HP OpenView
Business Process Insight

Integration Training Guide
Customizing Dashboards

Software Version: 02.00

January 2006

© Copyright 2005, 2006 Hewlett-Packard Development Company, L.P.

Legal Notices

Warranty

Hewlett-Packard makes no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
United States of America

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Copyright Notices

© Copyright 2005, 2006 Hewlett-Packard Development Company, L.P.

No part of this document may be copied, reproduced, or translated into another language without the prior written consent of Hewlett-Packard Company. The information contained in this material is subject to change without notice.

Trademark Notices

Java™ is a US trademark of Sun Microsystems, Inc.

Microsoft® is a US registered trademark of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Windows® and MS Windows® are US registered trademarks of Microsoft Corporation.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Support

Please visit the HP OpenView web site at:

http://www.managementsoftware.hp.com/

This web site provides contact information and details about the products, services, and support that HP OpenView offers.

You can also go directly to the support web site at:

http://www.hp.com/managementsoftware/support

HP OpenView online software support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valuable support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit and track progress on support cases
- Manage a support contract
- Look up HP support contacts
- Review information about available services
- Enter discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and log in. Many also require a support contract.

To find more information about access levels, go to:

http://www.hp.com/managementsoftware/access_level

To register for an HP Passport ID, go to:

http://www.managementsoftware.hp.com/passport-registration.html

contents

Chapter 1 Using The OVBPI Dashboard 11
A Generic Dashboard 12
Starting the Dashboard 13
Basic Operation 14
Flow Status 15
Node Rates 16
My Flows/My Services 17
My Flows 17
My Services 18
Superseded Flows 19
Default View 19
My Flows 20
Do Not Show Superseded Flows 20
Directory Structure 21
Configuration 22
Maintaining The Dashboard 24
Localization 25
The i18n Tag 25
The i18n Bundle 25
Text Within the JSPs 26
Obvious Labels 28
Tomcat Specific Settings 29
Tomcat Startup Options 29
Stdout/Stderr Log Output 32

Lab - Using the OVBPI Dashboard 33
My Flows 33
My Services 34
Superseded Flows 34
New Browser 35

Chapter 2 Customizing The Dashboard 37
Basics 38
Using the OVBPI Database 38
Required Skill Sets 38
Tomcat JSP Compilation Logs 39
More Directory Structure 40
Creating a Custom Dashboard 40
Architectural Overview 41
Java Server Pages (JSPs) 42
Taglibs 43
Available Libraries 43
Documentation (Javadocs) 44
Basic Operation 44
Two Types of Tags 46
Java Beans 50
Documentation (Javadocs) 50
The Next Step 51

Chapter 3 Working With Flows 53
<flow> Tag Hierarchy 54
Accessing Flow Details 56
Listing Flow Definitions 56
Filtering Flows By Name 57
Multiple Flow Names (<util:args>) 58
Filtering By Flow Status 59
Getting the Flow ID 61
Listing Node Instance Details 62
Drawing Flow Diagrams 64
Flow Diagram <flowImage> 64

Flow Instance Diagram <flowInstanceImage> 66
Flow Instance Timeline <flowInstanceTimelineImage> 68
Setting Background Color 69
Activating Node URLs 70
Showing Metric Flags 72
Further Customization 72
Flow Annotations 73
DefaultFlowAnnotationBean 74
Developing Your Own Annotation 76
How to Write an Annotation 79
Setting Left Text 87
Setting Right Text 89
Setting Left Text Color 91
Setting Right Text Color 92
Setting Node Label Text 93
Setting Node Label Text Color 94
Setting Node Tooltip 95
Setting Node Image 97
Example Flow Diagrams 99
The OVBPI Dashboard 101
Custom Flow Drawing 101
Flow Drawing 101
Flow Instance Drawing 102
Example - Flow Diagram 103
Example - Flow Instance Diagram 104
Lab - Drawing Flows 106
Basic Flow Settings 106
Custom Left/Right Node Text 106
Custom Colors 107
OVBPI Dashboard 107

Chapter 4 Working With Sliders 109
The Slider Bean 110
Javadocs 110
Setting Up The Slider 110
Producing a Slider Picture 112

Displaying a Slider 113
Resulting HTML Page 114
Setting Range Colors 114
Multiple Sliders on a Page 116
Adding a URL 117
Lab - Drawing Sliders 118
Basic Flow Definition List 118
Adding a Slider 118
Linking the Slider with a URL 118

