysis. For instance, if you specify *5* here, then for a slow/stalled/error transaction, the **SQL Queries** window will display only those queries that have been executing for over 5 milliseconds. This way, you get to focus on only those queries that could have contributed to the slowness. By default, the value of this parameter is set to 10 ms.

9. **HEALTHY URL TRACE** – By default, this flag is set to **No**. This means that eG will not collect detailed diagnostics for those transactions that are healthy. If you want to enable the detailed diagnosis capability for healthy transactions as well, then set this flag to **Yes**.

10. **MAX HEALTHY URLS PER TEST PERIOD** – **This parameter is applicable only if the HEALTHY URL TRACE flag is set to 'Yes'.** Here, specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Healthy transactions* measure, every time the test runs. By default, this is set to 5*0*, indicating that the detailed diagnosis of the *Healthy transactions* measure will by default list the top-50 transactions, arranged in the descending order of their response times.

11. **MAX SLOW URLS PER TEST PERIOD** - Specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Slow transactions* measure, every time the test runs. By default, this is set to *10*, indicating that the detailed diagnosis of the *Slow transactions* measure will by default list the top-10 transactions, arranged in the descending order of their response times.

12. **MAX STALLED URLS PER TEST PERIOD** - Specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Stalled transactions* measure, every time the test runs. By default, this is set to *10*, indicating that the detailed diagnosis of the *Stalled transactions* measure will by default list the top-10 transactions, arranged in the descending order of their response times.
13. **MAX ERROR URLS PER TEST PERIOD** - Specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Error transactions* measure, every time the test runs. By default, this is set to *10*, indicating that the detailed diagnosis of the *Error transactions* measure will by default list the top-10 transactions, in terms of the number of errors they encountered.
14. **ADVANCED SETTINGS** – To optimize transaction performance and conserve space in the eG database, many restraints have been applied by default on the agent's ability to collect and report detailed diagnostics. Depending upon how well-tuned your eG database is and the level of visibility you require into transaction performance, you may choose to either retain these default settings or override them. If you choose not to disturb the defaults, then set the **ADVANCED SETTINGS** flag to **No**. If you want to modify the defaults, then set the **ADVANCED SETTINGS** flag to **Yes**.
15. **POJO METHOD TRACING LIMIT** and **POJO METHOD TRACING CUTOFF TIME** - **These parameters will appear only if the ADVANCED SETTINGS flag is set to 'true'**. Typically, if the **MONITORING MODE** of this test is set to **Profiler** , then, as part of the detailed diagnostics of a transaction, eG reports the execution time of every POJO, non-POJO, and recursive (i.e. methods that call themselves) method call that a JVM node makes when processing that transaction. Of these, POJO method calls are the most expensive, as they are usually large in number. To ensure that attempts made to collect detailed measures related to POJO method calls do not impact the overall responsiveness of the monitored transaction, eG, by default, collects and reports the execution time of only the following POJO method calls:

    - The first 1000 POJO method calls made by the target JVM node for that transaction; (OR)
    - The POJO method calls that were made by the target JVM node within 10 seconds from the start of the monitored transaction on that node;

    Accordingly, the **POJO METHOD TRACING LIMIT** is set to 1000 by default, and the **POJO METHOD TRACING CUTOFF TIME** is set to 10 (seconds) by default. Of these two limits, whichever limit is reached first will automatically be applied by eG for determining when to stop POJO tracing. In other words, once a JVM node starts processing a transaction, the agent begins tracking the POJO method calls made by that node for that transaction. In the process, if the agent finds that the configured tracing limit is reached before the tracing cutoff time is reached, then the agent will stop tracking the POJO method calls, as soon as the tracing limit is reached. On the other hand, if the tracing limit is not reached, then the agent will continue tracking the POJO method calls until the tracing cutoff time is reached. At the end of the cutoff time, the agent will stop tracking the POJO method calls. For instance, if the JVM node makes 1000 POJO method calls within say, 6 seconds from when it began processing the transaction, then the eG agent will not wait for the cutoff time of 10 seconds to be reached; instead, it will stop tracing at the end of the thousandth POJO method call, and report the execution time of each of the 1000 calls alone. On the other hand, if the JVM node does not make over 1000 POJO method calls till the 10 second cutoff expires, then the eG agent continues tracking the POJO method calls till the end of 10 seconds, and reports the details of all those that were calls made till the cutoff time.

Depending upon how many POJO calls you want to trace and how much overhead you want to impose on the agent and on the transaction, you can increase / decrease the **POJO METHOD TRACING LIMIT** and **POJO METHOD TRACING CUTOFF TIME** specifications.

16. **NON-POJO METHOD TRACING LIMIT** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, when reporting the detailed diagnosis of a transaction on a particular JVM node, this test reports the execution time of only the first 1000 non-POJO method calls (which includes JMS, JCO, HTTP, Java, SQL, etc.) that the target JVM node makes for that transaction. This is why, the non-pojo method tracing limit parameter is set to 1000 by default. If you want, you can change the tracing limit to enable the test to report the details of more or fewer non-POJO method calls made by a JVM node. While a high value for this parameter may take you closer to identifying the non-POJO method that could have caused the transaction to slowdown on a particular JVM node, it may also marginally increase the overheads of the transaction and the eG agent.
17. **RECURSIVE METHOD TRACING LIMIT** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. A recursive method is a method that calls itself. By default, when reporting the detailed diagnosis of a transaction on a particular JVM node, this test reports the execution time of only the first 1000 recursive method calls (which includes JMS, JCO, HTTP, Java, SQL, etc.) that the target JVM node makes for that transaction. This is why, the **RECURSIVE METHOD TRACING LIMIT** parameter is set to 1000 by default. If you want, you can change the tracing limit to enable the test to report the details of more or fewer recursive method calls made by a JVM node. While a high value for this parameter may take you closer to identifying the recursive method that could have caused the transaction to slowdown on a particular JVM node, it may also marginally increase the overheads of the transaction and the eG agent.
18. **EXCEPTION STACKTRACE LINES** –**This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. As part of detailed diagnostics, this test, by default, lists the first 10 stacktrace lines of each JavaScript error/exception that it captures on the target JVM node for a specific transaction, so as to enable easy and efficient troubleshooting. This is why, the **EXCEPTION STACKTRACE LINES** parameter is set to 10 by default. If required, you can have this test display more or fewer stacktrace lines by overriding this default setting.
19. **INCLUDED EXCEPTIONS** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, this test flags the transactions in which the following errors/exceptions are captured, as *Error transactions*:

    - All unhandled exceptions;
    - Both handled and unhandled SQL exceptions/errors

    This implies that if a programmatically-handled non-SQL exception occurs in a transaction, such a transaction, by default, will not be counted as an *Error transaction* by this test.

    Sometimes however, administrators may want to be alerted even if some non-SQL exceptions that have already been handled programmatically, occur. This can be achieved by configuring a comma-separated list of these exceptions in the **INCLUDED EXCEPTIONS** text box. Here, each exception you want to include has to be defined using its fully qualified exception class name. For instance, your **INCLUDED EXCEPTIONS** specification can be as follows: java.lang.NullPointerException, java.lang.IndexOutOfBoundsException. **Note that wild card characters cannot be used as part of**

your specification. Once the exceptions to be included are configured, then this test will count all transactions in which such exceptions are captured as *Error transactions*.

20. **IGNORED EXCEPTIONS** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, this test flags the transactions in which the following errors/exceptions are captured, as *Error transactions*:

    - All unhandled exceptions;
    - Both handled and unhandled SQL exceptions/errors

    Sometimes however, administrators may want eG to disregard certain unhandled exceptions (or handled SQL exceptions), as they may not pose any threat to the stability of the transaction or to the web site/web application. To achieve this, administrators can configure a comma-separated list of such inconsequential exceptions in the **IGNORED EXCEPTIONS** text box. Here, you need to configure each exception you want to exclude using its fully qualified exception class name. For instance, your **EXCLUDED EXCEPTIONS** specification can be as follows: java.sql.SQLException,java.io.FileNotFoundException. **Note that wild card characters cannot be used as part of your specification**. Once the exceptions to be excluded are configured, then this test will exclude all transactions in which such exceptions are captured from its count of Error transactions.

21. **IGNORED CHARACTERS** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, eG excludes all transaction URLs that contain the '\' character from monitoring. If you want eG to ignore transaction URLs with any other special characters, then specify these characters as a comma-separated list in the **IGNORED CHARACTERS** text box. For instance, your specification can be: \\,&,~

22. **MAX GROUPED URLS PER MEASURE PERIOD** - **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. This test groups URLs according to the **MAX URL SEGMENTS** specification. These grouped URLs will be the descriptors of the test. For each grouped URL, response time metrics will be aggregated across all transaction URLs in that group and reported.

    When monitoring web sites/web applications to which the transaction volume is normally high, this test may report metrics for hundreds of descriptors. If all these descriptors are listed in the **Layers** tab page of the eG monitoring console, it will certainly clutter the display. To avoid this, by default, the test displays metrics for a maximum of 50 descriptors – i.e., 50 grouped URLs alone – in the eG monitoring console, during every measure period. This is why, the **MAX GROUPED URLS PER MEASURE PERIOD** parameter is set to 50 by default.

    To determine which 50 grouped URLs should be displayed in the eG monitoring console, the eG BTM follows the below-mentioned logic:

    - Top priority is reserved for URL groups with error transactions. This means that eG BTM first scans URL groups for error transactions. If error transactions are found in 50 URL groups, then eG BTM computes the aggregated response time of each of the 50 groups, sorts the error groups in the descending order of their response time, and displays all these 50 groups alone as the descriptors of this test, in the sorted order.
    - On the other hand, if error transactions are found in only one / a few URL groups – say, only 20 URL groups – then, eG BTM will first arrange these 20 grouped URLs in the descending order of their

response time. It will then compute the aggregated response time of the transactions in each of the other groups (i.e., the error-free groups) that were auto-discovered during the same measure period. These other groups are then arranged in the descending order of the aggregated response time of their transactions. Once this is done, eG BTM will then pick the top-30 grouped URLs from this sorted list.

In this case, when displaying the descriptors of this test in the **Layers** tab page, the 20 error groups are first displayed (in the descending order of their response time), followed by the 30 'error-free' groups (also in the descending order of their response time).

At any given point in time, you can increase/decrease the maximum number of descriptors this test should supportby modifying the value of the **MAX GROUPED URLS PER MEASURE PERIOD** parameter.

23. **MAX SQL QUERIES PER TRANSACTION** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. Typically, from the detailed diagnosis of a slow/stalled/error transaction on a JVM node, you can drill down to view the SQL queries (if any) executed by that transaction from that node and the execution time of each query. By default, eG picks the first **500** SQL queries executed by the transaction, compares the execution time of each query with the **SQL EXECUTION CUTOFF** configured for this test, and displays only those queries with an execution time that is higher than the configured cutoff. This is why, the **MAX SQL QUERIES PER TRANSACTION** parameter is set to 500 by default.

    To improve agent performance, you may want the **SQL EXECUTION CUTOFF** to be compared with the execution time of a less number of queries – say, 200 queries. Similary, to increase the probability of capturing more number of long-running queries, you may want the sql execution cutoff to be compared with the execution time of a large number of queries – say, 1000 queries. For this, you just need to modify the **MAX SQL QUERIES PER TRANSACTION** specification to suit your purpose.

24. **TIMEOUT**– By default, the eG agent will wait for 1000 milliseconds for a response from the eG Application Server agent. If no response is received, then the test will timeout. You can change this timeout value, if required.

25. **DD FREQUENCY** – Refers to the frequency with which detailed diagnosis measures are to be generated for this test. The default is 1:1. This indicates that, by default, detailed measures will be generated every time this test runs, and also every time the test detects a problem. You can modify this frequency, if you so desire. Also, if you intend to disable the detailed diagnosis capability for this test, you can do so by specifying none against **DD FREQUENCY**.

26. **DETAILED DIAGNOSIS** – To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

    The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

    - The eG manager license should allow the detailed diagnosis capability

- Both the normal and abnormalfrequencies configured for the detailed diagnosis measures should not be 0.

Measures reported by the test:

Measurement	Description	Measurement Unit	Interpretation
All transactions	Indicates the total number of requests received for transactions of this pattern during the last measurement period.	Number	By comparing the value of this measure across transaction patterns, you can identify the most popular transaction patterns. Using the detailed diagnosis of this measure, you can then figure out which specific transactions of that pattern are most requested. For the **Summary** descriptor, this measure will reveal the total number of transaction requests received by the target JVM during the last measurement period. This is a good indicator of the transaction workload on that JVM.
Avg response time	Indicates the average time taken by the transactions of this pattern to complete execution.	Secs	Compare the value of this measure across patterns to isolate the type of transactions that were taking too long to execute. You can then use the detailed diagnosis of the All transactions measure of that group to know how much time each transaction in that group took to execute. This will lead you to the slowest transaction. For the **Summary** descriptor, this measure will reveal the average responsiveness of all the transaction requests received by the target JVM during the last measurement period. An abnormally low value for this measure for the Summary descriptor could indicate a serious processing bottleneck on the target JVM.

Measurement	Description	Measurement Unit	Interpretation
Healthy transactions	Indicates the number of healthy transactions of this pattern.	Number	By default, this measure will report the count of transactions with a response time less than 4000 milliseconds. You can change this default setting by modifying the thresholds of the *Avg response time* measure using the eG admin interface. For the **Summary** descriptor, this measure will report the total number of healthy transactions on the target JVM.
Healthy transactions percentage	Indicates what percentage of the total number of transactions of this pattern is healthy.	Percent	To know which are the healthy transactions, use the detailed diagnosis of this measure. For the **Summary** descriptor, this measure will report the overall percentage of healthy transactions on the target JVM.
Slow transactions	Indicates the number of transactions of this pattern that were slow during the last measurement period.	Number	By default, this measure will report the number of transactions with a response time higher than 4000 milliseconds and lesser than 60000 milliseconds. You can change this default setting by modifying the thresholds of the *Avg response time* measure using the eG admin interface. A high value for this measure is a cause for concern, as too many slow transactions means that user experience with the web application is poor. For the **Summary** descriptor, this measure will report the total number of slow transactons on the target JVM. This is a good indicator of the processing power of the target JVM.
Slow transaction response time	Indicates the average time taken by the slow transactions of this pattern to execute.	Secs	For the **Summary** descriptor, this measure will report the average response time of all the slow transactions on the target JVM.
Slow transactions	Indicates what	Percent	Use the detailed diagnosis of this measure

Measurement	Description	Measurement Unit	Interpretation
percentage	percentage of the total number of transactions of this pattern is currently slow.		to know which precise transactions of a pattern are slow. You can drill down from a slow transaction to know what is causing the slowness. For the **Summary** descriptor, this measure will report the overall percentage of slow transactions on the monitored JVM.
Error transactions	Indicates the number of transactions of this pattern that experienced errors during the last measurement period.	Number	A high value is a cause for concern, as too many error transactions to a web application can significantly damage the user experience with that application. For the **Summary** descriptor, this measure will report the total number of error transactons on the target JVM. This is a good indicator of how error-prone the target JVM is.
Error transactions response time	Indicates the average duration for which the transactions of this pattern were processed before an error condition was detected.	Secs	The value of this measure will help you discern if error transactions were also slow. For the **Summary** descriptor, this measure will report the average response time of all error transactions on the target JVM.
Error transactions percentage	Indicates what percentage of the total number of transactions of this pattern is experiencing errors.	Percent	Use the detailed diagnosis of this measure to isolate the error transactions. You can even drill down from an error transaction in the detailed diagnosis to determine the cause of the error. For the **Summary** descriptor, this measure will report the overall percentage of transactions of this pattern on the target JVM that is currently experiencing errors.
Stalled transactions	Indicates the number of transactions of this pattern that were stalled during the last measurement period.	Number	By default, this measure will report the number of transactions with a response time higher than 60000 milliseconds. You can change this default setting by modifying the thresholds of the *Avg*

Measurement	Description	Measurement Unit	Interpretation
			response time measure using the eG admin interface. A high value is a cause for concern, as too many stalled transactions means that user experience with the web application is poor. For the **Summary** descriptor, this measure will report the total number of stalled transactons on the target JVM.
Stalled transactions response time:	Indicates the average time taken by the stalled transactions of this pattern to execute.	Secs	For the **Summary** descriptor, this measure will report the average response time of all stalled transactions on the target JVM.
Stalled transactions percentage	Indicates what percentage of the total number of transactions of this pattern is stalling.	Percent	Use the detailed diagnosis of this measure to know which precise transactions of a pattern are stalled. You can drill down from a stalled transaction to know what is causing that transaction to stall. For the **Summary** descriptor, this measure will report the overall percentage of transactions of this pattern on the target JVM that is stalling.
Slow SQL statements executed	Indicates the number of slow SQL queries that were executed by the transactions of this pattern during the last measurement period.	Number	For the **Summary** descriptor, this measure will report the total number of slow SQL queries executed by all transactions to the target JVM.
Slow SQL statement time	Indicates the average execution time of the slow SQL queries that were run by the transactions of this pattern.	Secs	If there are too many slow transactions of a pattern, you may want to check the value of this measure for that pattern to figure out if query execution is slowing down the transactions. Use the detailed diagnosis of the *Slow transactions* measure to identify the precise slow transaction. Then, drill down from that

Measurement	Description	Measurement Unit	Interpretation
			slow transaction to confirm whether/not database queries have contributed to the slowness. Deep-diving into the queries will reveal the slowest queries and their impact on the execution time of the transaction.

1.5 Key Java Business Transactions Test

For any business-critical application, some transactions will always be considered key from the point of view user experience and business impact. For instance, in the case of a retail banking web application, fund transfers executed online are critical transactions that have to be tracked closely for delays / errors, as problems in the transaction will cost both consumers and the company dearly. Using the Java Key Business Transactions test, administrators can perform focused monitoring of such critical transactions alone.

For each transaction URL pattern configured for monitoring on a JVM node, this test reports the count of requests for that transaction pattern, and the count and percentage of transactions of that pattern that were slow / stalling / error-prone. Detailed diagnostics provided by the test highlight the slow / stalled / error transactions of a pattern, and pinpoint the precise reason why that key transaction slowed down / stalled / encountered errors - is it because of an inefficient database query? is it because of a processing bottleneck on the JVM node? or is it owing to slow remote service calls? This way, the test enables you to quickly detect inconsistencies in the performance of your critical business transactions and accurately isolate its root-cause, so that you can fix the issues well before users notice them.

.

Target of the Test: A BTM-enabled JVM

Agent deploying the test : An internal/remote agent

Output of the test: One set of results for each URL pattern configured for monitoring

Test parameters:

1. **TEST PERIOD** - How often should the test be executed
2. **HOST** - The host for which the test is to be configured

3. **BTM PORT** -Specify the port number specified as **BTM_Port** in the **btmOther.props** file on the JVM node being monitored. If the JVM is being monitored in an agent-based manner, then the btmOther.props file will be in the **<EG_AGENT_INSTALL_DIR>\lib\bm** directory.

4. **URL PATTERNS** - Provide a comma-separated list of *PatternName:URLPattern* pairs to be monitored. The *PatternName* can be any name that uniquely identifies the pattern. These *PattenNames* will be the descriptors of this test. For the URLPattern, you can either provide the exact URL to be monitored , or can provide a pattern. For instance, if you want to monitor requests to distinct and specific web pages - say, login.jsp and payment.jsp of a web application - then you can specify the exact URL of these web pages as your **URL PATTERNS** . In this case your specification will be*,Login:/web/login.jsp,Payment:/web/payment.jsp*. On the other hand, if you want to monitor requests to all payment-related web pages in a web application - say, *payment.jsp, creditcardpayment.jsp, debitcardpayment.jsp, onlinepayment.jsp*, and more - and you want the metrics to be grouped under a single head called *Payment*, then you can specify a pattern instead of the exact URL. In this case, your **URL PATTERNS** specification will be *Payment:*payment**. The leading '*' in the specification signifies any number of leading characters, while the trailing '*' signifies any number of trailing characters. This means that the specification in our example will track requests to all pages with names that contain the word *payment*. Your *URLPattern* can also be **expr* or *expr** or **expr1*expr2** or *expr1*expr2*, etc.

5. **KEY EXCLUDED PATTERNS** - By default, this test does not track requests to the following URL patterns:

   .ttf, *.otf, *.woff, *.woff2, *.eot, *.cff, *.afm, *.lwfn, *.ffil, *.fon, *.pfm, *.pfb, *.std, *.pro, *.xsf, *.jpg, *.jpeg, *.jpe, *.jif, *.jfif, *.jfi, *.jp2, *.j2k, *.jpf, *.jpx, *.jpm, *.jxr, *.hdp, *.wdp, *.mj2, *.webp, *.gif, *.png, *.apng, *.mng, *.tiff, *.tif, *.xbm, *.bmp, *.dib, *.svg, *.svgz, *.mpg, *.mpeg, *.mpeg2, *.avi, *.wmv, *.mov, *.rm, *.ram, *.swf, *.flv, *.ogg, *.webm, *.mp4, *.ts, *.mid, *.midi, *.rm, *.ram, *.wma, *.aac, *.wav, *.ogg, *.mp3, *.mp4, *.css, *.js, *.ico|/egurkha

   If required, you can remove one/more patterns from this default list, so that such patterns are monitored, or can append more patterns to this list in order to exclude them from monitoring.

6. **METHOD EXEC CUTOFF (MS)**– From the detailed diagnosis of slow/stalled/error transactions, you can drill down and perform deep execution analysis of a particular transaction. In this drill-down, the methods invoked by that slow/stalled/error transaction are listed in the order in which the transaction calls the methods. By configuring a **METHOD EXECUTION CUTOFF**, you can make sure that methods that have been executing for a duration greater the specified cutoff are alone listed when performing execution analysis. For instance, if you specify *5* here, then the **Execution Analysis** window for a slow/stalled/error transaction will list only those methods that have been executing for over 5 milliseconds. This way, you get to focus on only those methods that could have caused the slowness, without being distracted by inconsequential methods. By default, the value of this parameter is set to 250 ms.

7. **SQL EXECUTION CUTOFF (MS)** – Typically, from the detailed diagnosis of a slow/stalled/error transaction on a JVM node, you can drill down to view the SQL queries (if any) executed by that transaction from that node and the execution time of each query. By configuring a **SQL EXECUTION CUTOFF**, you can make sure that queries that have been executing for a duration greater the specified cutoff are alone listed when performing query analysis. For instance, if you specify *5* here, then for a

slow/stalled/error transaction, the **SQL Queries** window will display only those queries that have been executing for over 5 milliseconds. This way, you get to focus on only those queries that could have contributed to the slowness. By default, the value of this parameter is set to 10 ms.

8. **HEALTHY URL TRACE** – By default, this flag is set to **No**. This means that eG will not collect detailed diagnostics for those transactions that are healthy. If you want to enable the detailed diagnosis capability for healthy transactions as well, then set this flag to **Yes**.
9. **MAX HEALTHY URLS PER TEST PERIOD** – **This parameter is applicable only if the HEALTHY URL TRACE flag is set to 'Yes'.** Here, specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Healthy transactions* measure, every time the test runs. By default, this is set to 5*0*, indicating that the detailed diagnosis of the *Healthy transactions* measure will by default list the top-50 transactions, arranged in the descending order of their response times.
10. **MAX SLOW URLS PER TEST PERIOD** - Specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Slow transactions* measure, every time the test runs. By default, this is set to *10*, indicating that the detailed diagnosis of the *Slow transactions* measure will by default list the top-10 transactions, arranged in the descending order of their response times.
11. **MAX STALLED URLS PER TEST PERIOD** - Specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Stalled transactions* measure, every time the test runs. By default, this is set to *10*, indicating that the detailed diagnosis of the *Stalled transactions* measure will by default list the top-10 transactions, arranged in the descending order of their response times.
12. **MAX ERROR URLS PER TEST PERIOD** - Specify the number of top-n transactions that should be listed in the detailed diagnosis of the *Error transactions* measure, every time the test runs. By default, this is set to *10*, indicating that the detailed diagnosis of the *Error transactions* measure will by default list the top-10 transactions, in terms of the number of errors they encountered.
13. **ADVANCED SETTINGS** – To optimize transaction performance and conserve space in the eG database, many restraints have been applied by default on the agent's ability to collect and report detailed diagnostics. Depending upon how well-tuned your eG database is and the level of visibility you require into transaction performance, you may choose to either retain these default settings or override them. If you choose not to disturb the defaults, then set the **ADVANCED SETTINGS** flag to **No**. If you want to modify the defaults, then set the **ADVANCED SETTINGS** flag to **Yes**.
14. **POJO METHOD TRACING LIMIT** and **POJO METHOD TRACING CUTOFF TIME** - **These parameters will appear only if the ADVANCED SETTINGS flag is set to 'true'**. Typically, if the **MONITORING MODE** of this test is set to **Profiler** , then, as part of the detailed diagnostics of a transaction, eG reports the execution time of every POJO, non-POJO, and recursive (i.e. methods that call themselves) method call that a JVM node makes when processing that transaction. Of these, POJO method calls are the most expensive, as they are usually large in number. To ensure that attempts made to collect detailed measures related to POJO method calls do not impact the overall responsiveness of the monitored transaction, eG, by default, collects and reports the execution time of only the following POJO method calls:
    - The first 1000 POJO method calls made by the target JVM node for that transaction; (OR)
    - The POJO method calls that were made by the target JVM node within 10 seconds from the start of the monitored transaction on that node;

Accordingly, the **POJO METHOD TRACING LIMIT** is set to 1000 by default, and the **POJO METHOD TRACING CUTOFF TIME** is set to 10 (seconds) by default. Of these two limits, whichever limit is reached first will automatically be applied by eG for determining when to stop POJO tracing. In other words, once a JVM node starts processing a transaction, the agent begins tracking the POJO method calls made by that node for that transaction. In the process, if the agent finds that the configured tracing limit is reached before the tracing cutoff time is reached, then the agent will stop tracking the POJO method calls, as soon as the tracing limit is reached. On the other hand, if the tracing limit is not reached, then the agent will continue tracking the POJO method calls until the tracing cutoff time is reached. At the end of the cutoff time, the agent will stop tracking the POJO method calls. For instance, if the JVM node makes 1000 POJO method calls within say, 6 seconds from when it began processing the transaction, then the eG agent will not wait for the cutoff time of 10 seconds to be reached; instead, it will stop tracing at the end of the thousandth POJO method call, and report the execution time of each of the 1000 calls alone. On the other hand, if the JVM node does not make over 1000 POJO method calls till the 10 second cutoff expires, then the eG agent continues tracking the POJO method calls till the end of 10 seconds, and reports the details of all those that were calls made till the cutoff time.

Depending upon how many POJO calls you want to trace and how much overhead you want to impose on the agent and on the transaction, you can increase / decrease the **POJO METHOD TRACING LIMIT** and **POJO METHOD TRACING CUTOFF TIME** specifications.