Chapter 5 Working With Metrics 119
Definitions 120
Code Examples 122
Statistical Data and Graphs 126
metricStatistics 126
metricStatisticsList 127
buildGraphDataset 127
statisticalGraph 127
Code Examples 128
Instance Values 137
flowInstanceMetricValueList 137
flowInstanceMetricValue 138
Code Example 138
Dials 140
Code Examples 141
Alerts 151
raisedAlertList 151
latestRaisedAlert 151
maximumAlertStatus 152
Code Example 152

Chapter 6 Direct OVBPI Database Access 155
Connecting 156
Issuing SQL Statements 158
Fixed SQL Statements 158
Prepared SQL Statements 161

Which One to Use? 163
Additional Helper Beans 164
Constants Bean 164
DBSql Bean 165
Associated Data Table 166
Getting the Name of the Data Table 166
Displaying the Associated Data Table Name. 168
SQL Issues 169
Deadlocks When Using a Microsoft SQL Server Database. 169
Select Columns by Name 170
Example Customizations 171
Listing Specific Flow Instances 171
Joining the Associated Data 177
Lab - Direct SQL. 182
Time to a Node. 182

1

Using The OVBPI Dashboard

OpenView Business Process Insight (OVBPI) provides a Business Process Dashboard which allows you to monitor your business.

The OVBPI Business Process Dashboard enables you to:

- View the overall status of your business and IT services - including links to OpenView Operations (OVO), OpenView Internet Services (OVIS) and OpenView Service Desk (OVSD) information
- View all the flows within your OVBPI system
- View individual flow instances and associated data values
- Visualize your business flows as flow diagrams
- View your business metrics
- View the flows that depend on any given IT service
- ...and much more!

This chapter looks at how to use the OVBPI Dashboard and its basic operation.

A Generic Dashboard

The OVBPI Dashboard is written to visually display information about any business flow. That is, it is a “generic” business dashboard. This makes it extremely useful when you are first developing your business flows, as you are able to get the flow running and immediately view the OVBPI (and IT) statistics. However, not every customer is going to want to report their business in the same way, or with the same look-and-feel.

Although a great deal of care has been taken to provide a well engineered, fully functional, fully localizable, business process dashboard - with an intuitive user interface - it is not intended to be the only way to view your business information. Indeed, the OVBPI Dashboard is provided more as an example of how you might build a business dashboard; see Chapter 2, Customizing The Dashboard for details of how to extend and customize the OVBPI Dashboard.

Starting the Dashboard

The OVBPI Dashboard runs within a Web browser. Before starting the OVBPI Dashboard, make sure that you have started the `Servlet Engine` component using the OVBPI Administration Console.

To start the Dashboard you can either select:

`start->Programs->HP OpenView->Business Process Insight->Dashboard 2.0`

...or...

Start a Web browser and use the following URL:

`http://hostname:44080/ovbpidashboard2-0`

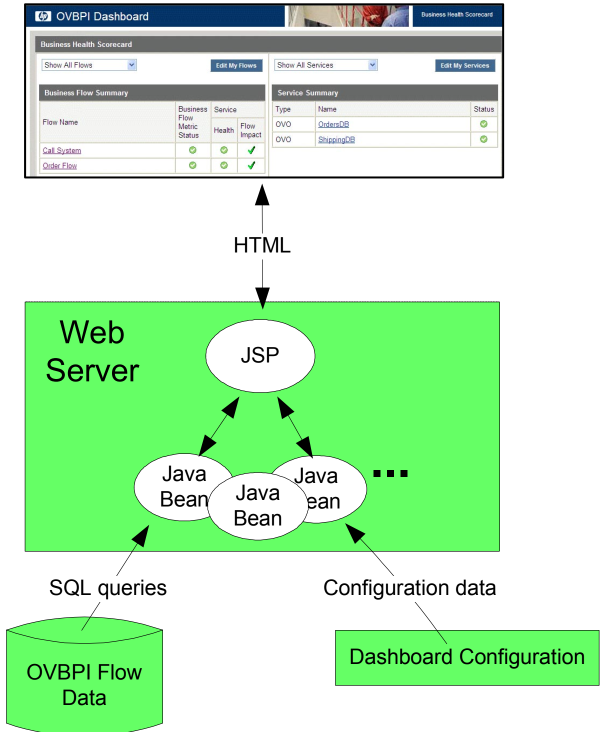
where:

- *`hostname`* is the hostname on which the Servlet Engine component is running
- `44080` is the default port number for the Servlet Engine

Basic Operation

The Dashboard consists of a set of Java Server Pages (JSPs) and Java beans that use the data in the OVBPI database to represent flow information to the browser.