15. **NON-POJO METHOD TRACING LIMIT** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, when reporting the detailed diagnosis of a transaction on a particular JVM node, this test reports the execution time of only the first 1000 non-POJO method calls (which includes JMS, JCO, HTTP, Java, SQL, etc.) that the target JVM node makes for that transaction. This is why, the non-pojo method tracing limit parameter is set to 1000 by default. If you want, you can change the tracing limit to enable the test to report the details of more or fewer non-POJO method calls made by a JVM node. While a high value for this parameter may take you closer to identifying the non-POJO method that could have caused the transaction to slowdown on a particular JVM node, it may also marginally increase the overheads of the transaction and the eG agent.
16. **RECURSIVE METHOD TRACING LIMIT** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. A recursive method is a method that calls itself. By default, when reporting the detailed diagnosis of a transaction on a particular JVM node, this test reports the execution time of only the first 1000 recursive method calls (which includes JMS, JCO, HTTP, Java, SQL, etc.) that the target JVM node makes for that transaction. This is why, the **RECURSIVE METHOD TRACING LIMIT** parameter is set to 1000 by default. If you want, you can change the tracing limit to enable the test to report the details of more or fewer recursive method calls made by a JVM node. While a high value for this parameter may take you closer to identifying the recursive method that could have caused the transaction to slowdown on a particular JVM node, it may also marginally increase the overheads of the transaction and the eG agent.
17. **EXCEPTION STACKTRACE LINES** –**This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. As part of detailed diagnostics, this test, by default, lists the first 10 stacktrace lines of each JavaScript error/exception that it captures on the target JVM node for a specific transaction, so as to enable easy and efficient troubleshooting. This is why, the **EXCEPTION STACKTRACE LINES** parameter is set to 10 by default. If required, you can have this test display more

or fewer stacktrace lines by overriding this default setting.

18. **INCLUDED EXCEPTIONS** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, this test flags the transactions in which the following errors/exceptions are captured, as *Error transactions*:

    - All unhandled exceptions;
    - Both handled and unhandled SQL exceptions/errors

    This implies that if a programmatically-handled non-SQL exception occurs in a transaction, such a transaction, by default, will not be counted as an *Error transaction* by this test.

    Sometimes however, administrators may want to be alerted even if some non-SQL exceptions that have already been handled programmatically, occur. This can be achieved by configuring a comma-separated list of these exceptions in the **INCLUDED EXCEPTIONS** text box. Here, each exception you want to include has to be defined using its fully qualified exception class name. For instance, your **INCLUDED EXCEPTIONS** specification can be as follows: java.lang.NullPointerException, java.lang.IndexOutOfBoundsException. **Note that wild card characters cannot be used as part of your specification**. Once the exceptions to be included are configured, then this test will count all transactions in which such exceptions are captured as *Error transactions*.

19. **IGNORED EXCEPTIONS** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, this test flags the transactions in which the following errors/exceptions are captured, as *Error transactions*:

    - All unhandled exceptions;
    - Both handled and unhandled SQL exceptions/errors

    Sometimes however, administrators may want eG to disregard certain unhandled exceptions (or handled SQL exceptions), as they may not pose any threat to the stability of the transaction or to the web site/web application. To achieve this, administrators can configure a comma-separated list of such inconsequential exceptions in the **IGNORED EXCEPTIONS** text box. Here, you need to configure each exception you want to exclude using its fully qualified exception class name. For instance, your **EXCLUDED EXCEPTIONS** specification can be as follows: java.sql.SQLException,java.io.FileNotFoundException. **Note that wild card characters cannot be used as part of your specification**. Once the exceptions to be excluded are configured, then this test will exclude all transactions in which such exceptions are captured from its count of Error transactions.

20. **IGNORED CHARACTERS** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. By default, eG excludes all transaction URLs that contain the '\' character from monitoring. If you want eG to ignore transaction URLs with any other special characters, then specify these characters as a comma-separated list in the **IGNORED CHARACTERS** text box. For instance, your specification can be: \\,&,~

21. **MAX GROUPED URLS PER MEASURE PERIOD** - **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. This test groups URLs according to the **MAX URL SEGMENTS** specification. These grouped URLs will be the descriptors of the test. For each grouped URL, response time metrics will be aggregated across all transaction URLs in that group and reported.

When monitoring web sites/web applications to which the transaction volume is normally high, this test may report metrics for hundreds of descriptors. If all these descriptors are listed in the **Layers** tab page of the eG monitoring console, it will certainly clutter the display. To avoid this, by default, the test displays metrics for a maximum of 50 descriptors – i.e., 50 grouped URLs alone – in the eG monitoring console, during every measure period. This is why, the **MAX GROUPED URLS PER MEASURE PERIOD** parameter is set to 50 by default.

To determine which 50 grouped URLs should be displayed in the eG monitoring console, the eG BTM follows the below-mentioned logic:

- Top priority is reserved for URL groups with error transactions. This means that eG BTM first scans URL groups for error transactions. If error transactions are found in 50 URL groups, then eG BTM computes the aggregated response time of each of the 50 groups, sorts the error groups in the descending order of their response time, and displays all these 50 groups alone as the descriptors of this test, in the sorted order.
- On the other hand, if error transactions are found in only one / a few URL groups – say, only 20 URL groups – then, eG BTM will first arrange these 20 grouped URLs in the descending order of their response time. It will then compute the aggregated response time of the transactions in each of the other groups (i.e., the error-free groups) that were auto-discovered during the same measure period. These other groups are then arranged in the descending order of the aggregated response time of their transactions. Once this is done, eG BTM will then pick the top-30 grouped URLs from this sorted list.

  In this case, when displaying the descriptors of this test in the **Layers** tab page, the 20 error groups are first displayed (in the descending order of their response time), followed by the 30 'error-free' groups (also in the descending order of their response time).

  At any given point in time, you can increase/decrease the maximum number of descriptors this test should supportby modifying the value of the **MAX GROUPED URLS PER MEASURE PERIOD** parameter.

22. **MAX SQL QUERIES PER TRANSACTION** – **This parameter will appear only if the ADVANCED SETTINGS flag is set to 'true'**. Typically, from the detailed diagnosis of a slow/stalled/error transaction on a JVM node, you can drill down to view the SQL queries (if any) executed by that transaction from that node and the execution time of each query. By default, eG picks the first **500** SQL queries executed by the transaction, compares the execution time of each query with the **SQL EXECUTION CUTOFF** configured for this test, and displays only those queries with an execution time that is higher than the configured cutoff. This is why, the **MAX SQL QUERIES PER TRANSACTION** parameter is set to 500 by default.

    To improve agent performance, you may want the **SQL EXECUTION CUTOFF** to be compared with the execution time of a less number of queries – say, 200 queries. Similary, to increase the probability of capturing more number of long-running queries, you may want the sql execution cutoff to be compared with the execution time of a large number of queries – say, 1000 queries. For this, you just need to modify the **MAX SQL QUERIES PER TRANSACTION** specification to suit your purpose.

23. **TIMEOUT**– By default, the eG agent will wait for 1000 milliseconds for a response from the eG

Application Server agent. If no response is received, then the test will timeout. You can change this timeout value, if required.

24. **DD FREQUENCY** – Refers to the frequency with which detailed diagnosis measures are to be generated for this test. The default is 1:1. This indicates that, by default, detailed measures will be generated every time this test runs, and also every time the test detects a problem. You can modify this frequency, if you so desire. Also, if you intend to disable the detailed diagnosis capability for this test, you can do so by specifying none against **DD FREQUENCY**.
25. **DETAILED DIAGNOSIS** – To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

    The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

    - The eG manager license should allow the detailed diagnosis capability
    - Both the normal and abnormalfrequencies configured for the detailed diagnosis measures should not be 0.

Measures reported by the test:

Measurement	Description	Measurement Unit	Interpretation
All transactions	Indicates the total number of requests received for transactions of this pattern during the last measurement period.	Number	By comparing the value of this measure across transaction patterns, you can identify the most popular transaction patterns. Using the detailed diagnosis of this measure, you can then figure out which specific transactions of that pattern are most requested.
Avg response time	Indicates the average time taken by the transactions of this pattern to complete execution.	Secs	Compare the value of this measure across patterns to isolate the type of transactions that were taking too long to execute. You can then use the detailed diagnosis of the All transactions measure of that group to know how much time each transaction in that group took to execute. This will lead you to the slowest transaction.

Measurement	Description	Measurement Unit	Interpretation
Healthy transactions	Indicates the number of healthy transactions of this pattern.	Number	
Healthy transactions percentage	Indicates what percentage of the total number of transactions of this pattern is healthy.	Percent	To know which are the healthy transactions, use the detailed diagnosis of this measure.
Slow transactions	Indicates the number of transactions of this pattern that were slow during the last measurement period.	Number	This measure will report the number of transactions with a response time higher than the configured **SLOW TRANSACTION CUTOFF (MS)**. A high value is a cause for concern, as too many slow transactions means that user experience with the web application is poor.
Slow transaction response time	Indicates the average time taken by the slow transactions of this pattern to execute.	Secs	
Slow transactions percentage	Indicates what percentage of the total number of transactions of this pattern is currently slow.	Percent	Use the detailed diagnosis of this measure to know which precise transactions of a pattern are slow. You can drill down from a slow transaction to know what is causing the slowness.
Error transactions	Indicates the number of transactions of this pattern that experienced errors during the last measurement period.	Number	A high value is a cause for concern, as too many error transactions to a web application can significantly damage the user experience with that application.
Error transactions response time	Indicates the average duration for which the transactions of this pattern were processed before an error condition was detected.	Secs	The value of this measure will help you discern if error transactions were also slow.

Measurement	Description	Measurement Unit	Interpretation
Error transactions percentage	Indicates what percentage of the total number of transactions of this pattern is experiencing errors.	Percent	Use the detailed diagnosis of this measure to isolate the error transactions. You can even drill down from an error transaction in the detailed diagnosis to determine the cause of the error.
Stalled transactions	Indicates the number of transactions of this pattern that were stalled during the last measurement period.	Number	This measure will report the number of transactions with a response time higher than the configured **STALLED TRANSACTION CUTOFF (MS**). A high value is a cause for concern, as too many stalled transactions means that user experience with the web application is poor.
Stalled transactions response time:	Indicates the average time taken by the stalled transactions of this pattern to execute.	Secs	
Stalled transactions percentage	Indicates what percentage of the total number of transactions of this pattern is stalling.	Percent	Use the detailed diagnosis of this measure to know which precise transactions of a pattern are stalled. You can drill down from a stalled transaction to know what is causing that transaction to stall.
Slow SQL statements executed	Indicates the number of slow SQL queries that were executed by the transactions of this pattern during the last measurement period.	Number	
Slow SQL statement time	Indicates the average execution time of the slow SQL queries that were run by the transactions of this pattern.	Secs	If there are too many slow transactions of a pattern, you may want to check the value of this measure for that pattern to figure out if query execution is slowing down the transactions. Use the detailed diagnosis of the *Slow transactions*

Measurement	Description	Measurement Unit	Interpretation
			measure to identify the precise slow transaction. Then, drill down from that slow transaction to confirm whether/not database queries have contributed to the slowness. Deep-diving into the queries will reveal the slowest queries and their impact on the execution time of the transaction.

1.6 Detailed Diagnostics

By reporting detailed diagnostics on transaction responsiveness and errors, eG Enterprise not only points you to the slow/stalled/error transaction URLs, but also reveals what could be causing the slowness/errors.

Figure 1.29 reveals detailed diagnosis of the Avg response time measure of the OrderProcessing grouped URL of the **Java Business Transactions** test.

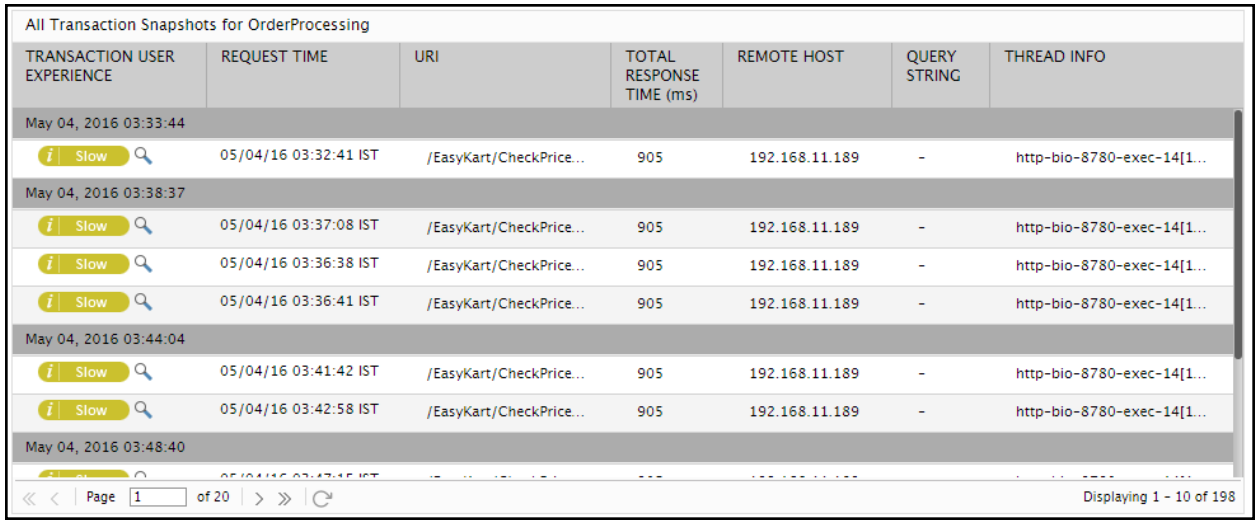


Figure 1.29: Detailed diagnosis of the Avg response time measure of the Java Business Transactions test

The detailed diagnosis reveals the individual transaction URLs in the grouped URL that users requested for, the total response time of each transaction, the client (remote host) from which each transaction request was received, the thread executing the transaction, and the query string of the transaction URL. The per-transaction response time displayed in Figure 1.29 includes the following:

- The total time for which the transaction request was processed by the target JVM and by other BTM-enabled JVMs in the transaction path thereafter, until the time the response for that transaction request was sent out by the target JVM;

- The time taken by external calls (SQL query / HTTP / JMX / Java / JMS / SAP JCO / async) to other JVMs or backends in the transaction path;

Additionally, the overall experience of the users with each transaction – whether it is slow, stalled, or error - is also revealed in the **TRANSACTION USER EXPERIENCE** column. The per-transaction statistics are also sorted in the descending order of the transaction response time, starting with the slowest transaction and ending with the healthiest one. In the event that the Avg response time of a grouped URL registers an abnormally high value, you can use these detailed metrics to quickly and accurately identify the exact transaction in the group that is significantly contributing to the poor user experience with the group.