Figure 1 Basic Dashboard Operation



where:

1. When you browse to the Dashboard URL, a JSP is invoked
2. The JSP then calls the appropriate underlying Java beans to load up the dashboard configuration and access the OVBPI SQL database. The Java beans retrieve the requested OVBPI data
3. The JSP then formats this data to the user's Web browser

Flow Status

When displaying the status of a flow, the Dashboard uses the following terms:

- `Blocked`

  This means that at least one instance of the flow is active in a node that cannot proceed due to a problem in an underlying IT service.

- `At Risk`

  This means that there are no flow instances in a blocked state, however there are flow instances coming along that may hit the blockage sometime in the future.

- `Healthy`

  This means that all active flow instances are currently passed the blockage and can therefore continue to run to completion.

Node Rates

Within the OVBPI Dashboard you can display the overall details of a flow and then drill into any node to see overall statistics for that node. Such statistics include the current node `Instance Rate` and current node `Weight Rate`. These rate values give you an idea of the current throughput at a given node. These rates are expressed as a value per hour - in much the same way as the speedometer in your car tells you how far you would travel in a hour if you kept traveling at your current speed.

Be aware that these rate values can give slightly incorrect values when the node activity is low.

The rates eventually reset themselves to zero, if there is a prolonged period of inactivity.

My Flows/My Services

The main screen of the OVBPI Dashboard - the `Business Health Scorecard` screen - shows you all your currently deployed flows and associated IT services. The idea is that on one screen you see right across your business.

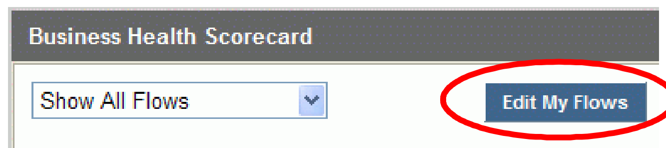
What if you are only interested in one, or a few, of the listed flows? Maybe you are responsible for monitoring the `Orders` Flow, and therefore do not really wish to have the other flows listed on your view of the OVBPI system?

You can tell the Dashboard which flows, and which IT services, you wish to see when you enter this screen.

My Flows

When you run the Dashboard, you can click on the `Edit My Flows` button:

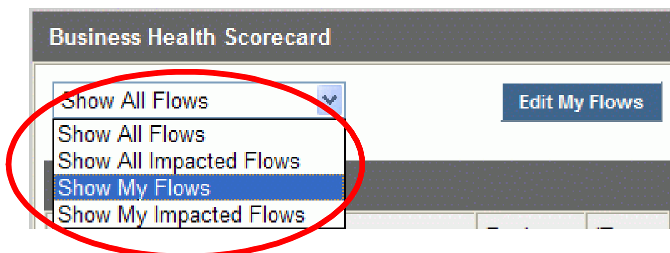
Figure 2 Edit My Flows



You are then presented with a list of available flows. You simply select the one(s) that you are interested in, and press the `Save` button.

Back at the main screen, you can now click on the select list and choose `Show My Flows`:

Figure 3 Show My Flows



The page now shows only those flows that you selected.

You can also select to only show your flows if they are currently impacted in some way.

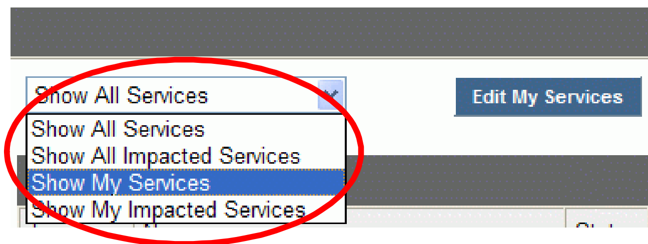
My Services

Just as you can specify the flows that you are interested in, you can specify the IT services that you are interested in.

Through the use of the `Edit My Services` button you can select the IT service(s) that you wish to monitor.

You can then specify that you only wish to see these services - by selecting `Show My Services` in the service pull down:

Figure 4 Show My Services



You can also select to only show your IT services if they are currently impacted in some way.