Similar diagnostics are also available for the Slow transaction percentage, Stalled transaction percentage, and Error transaction percentage measures of the **Java Business Transactions** test. With the help of these detailed measures, you will be able to quickly and accurately identify the slow, stalled, and error transactions in a grouped URL.

Once a slow/stalled transaction is revealed, the next question is what is causing the transaction to slowdown. Transaction responsiveness can be impacted by any of the following factors:

- An inefficient database query run by the target JVM node;
- In a multi-JVM environment, a time-consuming POJO / non-POJO method called by any JVM node;
- A poorly responsiveness remote service call made by the target JVM node;

With the help of illustrated examples, the links below describe how drill-downs from the detailed diagnostics enable accurate isolation of the root-cause of a transaction slowdown / errors in a transaction.

Detailed Diagnostics Revealing that an Inefficient Database Query is the Reason for a Slow Transaction

Detailed Diagnostics Revealing that a Slow JVM Node is Causing Transactions to Slowdown

Detailed Diagnostics Revealing the Root-cause of an Error Transaction

Detailed Diagnostics Revealing that a Remote Service Call is the Reason Why a Transaction Slowed Down

1.6.1 Detailed Diagnostics Revealing that an Inefficient Database Query is the Reason for a Slow Transaction

Let us consider the example of the EasyKart web application, which enables users to quickly shop for products. Say that this web application has been deployed on the Oracle WebLogic server, EasyKart:80. Users of EasyKart complained that every time they tried to browse the catalog of products on the EasyKart web site, the response was very poor. Using eG's Java Business Transactions test of the EasyKart:80 Oracle WebLogic server, you can promptly capture this anomaly! As you can see in Figure 1 below, the Java Business Transactions test has accurately captured and reported that the Slow transactions percentage for the /EasyKart/BrowseProducts.jsp transaction is 100%. This means that 100% of the requests for the EasyKart/BrowseProducts.jsp transaction were serviced slowly (see Figure 1)!

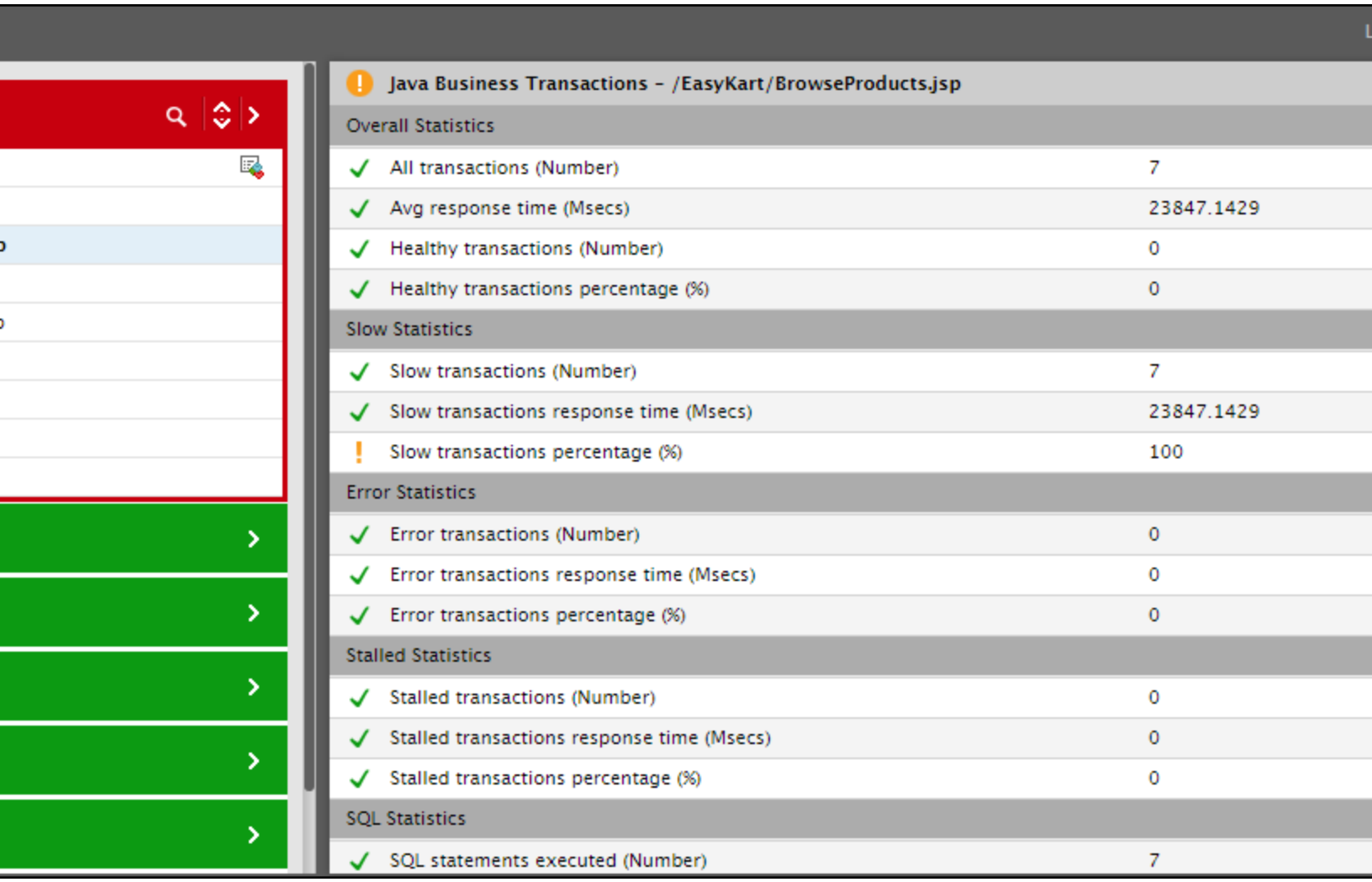
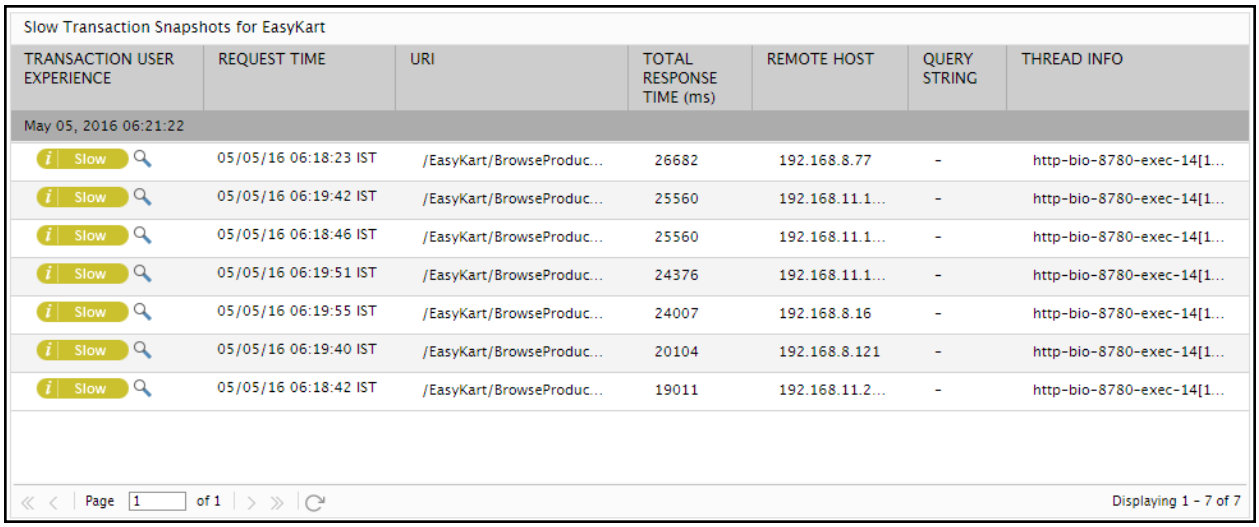


Figure 1.30: The Layers tab page indicating that all requests for /EasyKart/BrowseProducts.jsp were slow

To know which request received the slowest response, click the **DIAGNOSIS** icon against the Slow transactions percentage measure in 1.6.1. Figure 1.31 will then appear listing all the transaction requests that were slow, the time at which each request was sent, the total response time of every request, the client from which the, request was received, the query string of the transaction URL, and the thread executing the request.


Slow Transaction Snapshots for EasyKart

TRANSACTION USER EXPERIENCE	REQUEST TIME	URI	TOTAL RESPONSE TIME (ms)	REMOTE HOST	QUERY STRING	THREAD INFO
May 05, 2016 06:21:22						
Slow	05/05/16 06:18:23 IST	/EasyKart/BrowseProduc...	26682	192.168.8.77	-	http-bio-8780-exec-14[1...
Slow	05/05/16 06:19:42 IST	/EasyKart/BrowseProduc...	25560	192.168.11.1...	-	http-bio-8780-exec-14[1...
Slow	05/05/16 06:18:46 IST	/EasyKart/BrowseProduc...	25560	192.168.11.1...	-	http-bio-8780-exec-14[1...
Slow	05/05/16 06:19:51 IST	/EasyKart/BrowseProduc...	24376	192.168.11.1...	-	http-bio-8780-exec-14[1...
Slow	05/05/16 06:19:55 IST	/EasyKart/BrowseProduc...	24007	192.168.8.16	-	http-bio-8780-exec-14[1...
Slow	05/05/16 06:19:40 IST	/EasyKart/BrowseProduc...	20104	192.168.8.121	-	http-bio-8780-exec-14[1...
Slow	05/05/16 06:18:42 IST	/EasyKart/BrowseProduc...	19011	192.168.11.2...	-	http-bio-8780-exec-14[1...

Page 1 of 1 Displaying 1 - 7 of 7

Figure 1.31: Detailed Diagnosis of the Slow transactions percentage measure

Since the requests are arranged in the descending order of their response time, a quick look at the detailed diagnostics will lead you to the precise request that is the slowest. But, why is response to this request slow? To answer this question, click the 'magnifying glass' icon against Slow in the **TRANSACTION USER EXPERIENCE** column of the slowest request (i.e., the topmost request in Figure 1.31).

Figure 1.32 will then appear revealing the cross-application flow of the slow transaction. This flow diagram clearly reveals the following:

- The JVMs and backends through which the transaction travelled;
- The time for which the transaction request was processed at each BTM-enabled JVM; **note that this time will not be computed for JVMs that are in the transaction path, but are not BTM-enabled and those that are BTM-enabled but are not managed by eG;**
- The time consumed by external calls made by the transaction and the number of times each type of call was made;

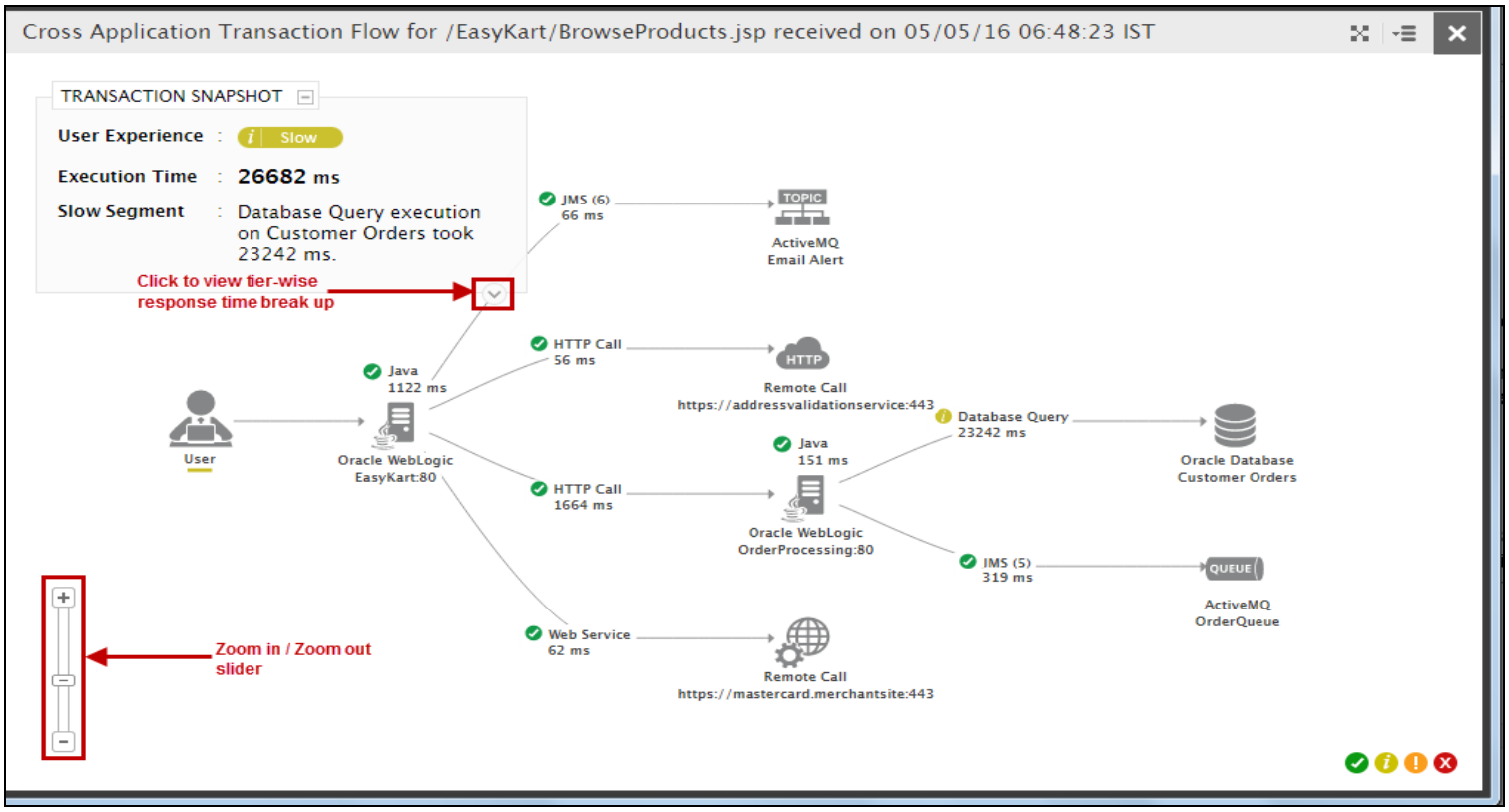


Figure 1.32: Cross-application transaction flow

Using conventional color codes and intuitive icons, the transaction flow chart precisely pinpoints where the transaction slowed down. In the case of Figure 1.32 above, from the color-coding it is clear that the **Database Query** executed by the Oracle WebLogic server – OrderProcessing:80 - is taking a long time for execution. The question now is which query is this. To determine that, click on **Database Query** in Figure 1.32.