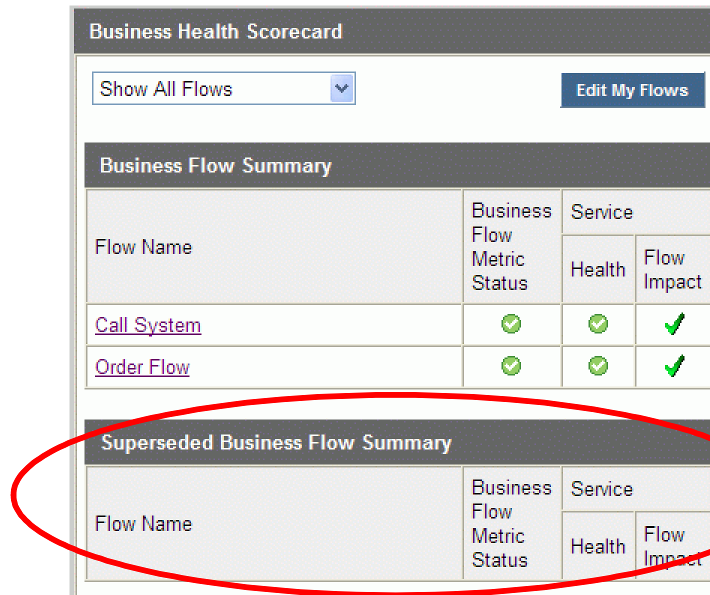
Superseded Flows

When you re-deploy a business flow, the OVBPI Engine wants to delete the previously deployed version of the flow. However, if there are still flow instances for this old flow in the system then the Engine cannot simply delete them. So, in this situation, the OVBPI Engine marks the previously deployed flow as “superseded”.

Default View

By default, the OVBPI Dashboard lists all your superseded flow definitions. These are shown on the main `Business Health Scorecard` screen underneath the main flow summary table.

Figure 5 Superseded Flow List



Listing the superseded flows like this is useful when you are developing your flows as you are able to see when/if your flows have become superseded.

However, what if you are giving a demonstration of the product and you have re-deployed the demo flow a number of times - and therefore have many superseded versions of the demo flow. Do you really want these superseded flow versions displayed across the main screen for the customer to see? Probably not; see Do Not Show Superseded Flows on page 20 to find out how to turn off display of superseded flows.

My Flows

By default, if you have selected the `Show My Flows` option on the main `Business Health Scorecard` page, the Dashboard also list superseded versions of your selected flows.

Do Not Show Superseded Flows

If you do not want the Dashboard to list superseded flows then you can disable this feature.

The steps to turn off display of superseded flows are as follows:

- Run the OVBPI Administration Console
- Select `Business Process Dashboard/General Settings` in the left-hand navigation pane
- Un-check the option `Show superseded flows?`
- Press the `Apply` button

  When the `Confirmation applied` dialog appears, click `OK`
- Refresh your Web browser, or start up a new OVBPI Dashboard, and the superseded flow table is no longer shown

  This change affects **all** OVBPI Dashboards.

Directory Structure

The Dashboard is installed as part of OVBPI in the following directory:

OVBPI-install-dir\nonOV\jakarta-tomcat-5.0.19\webapps\ovbpidashboard2-0

The Dashboard uses Tomcat as its Servlet Container.

The following is a brief explanation of the main sub directories that make up the Dashboard:

- gen

  This directory contains the JSPs that make up the Dashboard.

- style

  This directory contains the cascading style sheets (CSS) that determine the look-and-feel for the Dashboard.

- images

  This directory contains some images used by the JSPs.

- images\generated

  The Dashboard saves all run-time pictures (flow diagram, timelines, etc.) into this directory. You should see files created in this directory as users run the Dashboard; refer to Maintaining The Dashboard on page 24 for details of how to clean up these files.

- WEB-INF

  This directory contains two sub directories:

  – classes

    This contains the source code for some of the additional Java beans used by the JSPs.

    This is also where the Dashboard's configuration file is located; see section Configuration on page 22.

  – lib

    This contains the JAR files (Java code libraries) required by the JSPs.

Configuration

The Dashboard reads its configuration settings from the file: `WEB-INF\classes\DashboardConfig.properties`.

Although you can edit this file directly, it is not recommended. This file is maintained using the OVBPI Administration Console.

So if you need to alter a configuration setting within the Dashboard it is best to use the OVBPI Administration Console.

The most likely options that you might wish to reconfigure are:

- The `Page refresh delay` time

  For example, you could set this to `0` (zero) to disable automatic page refresh.

- The `Show superseded flows?` option

When you use the OVBPI Administration Console to apply configuration changes to the Dashboard, it actually rebuilds the `WEB-INF\classes\DashboardConfig.properties` file, based on a template found in *`OVBPI-install-dir`*`\newconfig\DataDir\conf\bia`.

Each time a JSP is run from a browser, it checks the time stamp on this `DashboardConfig.properties` file. If the time stamp has changed, the JSP re-reads this new configuration. Any changes made to the configuration affect all running Dashboards when they next access a JSP or refresh their current page.

You should always use the OVBPI Administration Console to make configuration changes to the Dashboard.

Any changes you make directly to the `DashboardConfig.properties` file are overwritten the next time you make **any** configuration changes through the OVBPI Administration Console.