Drilling down from **Database Query** in Figure 1.32 automatically opens the list of **SQL Queries** executed by the slow transaction in question (see Figure 1.33). The execution time of each query and what percentage of the total response time of the transaction each query is consuming will be displayed here. From Figure 1.33, it is evident that a **SELECT DISTINCT specials. . .** query is taking over 230000 milliseconds for execution – this is apparently 98.02% of the total response time of the target transaction. This time-consuming query is what is causing the transaction to slow down. To view the complete query, click on that query in the **SQL Queries** list of Figure 1.33. The detailed query will then be displayed in the **Query** section of Figure 1.33.

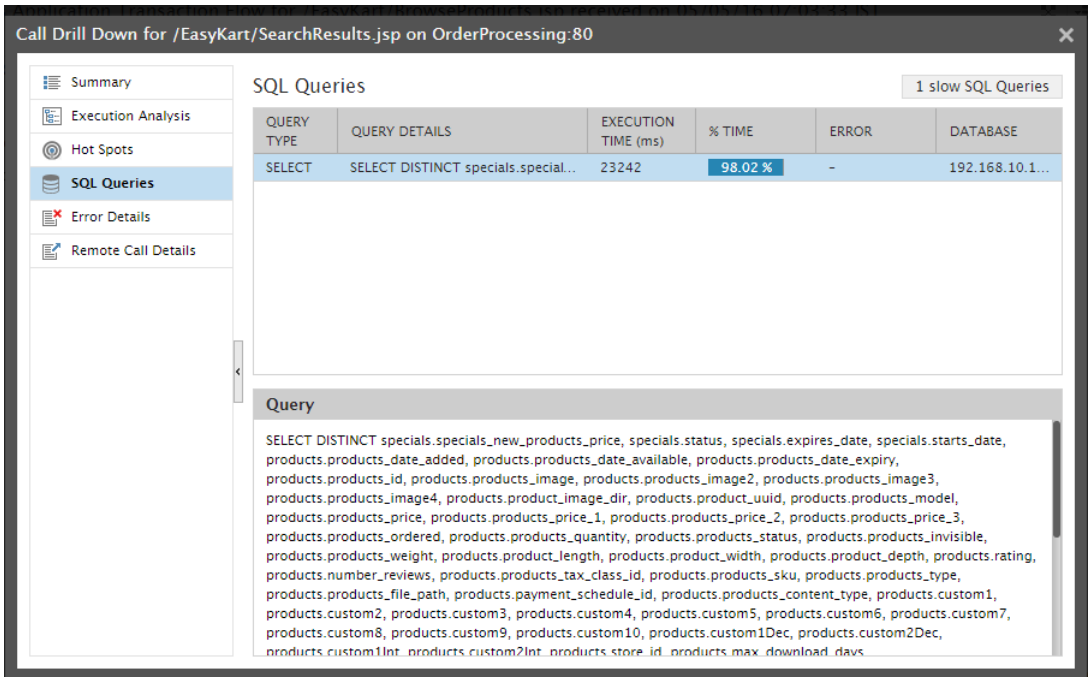


Figure 1.33: Analyzing the slow query

This way, using a short sequence of mouse clicks, you have zeroed-in on the source of the transaction slowness.

The **TRANSACTION SNAPSHOT** section in Figure 1.32 leads you to the same root-cause, **without requiring any clicks!** The details provided by this section are as follows:

- **User Experience:** The user experience with the BrowseProducts transaction; in our example, this is **Slow**
- **Execution Time:** The total response time of the BrowseProducts transaction;
- **Slow Segment:** Where exactly the BrowseProducts transaction slowed down;

From the **Slow Segment** display, it is evident that a database query executed by the BrowseProducts.jsp transaction on the Customer Orders database took over 23000 millisecs for execution, thereby slowing down the entire transaction! This corroborates our findings from the cross-application transaction flow and the subsequent query analysis.

Now, click on the down-arrow button at the bottom tip of the **TRANSACTION SNAPSHOT** section (as indicated by Figure 1.32). Doing so will reveal a tier-wise breakup of the transaction response time (see Figure 1.34). This way, you can quickly compare response time across tiers, and accurately isolate where the bottleneck lies – in this case, it is in the database queries.

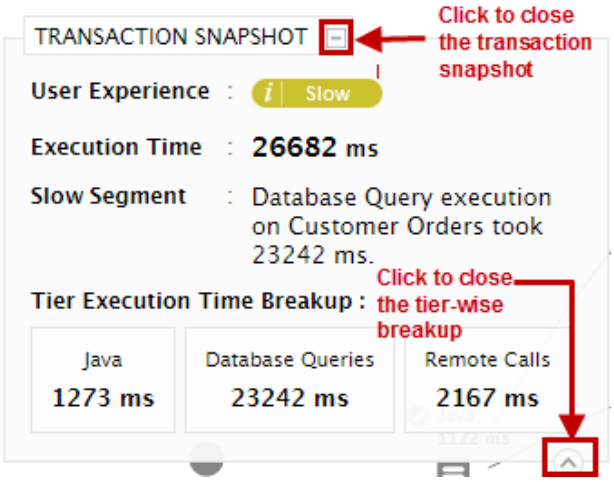


Figure 1.34: Tier-wise response time breakup

To close the tier-wise breakup, click on the up arrow button indicated by Figure 1.34.

You can even close the transaction snapshot pop-up if you want to by clicking on the button alongside the title **TRANSACTION SNAPSHOT** (as indicated by Figure 1.34).

Let us now revisit the cross-application flow diagram of the BrowseProducts transaction. You can use the top-down slider at the bottom, left corner of the flow diagram (as indicated by Figure 1.32) to zoom your diagram in and out.

Moreover, by default, the time spent by the transaction at every point cut is reported in milliseconds in the flow diagram. You can reconfigure the flow diagram to express the time spent as a percentage of total transaction response time instead. For this, first click the button at the right, top corner of the flow diagram. The options depicted by 1.6.1 will then appear.

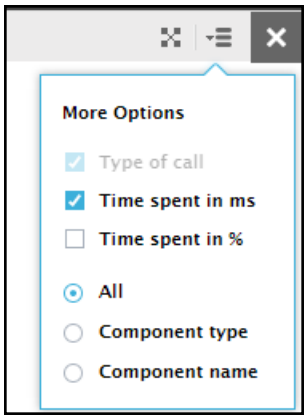


Figure 1.35: Expressing the time spent at every point cut as a percentage of total transaction response time

Uncheck the **Time spent in ms** check box in 1.6.1 and select the **Time spent in %** check box to make sure that the response time at every point cut is displayed as a percentage of total transaction response time. The percentage will enable you to better judge where the transaction spent maximum time.

You can also choose the **Component type** or **Component name** options in 1.6.1 to have the component type only or the component name only (as the case may be) displayed for each of the nodes in the cross-application transaction flow. By default, both component type and name will be displayed for each node.

Let us now explore the **Summary** section of the call drill down. For that, click the **Summary** option in the left panel of Figure 1.33. Figure 1.36 will appear.

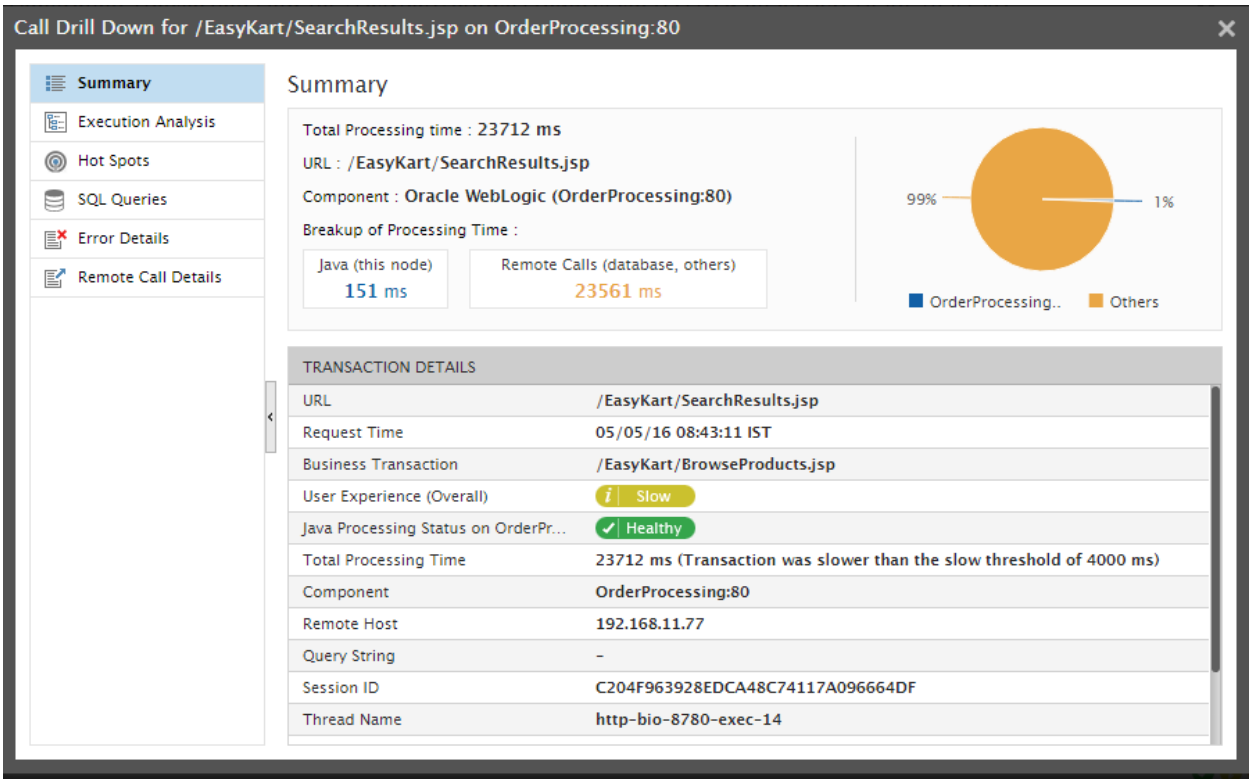


Figure 1.36: A summary of the performance of the JVM node, OrderProcessing:80

The **Summary** section provides a quick summary of the performance of the monitored transaction, EasyKart/BrowseProducts.jsp, on the JVM node that executed the slow database query – i.e., the Oracle WebLogic server, OrderProcessing:80.

From the **Summary**, you can infer that the BrowseProducts transaction was processed for a total of 23712 milliseconds on the OrderProcessing:80. If you take a look at the transaction topology now (see Figure 1.37), you will be able to understand that this processing time is the sum of the following:

- The time for which the transaction was processed internally by the Oracle WebLogic server – 151 ms
- The time taken by OrderProcessing:80 to execute a database query for the transaction and retrieve results – 23242 ms
- The time taken by OrderProcessing:80 to make a JMS call to a messaging server and pull data from the message queue OrderQueue – 319 ms

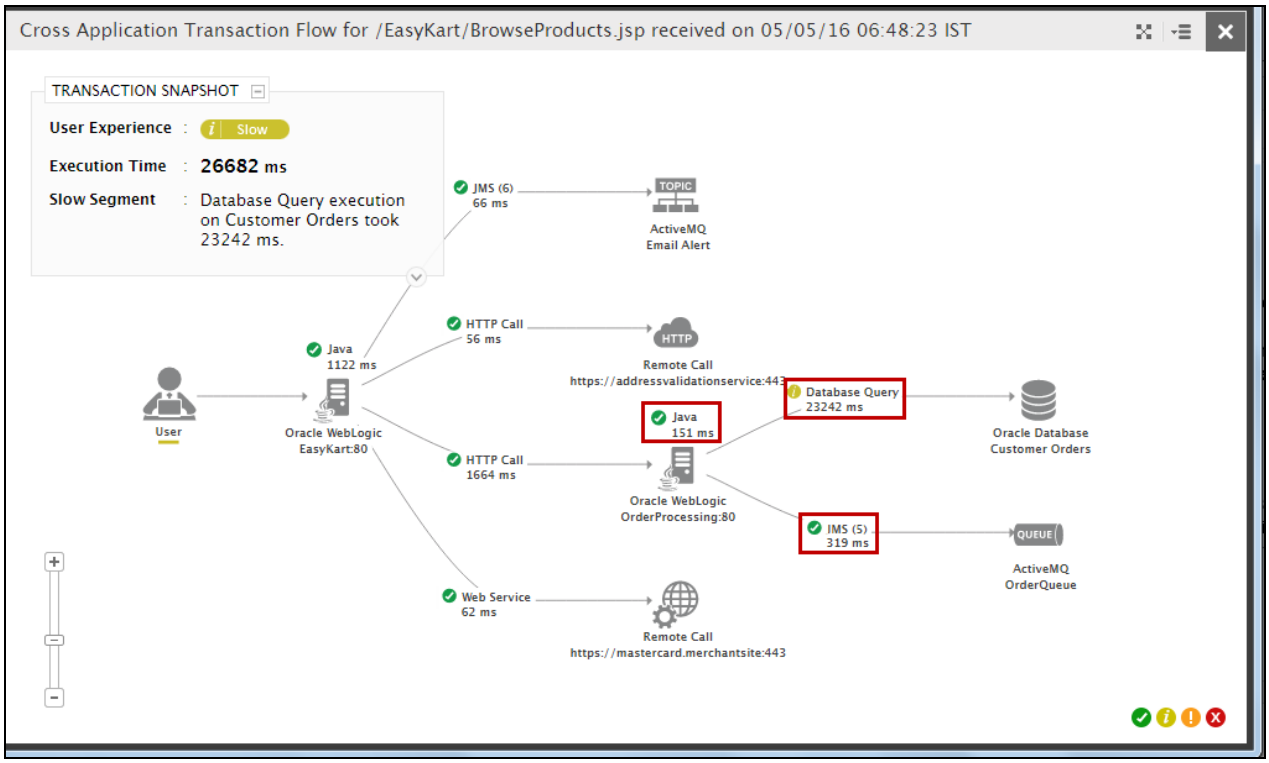


Figure 1.37: How the total processing time of the transaction on OrderProcessing:80 is computed

The **Breakup of Processing Time** section in Figure 1.36 clearly indicates how the **Total Processing time** is computed. From this section, you can also glean where the slowdown originated – within the JVM node? Or when making external calls from the JVM node? In the case of our example, the problem is with the remote calls.

Next, take a look at the **URL** displayed in the **Summary** section. As you can see, while the **Business Transaction** continues to be BrowseProducts.jsp, the **URL** is EasyKart/SearchResults.jsp. When tracing a transaction, if an HTTP call is made by a JVM node to another, then eG BTM not only discovers the type of call made, but also discovers the URL that was called.

This means that in the case of our example, when the user accessed the EasyKart/BrowseProducts.jsp on the EasyKart:80 server, the BrowseProducts page made an HTTP call to the OrderProcessing:80 server and hit the URL EasyKart/SearchResults.jsp. eG accurately discovered the exact URL that the BrowseProducts transaction accessed on the OrderProcessing:80 server and displayed that URL – i.e., EasyKart/SearchResults.jsp – against **URL** in Figure 1.36. Additionally, the **Summary** section also reports the **Query String** of the URL, the **Session ID** of the session in which the transaction is processed on the OrderProcessing:80 server, and **Thread Name** of the thread that processes the transaction.

The **Summary** section also differentiates between the overall **User Experience** of a transaction and the **Java Processing Status** of that transaction on a particular JVM node. In the case of our example, the **Summary** section clearly reveals that the **User Experience** of the tranaction is Slow. At the same time, eG has also detected that the transaction slowdown is not owing to the OrderProcessing:80 server – i.e., the slowness did

not occur because of a processing bottleneck on the OrderProcessing:80 server. This is why, eG maintains that the **Java Processing Status of the OrderProcessing:80** server is Healthy.

1.6.2 Detailed Diagnostics Revealing that a Slow JVM Node is Causing Transactions to Slowdown

Let us consider the example of the BTMDemoFiles web application, where the following transactions are either slow or stalled.

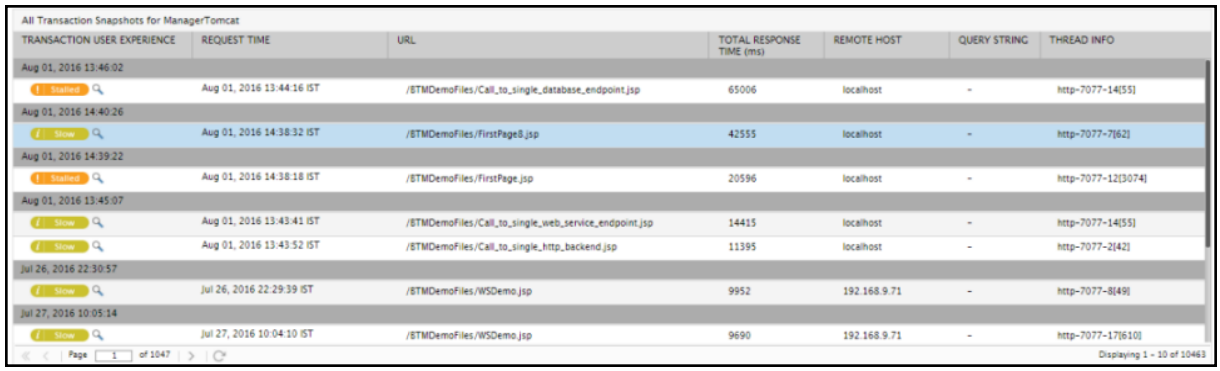


Figure 1.38: Detailed diagnosis of the Avg response time measure

Let us focus on the slow /BTMDemoFiles/FirstPage8.jsp in Figure 1.38. To zoom into the transaction, click on it. The flow of the FirstPage8.jsp transaction will then be displayed as depicted by Figure 1.39.

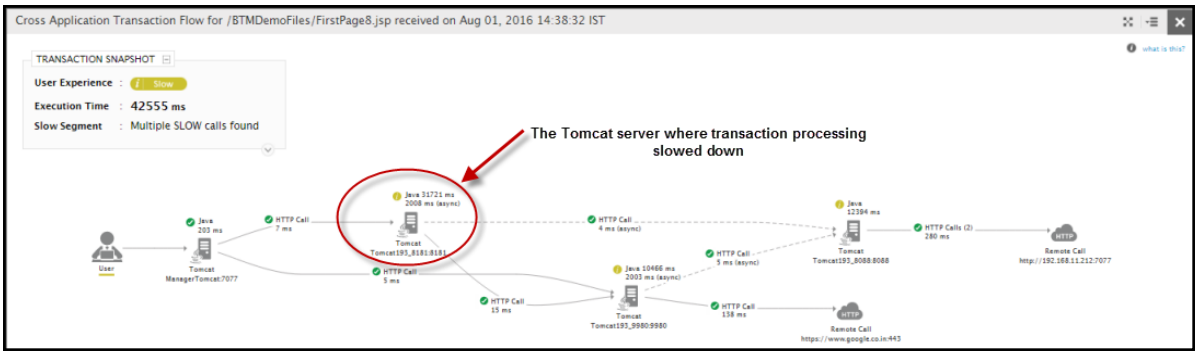


Figure 1.39: The cross-application flow of the FirstPage8.jsp transaction

From the transaction flow, it is evident that the transaction slowed down in 3 JVM nodes. By comparing the time the transaction spent in all 3 nodes, it can be inferred that maximum delay occurred on the Tomcat server, Tomcat193_8181:8181. The question now is what type of processing on the Tomcat server delayed the transaction in question. A closer look at the Tomcat server icon in Figure 1.39 will answer this question as well! As indicated by Figure 1.39, the Tomcat193_8181:8181 server processed Java methods synchronously for over 30000 milliseconds and asynchronously for over 2000 milliseconds. Comparing the two execution times points the needle of suspicion towards the synchronous Java calls made by the Tomcat server. If so,

which exact Java method is slowing down the transaction? To identify the same, let us zoom into the Tomcat server by clicking on it in Figure 1.39. An intermediate window depicted by Figure 1.40 will then appear.

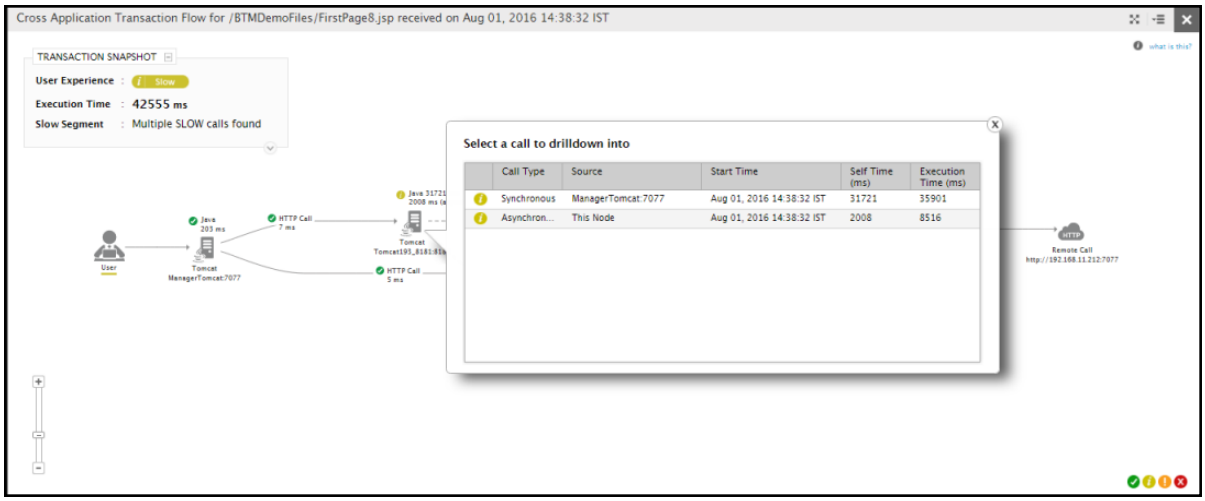


Figure 1.40: An intermediate modular window

This intermediate window will appear only under the following circumstances:

- If a node receives and processes multiple synchronous / asynchronous requests from one/more external sources; and/or
- If one/more asynchronous threads are invoked by a node in response to requests to it;

Typically, from this window, you will be able to quickly determine the number of synchronous and asynchronous calls that a particular JVM node processed. In the case of our example, we can clearly infer from the intermediate window that the Tomcat193_8181:8181 server executed a single synchronous call and a single asynchronous call.

For each synchronous and asynchronous call, this window will also display the self execution time and total execution time of that call. Self execution time is the time it took for the synchronous/asynchronous call to perform Java processing alone. Total execution time is the time taken by the synchronous/asynchronous call to perform both Java and non-Java (eg., HTTP, Database, etc.) processing. By comparing the self and total execution times across calls, you will be able to accurately identify the exact call that took too long to execute, the call type, and whether such a call was slow in processing Java or non-Java. Accordingly, we can clearly deduce from the intermediate window of Figure 1.40 that the synchronous calls made by the Tomcat193_8181:8181 server in our example performed Java processing for a much longer time than desired. To be able to precisely identify the exact Java method that caused the delay, click on the synchronous call in Figure 1.40.

Figure 1.41 will then appear.

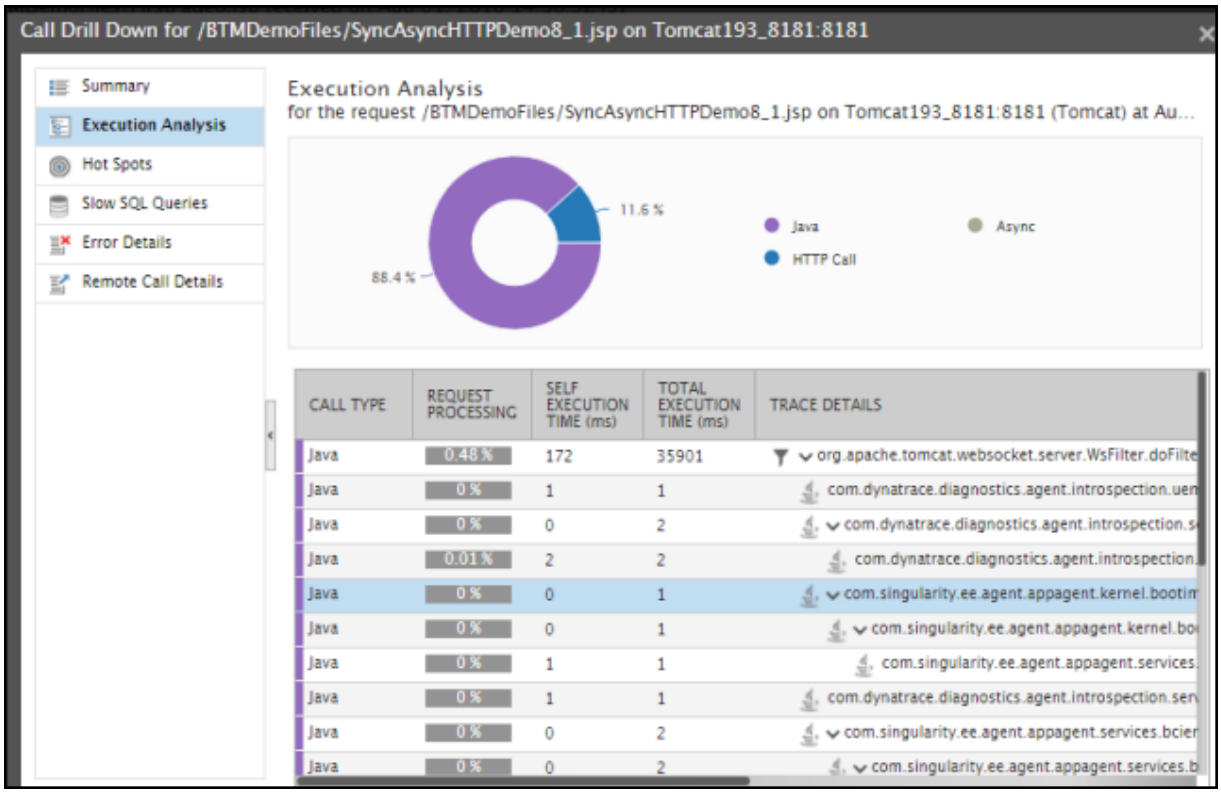


Figure 1.41: The call graph of the synchronous call

Figure 1.41 provides a detailed **Execution Analysis** of the synchronous call. As part of this analysis, a pie chart is presented that quickly reveals the percentage of time the Tomcat server in our example spent processing the server's Java code and making external JMS / SAP JCO / SQL query calls. The table below the pie chart in Figure 1.41 lists the exact methods that performed Java processing or made the remote calls. A quick look at this table reveals that the Java method, org.apache.tomcat.websocket.server.WsFilter.doFilter (ServletRequest,ServletResponse,FilterChain), invoked a series of child methods and external calls, which together took 35901 milliseconds to execute. Scrolling down the table (seeFigure 1.42) points you to exact Java method that took maximum execution time. In the case of our example, it is the sun.net.www.protocol.http.HttpURLConnection.connect() method, which took over 20000 milliseconds for execution.