Hidden Settings

The `DashboardConfig.properties` file does contain settings that are not available through the OVBPI Administration Console. These are settings that an administrator might wish to set up at install time and typically never change again. Some examples of these kinds of settings are:

- The images used to show whether a node is active, completed or impacted
- The images used to show underlying IT Service states - of `Normal`, `Warning`, `Minor`, `Major` and `Critical`

To alter such configuration options requires you to edit the template files in the following directory:

OVBPI-install-dir\newconfig\DataDir\conf\bia

The files are named:

```
DashboardConfig.mssql.properties
DashboardConfig.oracle.properties
```

You need to make the changes to both of these files and then make the changes directly to the active `DashboardConfig.properties` file (in your `WEB-INF\classes` directory). The reason you need to make the changes to the template files is in case you make any future configuration changes using the OVBPI Administration Console - this includes any configuration changes, even those not related to the Dashboard.

Maintaining The Dashboard

When the OVBPI Dashboard builds flow diagrams and time lines, these images are created and held on disc, on the Tomcat server, in the directory `images\generated`; see Directory Structure on page 21.

The number of files in this directory should not grow over time as the Java classes that create these files remove them when the Web session comes to an end. Therefore, this directory contains only images that are being used by active Web sessions. Note that, by default, a web session expires after 30 minutes of inactivity. If you close a Web browser, any session flow pictures are removed after a further 30 minutes has elapsed.

However, it looks like these files are **not** cleaned up when you shutdown the Tomcat server :-(So, your web administrator should monitor this directory to ensure that any files left behind are removed.

If the Web administrator wants to remove files from this directory they have two main options:

1. They can shutdown the Servlet Engine (Tomcat) and remove all the files from this `generated` directory
2. They can leave the Servlet Engine running and just remove all files that are over (for example) one day in age. The idea is that you do not want to remove any files that are being served to web browsers at the moment, so you might decide that if the files have been sitting there for more than a day then it is safe to remove them

Localization

The OVBPI Dashboard is fully localizable.

All of the text strings used on the Web pages (JSPs) are held in an external Java resource bundle (external file) allowing you to localize them as required to produce a localized version of the Dashboard.

Let’s take a look at how this is achieved.

The i18n Tag

Rather than a JSP containing hard-coded text strings, each JSP loads up an external file (called a “Resource Bundle”) that contains all the text strings it needs. Each of these text strings has been assigned a unique label (a key).

A Java bean is provided that allows access to a resource bundle. This bean is called: `com.hp.ov.bia.views.taglibs.i18n`.

So the first thing the JSP must do is to say that it wishes to use this Java bean...and give it a name. The line that declares this is as follows:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.i18n" prefix="i18n" %>
```

Now, any markup that begins with `<i18n:` refers to calls supported by the Java bean `com.hp.ov.bia.views.taglibs.i18n`.

The i18n Bundle

The JSP then needs to load up its resource bundle. This is achieved with the following line:

```
<i18n:bundle baseName="dashboard_localization" />
```

This tells the i18n Java bean to go looking for a resource bundle called “dashboard_localization”. Resource bundle file names end in “.properties”, so the default resource bundle therefore has the name: `dashboard_localization.properties`.

However, here is the clever bit... Resource bundles belong to families whose members share a common base name, but whose names also have additional components that identify their locales. So, if, for example, you wanted to make

use of a German version of the dashboard text strings, these could be provided in a resource bundle called `dashboard_localization_`**`de`**`.properties`. Each resource bundle in a family contains the same items, but the items have been translated for the locale represented by that resource bundle. If there are different resources for different countries, you can make specializations: for example, `dashboard_localization_`**`de_CH`**`.properties` would contain the text strings for the German language (de) in Switzerland (CH).

The `dashboard_localization` resource bundles are found within the JAR file:

```
WEB-INF\lib\bia-views-resources.jar
```

Text Within the JSPs

With the resource bundle loaded, the JSPs are able to display this localized text wherever it needs, using the `<i18n:message>` tag.

Let's look at some examples:

Simple Text

Consider the JSP: `flowInstance.jsp`

At the point where the JSP lists the nodes that make up the flow definition, it needs to display a title for this table. Rather than just having the text "Node List", you see the code:

```
<i18n:message key="FlowInstance.nodeList" />
```

This retrieves the text labelled `FlowInstance.nodeList` from the resource bundle. In the English-locale resource bundle there is an entry as follows:

```
FlowInstance.nodeList=Node List
```

Which means that the string `Node List` is returned. So the `<i18n:message...>` tag is replaced in the resulting HTML with the actual text: `Node List`.

Text with Parameters

How does it handle message strings that are not just fixed text?

Within the resource bundle some of the strings are defined to take parameters. In these cases, you see the `args=` option used.

Consider the JSP: `flowInstance.jsp`

At the point where the JSP is listing the associated flow instance data, it wants to display a heading that includes the actual name of the associated data definition.

The code is as follows:

```
<i18n:message key="FlowInstance.assocData"
              args="<%= new Object[] {flowInstanceDataBean.getName()} %>" />
```

For the English-locale resource bundle, the text labelled `FlowInstance.assocData` contains the string: `Associated Data ({0})`

The `{0}` specifies that the first parameter (they number from zero) is to be placed at this point.

The `args=` option needs to pass in a Java Object array (`Object []`) with the zero'th element containing the name of the associated data definition for this flow.

So, if the associated data table name was `Orders/My Data`, the message string would become: `Associated Data (Orders/My Data)`

Obvious Labels

To make it easier for developers to read and understand the JSPs, the labels assigned to each text string have been given meaningful names. The text labels tend to take the form:

```
<area/screen name>.<text name>
```

For example:

- `FlowInstance.nodeList`

  This is to do with a page that displays flow instances, and the text says something about a node list.

- `FlowInstance.assocData`

  This is to do with a page that displays flow instances, and the text displays something about the associated data.

- `ServiceHealth.priority`

  This is to do with a page that displays service health, and the text displays something about the priority.

Tomcat Specific Settings

The Servlet Engine that comes pre-configured with OVBPI is Apache Tomcat.

The version of Tomcat that ships with OVBPI is required for OVBPI components other than the Dashboard. You can use a different Servlet Engine for your customized dashboard; however the version of Tomcat that is installed with OVBPI is the one that is tested and recommended.

There are some Tomcat configuration settings that are set by the OVBPI installation, and they are as follows:

Tomcat Startup Options

When Tomcat is started, the following environment variables are set:

- `CATALINA_HOME`

  This is set to the version of Tomcat that comes installed with OVBPI.

  The setting takes the form:

  `CATALINA_HOME=`*`OVBPI-install-dir`*`\nonOV\jakarta-tomcat-5.0.19`

- `CATALINA_OPTS`

  This is set to pass in the details of the logging configuration for Tomcat.

  The setting takes the form:

```
CATALINA_OPTS=-DOV_INSTALL_DIR="OVBPI-install-dir"
     -DOV_DATA_DIR="OVBPI-install-dir\data"
     -Djava.util.logging.config.file=
          "OVBPI-install-dir\data\conf\bia
                       \bia_tomcat_loggingconfig.properties"
```

Logging

The logging level for the Dashboard is configured through the OVBPI Administration Console.

The OVBPI Dashboard log level is maintained in the file:

```
OVBPI-install-dir\data\conf\bia\
            bia_tomcat_loggingconfig.properties
```

When Tomcat is started, this logging configuration file is passed in (see section Tomcat Startup Options on page 29) and the logging output is redirected to the file:

```
OVBPI-install-dir\data\log\bia_tomcat0_0.log
```

If you want to see more detailed log information then use the OVBPI Administration Console to turn the Servlet Engine logging up to `FINER`, and restart the Servlet Engine.

Any JSP output to standard output (`stdout`) or standard error (`stderr`) is **not** redirected to this log file! (see Stdout/Stderr Log Output on page 32)

Auto Compiling JSPs

When Tomcat runs in development mode, it checks each JSP it displays to see if the page has been modified since the last time it was displayed. If the page has been modified, Tomcat compiles the page and you see this latest version.

The OVBPI installation of Tomcat defaults to a `development` mode.

Running Tomcat in development mode is advantageous when you are developing, but is not recommended for a production environment. The Tomcat documentation recommends that when running in a production environment that you set the development flag to `false`.

To set your Tomcat installation to run in a non-development mode, you need to edit the following file:

OVBPI-install-dir\nonOV\jakarta-tomcat-5.0.19\conf\web.xml

Set the development flag to `false`, then restart Tomcat (The `Servlet Engine` component using the OVBPI Administration Console).

The code segment within this `web.xml` file looks something like the following:

```
<servlet>
 <servlet-name>jsp</servlet-name>
 <servlet-class>org.apache.jasper.servlet.JspServlet
                                            </servlet-class>
...