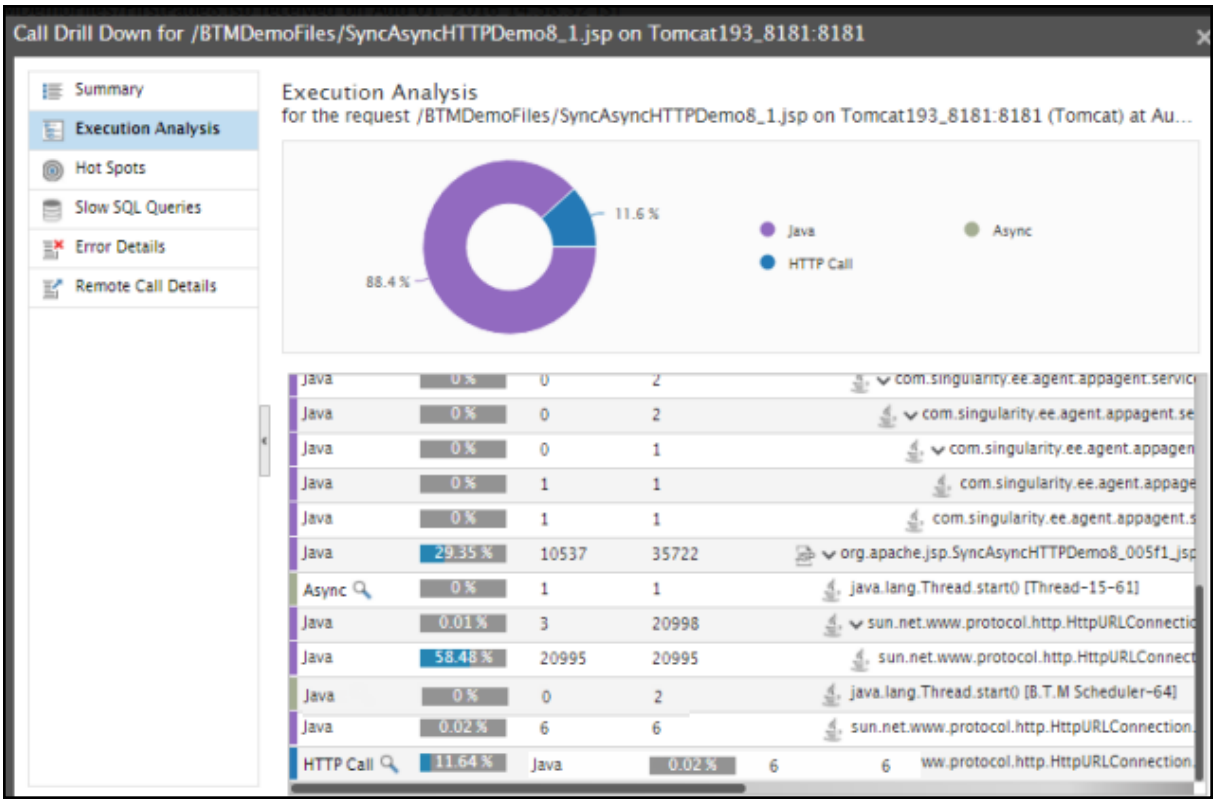


Figure 1.42: The Execution Analysis window pointing you to exact method that contributed to the slowness

Interestingly, Figure 1.42 also reveals that the parent Java method made an asynchronous (Async) call as well. Is it the same call that took over 8000 milliseconds for execution, as per the intermediate window of Figure 1.40? Let's find out. For that, click on the magnifying glass icon adjacent to the call type 'Async' in Figure 1.42. Figure 1.43 will then appear.

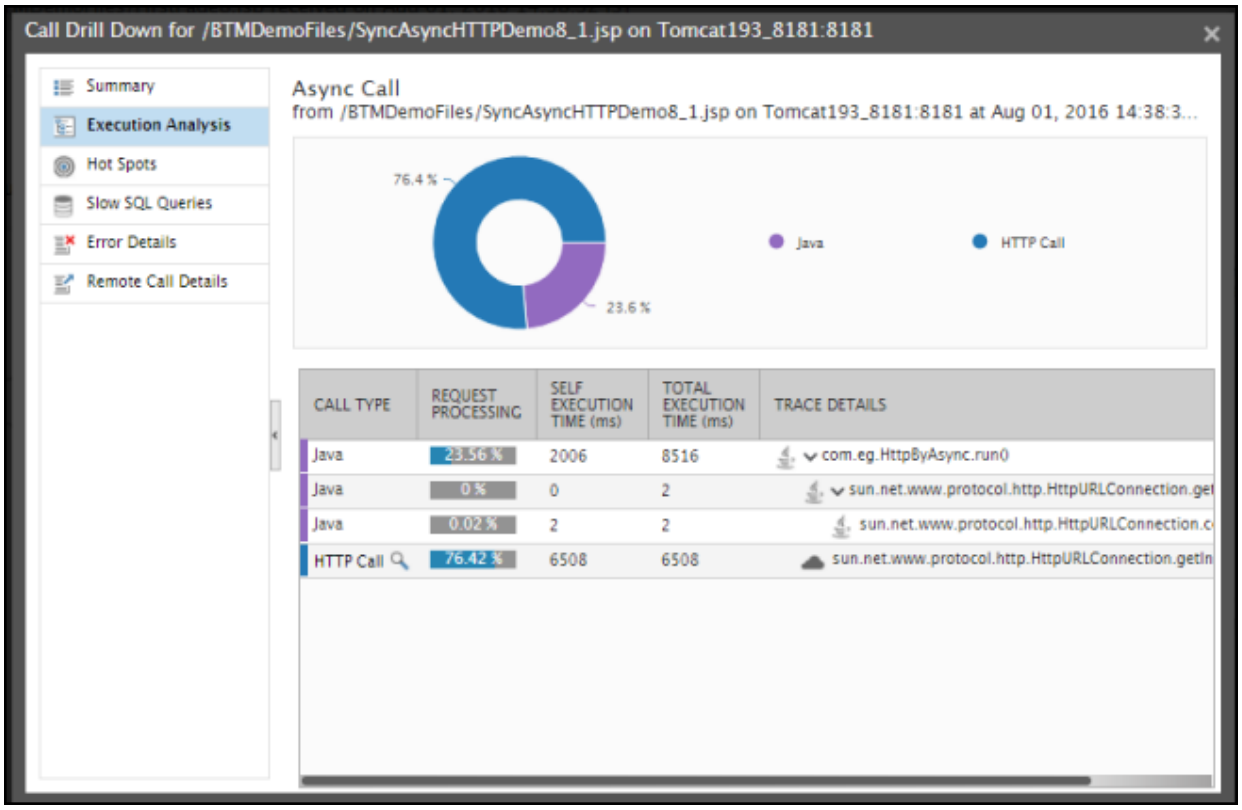


Figure 1.43: The Execution Analysis of the Async call

The Async call executed a Java method that totally took 8516 ms to execute. This is the same execution time that the Async call displayed in the intermediate window of Figure 1.40 has registered. This proves that both calls are one and the same! The intermediate window also reveals that the asynchronous call is slow as well. The Execution Analysis window points you to an HTTP call that this asynchronous thread made that took over 6500 seconds! It is clear that it is this HTTP call that slowed down the asynchronous processing!

This way, eG BTM enables you to diagnose the root-cause of slowness in your synchronous and asynchronous calls using just a few mouse clicks!

1.6.3 Detailed Diagnostics Revealing the Root-cause of an Error Transaction

The detailed diagnosis of the Error transactions measure reveals the complete URLs of the error transactions of a particular business transaction pattern. The total response time of each error transaction and the time at wihich every such transaction was requested can be ascertained from the detailed diagnosis. To zoom into the nature of the error and where it occurred, click on the 'magnifying glass' icon against the corresponding 'Error' icon in the **TRANSACTION USER EXPERIENCE** column of Figure 1.44.

Error Transaction Snapshots for EasyKart

TRANSACTION USER EXPERIENCE	REQUEST TIME	URI	TOTAL RESPONSE TIME (ms)	REMOTE HOST	QUERY STRING	THREAD INFO
May 09, 2016 08:41:12						
Error	05/09/16 08:38:36 IST	/EasyKart/StoreLocato...	25013	192.168.11.189	-	http-bio-8780-exec-14[1...
Error	05/09/16 08:38:14 IST	/EasyKart/StoreLocato...	25013	192.168.11.189	-	http-bio-8780-exec-14[1...

Figure 1.44: The detailed diagnosis of the Error transactions measure

1.6.3 will then appear, which will chart the entire path of the error transaction end-to-end. Using conventional color-codes, this visual representation will accurately pinpoint where the error has occurred

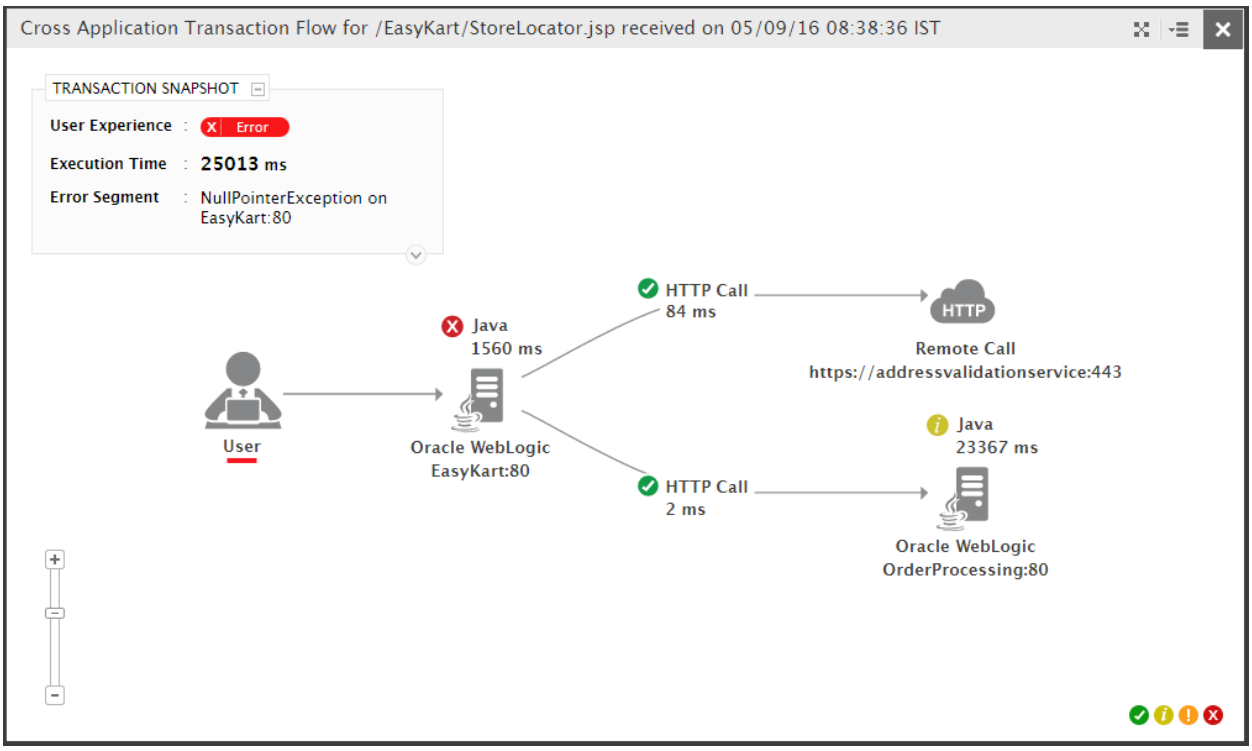


Figure 1.45: The error transaction path revealing where the error has occurred

In the example of 1.6.3 above, the error seems to have occurred on the EasyKart:80 (Oracle WebLogic) server being monitored. To know what the error is, click on the EasyKart:80 server in 1.6.3.

Figure 1.46 that appears next opens an **Error Details** section, which displays the complete details of the error.

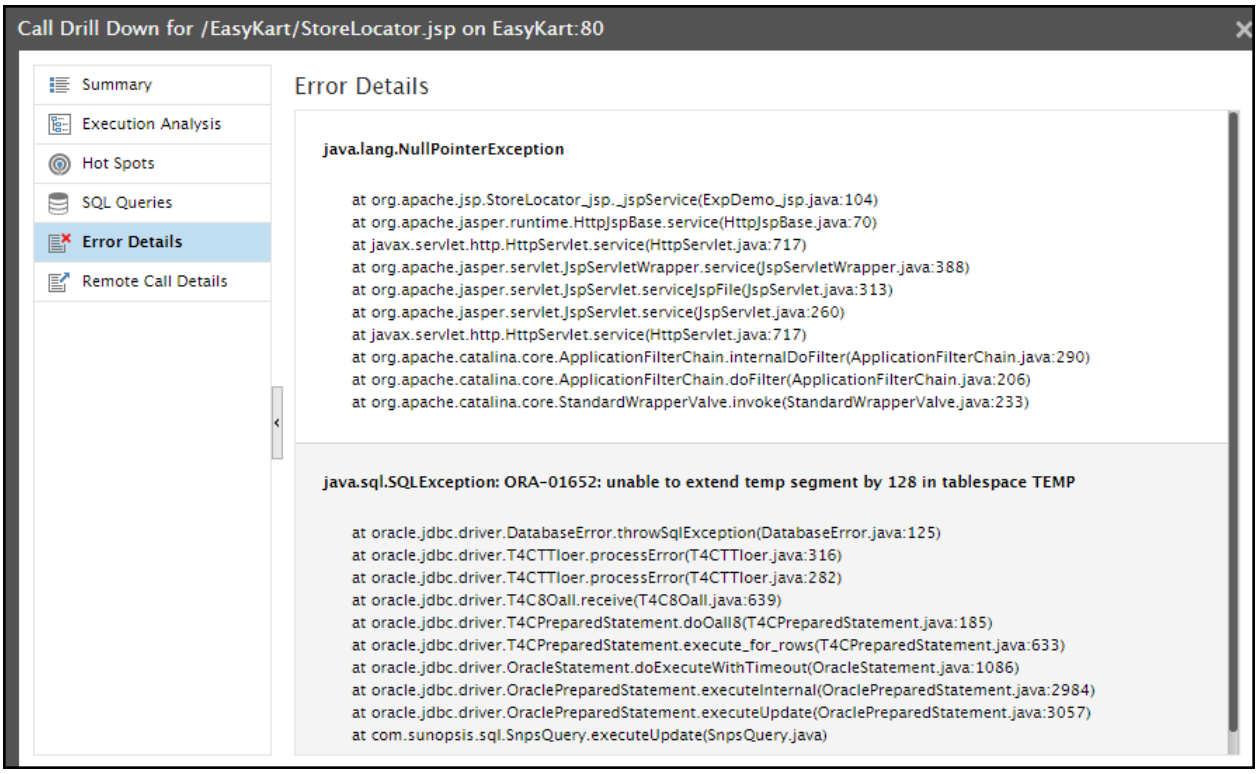


Figure 1.46: Error details

1.6.4 Detailed Diagnostics Revealing that a Remote Service Call is the Reason Why a Transaction Slowed Down

According to Figure 1.47 below, slowness has been detected in 9 transactions of the pattern, /EasyKart/PaymentPage.jsp. To know the exact URLs of the slow transactions, click on the 'magnifying glass' icon against Slow transactions in Figure 1.47.

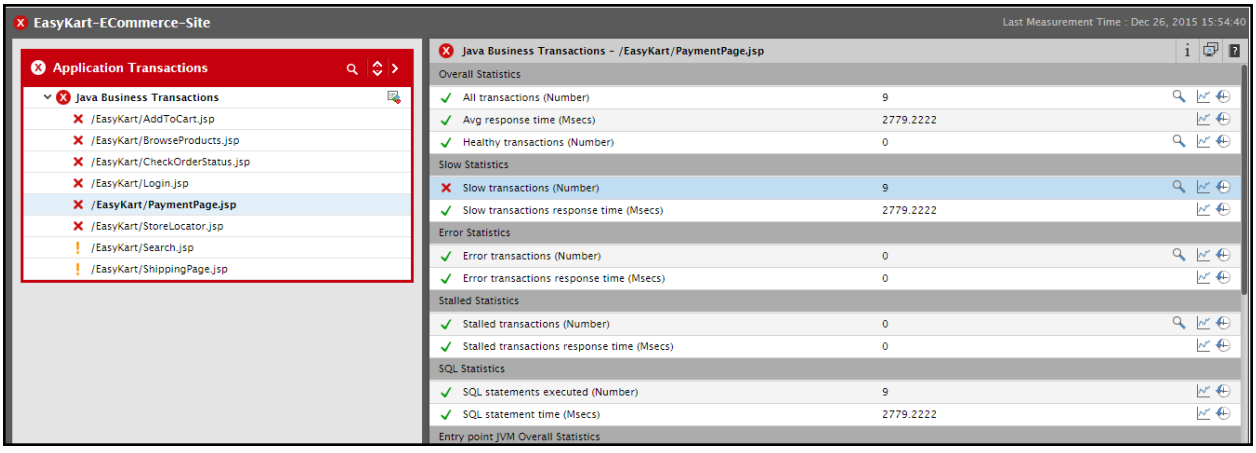


Figure 1.47: The Layers tab page revealing that 9 transactions of the pattern /EasyKart/PaymentPage.jsp are slow

Figure 1.48 will then appear listing the slow transactions URLs. To drill down to the source of the slowness of any of these transactions, click on the 'magnifying glass' icon alongside the 'Slow' icon of that transaction.

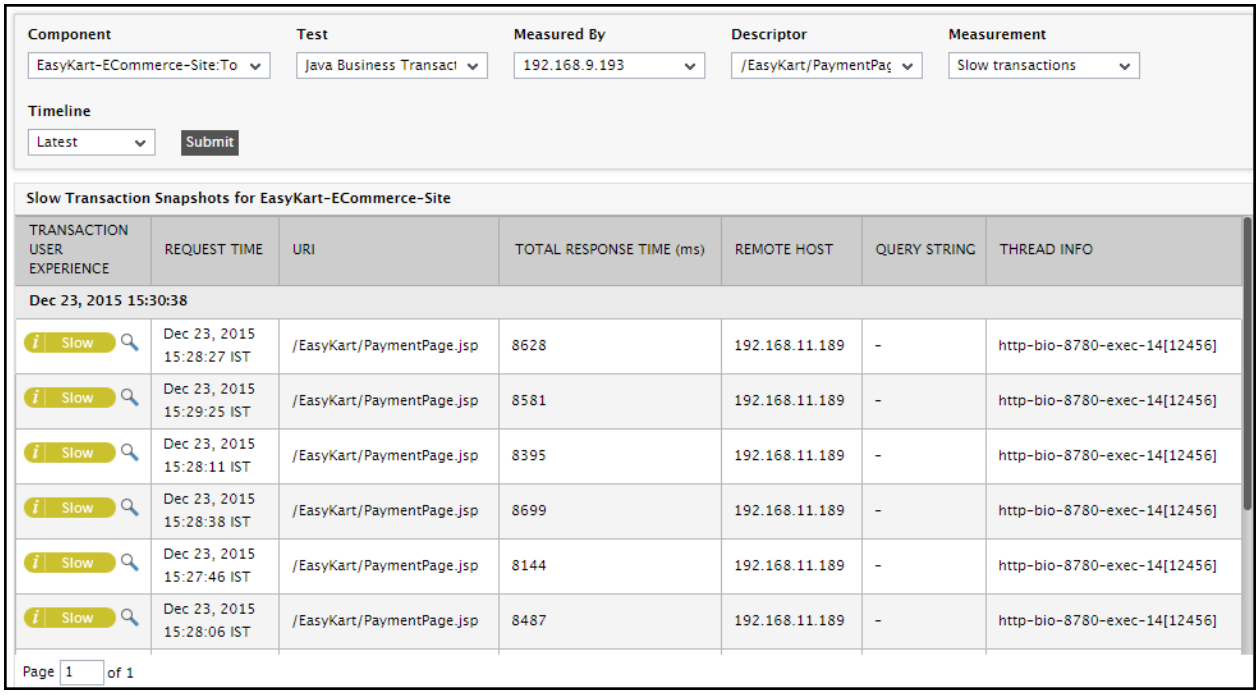


Figure 1.48: Detailed diagnosis listing the slow transactions of the pattern /EasyKart/PaymentPage.jsp

Figure 1.49 will then appear depicting how the transaction flows across the JVM and non-JVM nodes in its path. From Figure 1.49, it is clear that a Web service call made by the Tomcat OrderProcessing server to a mastercard site in the backend – probably for processing a credit card payment - is slowing down the transaction.

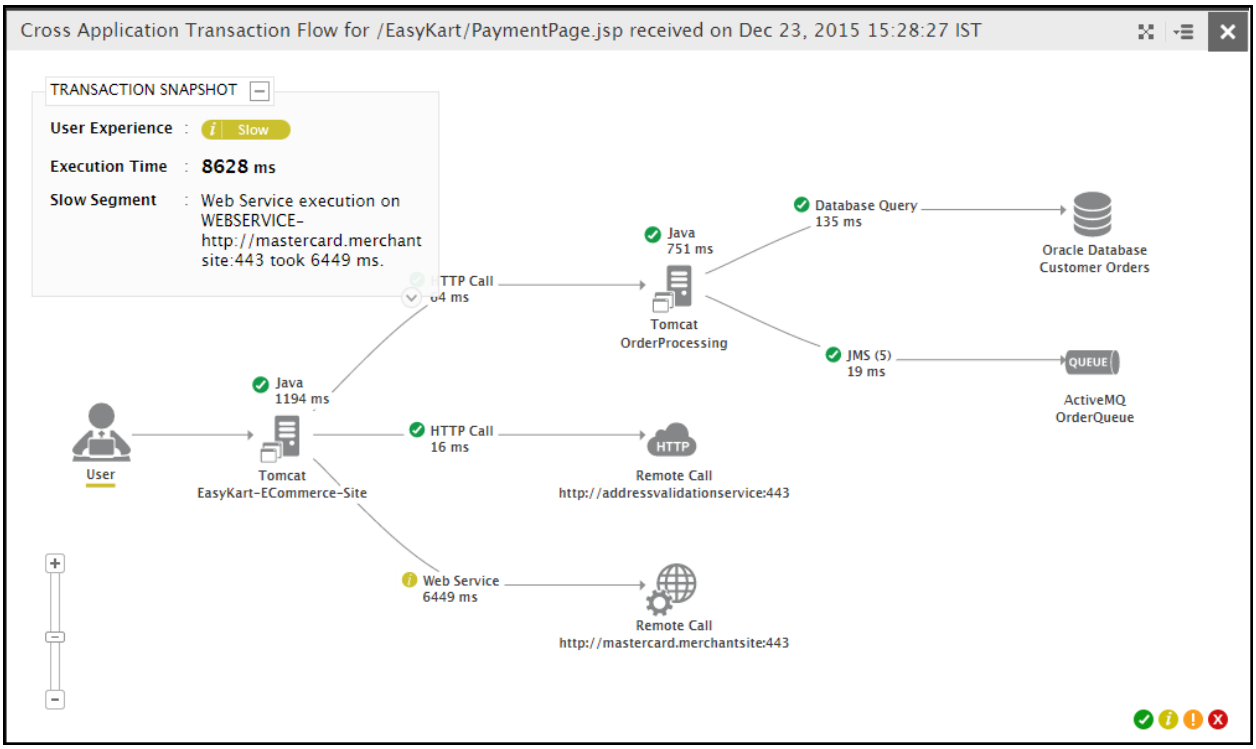


Figure 1.49: Cross-application transaction flow depicting that the problem is with the Web Service call

To know more about this call, click the Web Service icon in Figure 1.49. A **Remote Call Details** window will then open listing all the remote calls made by the Tomcat OrderProcessing server. From this window you can infer that the Web Service call made to the mastercard site is consuming nearly 75% of the transaction execution time. As you can see, a few quick mouse clicks from a Slow transaction in Figure 1.48 has lead you to the precise web service call that is delaying the transaction.

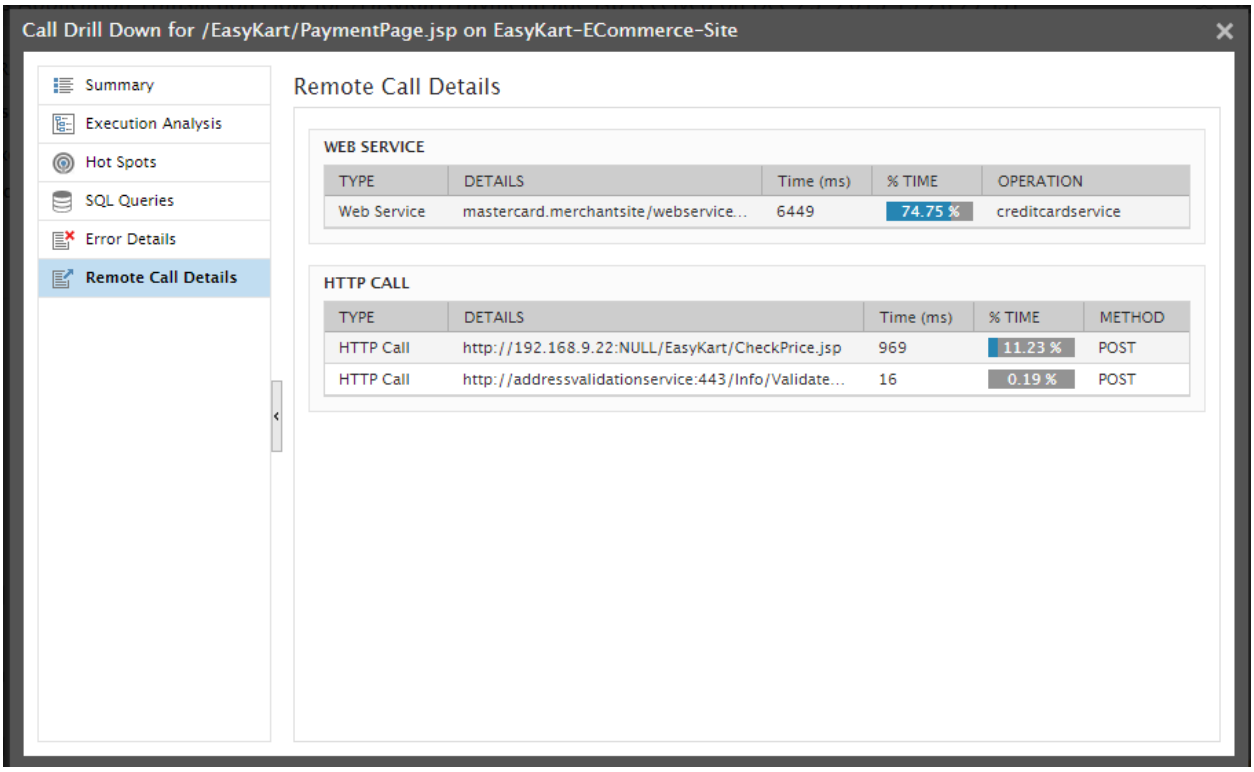


Figure 1.50: List of remote service calls made by the Tomcat OrderProcessing server

2

2

Conclusion

This document has clearly explained how eG Enterprise monitors **Business Transactions**. For more information on eG Enterprise, please visit our web site at www.eginnovations.com or write to us at sales@eginnovations.com.