<init-param>
    <param-name>development</param-name>
    <param-value>false</param-value>
</init-param>

...
```

To set Tomcat back to development mode, set the development option back to `true`, and restart Tomcat.

Stdout/Stderr Log Output

If your JSPs issue any output to stdout or stderr, this output is sent directly to Tomcat's stdout/stderr. This stderr/stdout output is not captured by the OVBPI Servlet Engine log files.

When OVBPI starts up Tomcat, two files are created to capture any stdout/stderr output. These two files are:

```
OVBPI-install-dir\data\log\tomcat_stdout.log
OVBPI-install-dir\data\log\tomcat_stderr.log
```

The `tomcat_stderr.log` file captures any errors that might occur during the start-up of the Tomcat servlet engine.

The `tomcat_stdout.log` file captures **both** stdout and stderr output from your JSPs.

So if your JSPs issue any output direct to stdout or stderr, look in the `data\log\tomcat_stdout.log` file.

If you restart the OVBPI Servlet Engine, this restarts Tomcat and this overwrites any previous output contained within the `tomcat_stdout.log` and `tomcat_stderr.log` files.

Lab - Using the OVBPI Dashboard

The purpose of this lab is to get you using the OVBPI Dashboard and explore some basic configuration options.

This lab assumes that you have already worked your way through the *OVBPI Integration Training Guide - Modeling Flows*, and completed that lab work.

My Flows

- Start up the OVBPI Dashboard

You should see at least the following flows listed:

```
Insurance Claim
Order Flow
```

Let’s configure your Web browser to focus on `Order Flow`:

- Click on the `Edit My Flows` button
- Check `Order Flow`
- Click the `Save` button

Now, back on the main screen:

- Select `Show My Flows` from the pull-down list

Your Web page should now only list the flow: `Order Flow`.

My Services

Let's do the same now for the IT service list, and narrow that down to just show the services that you are interested in:

- Click on the `Edit My Services` button
- Check the two services: `OrdersDB` and `ShippingDB`
- Click the `Save` button

Now, back on the main screen:

- Select `Show My Services` from the pull-down list

Your Web page should now only list the services: `OrdersDB` and `ShippingDB`.

Superseded Flows

Let's now remove any mention of superseded flows from this main screen. For this, you need to run the OVBPI Administration Console:

- Start up the OVBPI Administration Console
- In the left-hand navigation pane, click on the `General settings` option, within the `Business Process Dashboard` option
- In the right-hand pane, un-check the option `Show superseded flows?`
- Press the `Apply` button
- When the confirmation dialog appears - click `OK`

Now, back in your Web browser:

- Refresh your browser...

Your Dashboard main-page is now tailored to focus on the `Order Flow` and your screen is not cluttered-up with any mention of superseded flows.

New Browser

- Close your Web browser (`File->Close`)
- Start a fresh OVBPI Dashboard

Notice that your Dashboard starts up with the My Flow/My Service settings that you previously configured. Very useful!

Well done! You have reached the end of the lab.

2

Customizing The Dashboard

The Dashboard provided with OVBPI is a generic dashboard written to present the flow data in a generic way. Although it is a useful tool for initially monitoring your run time OVBPI system, it is not intended to be the only way for you to access and view OVBPI information. Your business is likely to have its specific needs for business reporting, for example, within an existing portal. Indeed, the OVBPI Dashboard is provided more as an example of how you might build a business dashboard. You can extend and replace the example Dashboard according to your business requirements.

This chapter looks at the basic components that make up the OVBPI Dashboard, and the skill sets required to customize your own dashboard.

Basics

Using the OVBPI Database

The Dashboard derives its statistics and information from the OVBPI database. For example, when the Dashboard needs to determine the state of a particular flow, it simply accesses the OVBPI database, reads the flow's state and reports this to the screen.

The OVBPI database is essentially divided into two main areas:

1. The Business Impact Engine tables

   These tables hold the details for your deployed flows. The tables track things such as: flow instances, node instances, and service impacts.

2. The Metric Engine Tables

   These tables hold all the current, and historical, details for any defined business metrics.

When customizing your own reporting dashboard you can simply extend the example OVBPI Dashboard code, or write your own dashboard.

When writing or extending a dashboard it is important that you understand the OVBPI database schema; the schema is fully described in the *OVBPI System Administration Guide*.

Required Skill Sets

The Dashboard accesses the OVBPI database using SQL calls. These results are then displayed to the Web browser through JSPs - with embedded Java.

The basic skill sets required for anyone who wants to customize the Dashboard are:

- SQL - Structure Query Language
- Java
- JSP - Java Server Pages
- CSS - Cascading Style Sheets

Tomcat JSP Compilation Logs

When you start developing your own JSPs you may get compilation errors.

With earlier versions of Tomcat these compilation errors used to appear in the Web browser that was trying to run the new JSP. However, with the version used by OVBPI, when a compilation error occurs, Tomcat normally displays a **standard error message** in the browser saying that something went wrong and then directs you to the Tomcat log files.

This standard error message contains text similar to the following:

```
No Java compiler was found to compile the generated source for the JSP.
This can usually be solved by copying manually $JAVA_HOME/lib/tools.jar from
the JDK to the common/lib directory of the Tomcat server, followed by a Tomcat
restart. If using an alternate Java compiler, please check its installation
and access path.
```

This message seems to be telling you that your Tomcat is not installed correctly and that you should try copying files around and checking all sorts of things.

...**Stop!...**

This message simply means:

> “There was a compilation error - check the log file.”

You can view these Tomcat log files using the OVBPI Administration Console.

So whenever you see the “No Java compiler...” message, simply run the OVBPI Administration Console and click to view the logs for the `Servlet Engine` component. Scroll down to the end of the file and you should be able to see your compilation error(s).

More Directory Structure

At install time, the OVBPI Dashboard is actually installed into two different locations:

- *OVBPI-install-dir*\nonOV\jakarta-tomcat-5.0.19\webapps\ovbpidashboard

  This contains the installed version of the files. That is, these are the files that are used when you run the OVBPI Dashboard.

- *OVBPI-install-dir*\examples\bia\BusinessProcessDashboard

  This contains a complete copy of the files that make up the OVBPI Dashboard.

  The idea is that these files provide a set of original Dashboard files.

See section Directory Structure on page 21 for details of each sub directory. The directory structure is the same for both the installed and the example dashboards.

Creating a Custom Dashboard

When writing your own custom dashboard you can work under the `webapps\ovbpidashboard` directory.

Make a copy of the `gen` directory and then do your work in the copy directory. For example, if you create a copy of the `gen` directory, called `myDashboard`, then the URL to run your dashboard is:

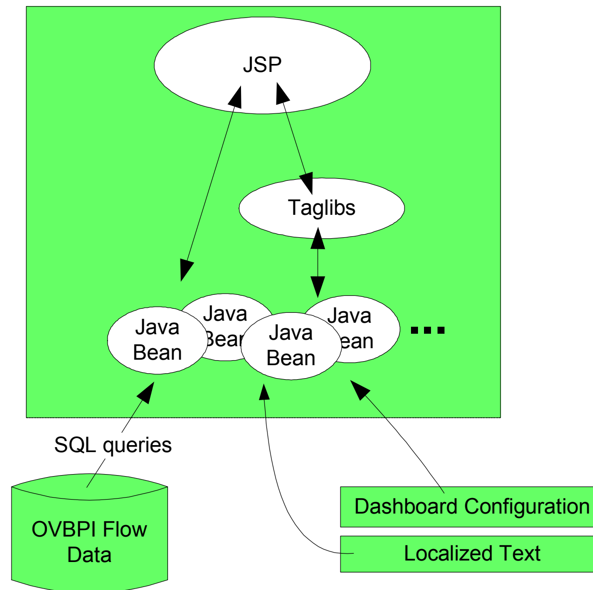
```
http://hostname:44080/ovbpidashboard2.0/myDashboard
```

By working in a copy of the `gen` directory, you leave the `gen` directory to provide the default dashboard behaviour.

Architectural Overview

The OVBPI Dashboard consists of a set of JSPs making use of a collection of JSP custom tag libraries (Taglibs) and back-end Java beans.

Figure 1 Architectural Overview



where:

- There is a set of Java beans that provide access to the OVBPI data base, localized text strings, dashboard configuration, etc. In other words, a set of Java beans that can provide the data.
- The JSPs call methods on these Java beans to retrieve the data they require, and display this in the Web browser.
- To simplify this, most of the common tasks required by the JSPs have been wrapped-up into JSP Taglibs. This greatly simplifies the JSP code as there is little or no need to actually write Java. The JSP can simply invoke a Taglib (written as an HTML tag) and this generates the necessary Java code to call the appropriate Java bean, retrieving the required data from the database.

Let's look at this in more detail...

Java Server Pages (JSPs)

The JSPs are available in the `gen` directory and are designed to be self explanatory.

For example:

- `flowInstance.jsp` shows the details of a selected flow instance
- `instances.jsp` lists the flow instances for a given flow

If you wish to know what a page does, simply run the Dashboard and as you display each screen, the URL shows you which JSP page is being invoked.

There are some JSPs that are reused across multiple pages; these are:

- `common.jsp`

  This holds all the utility methods that are common and used across all the pages. This page is included at the top of other JSP pages.

- `footer.jsp`

  This is called at the end of every page.

- `alertPage.jsp`

  This is called whenever there is an error that the page is coded to handle - such as a required parameter not being passed to a page, etc.

- `errorPage.jsp`

  This is called whenever an unexpected occurs.

The JSP pages are all commented throughout and are the best way to learn how the Dashboard is written and operates.

Taglibs

There are JSP custom tag libraries (Taglibs) for doing most of the standard things required within the dashboard.

Available Libraries

The following tag libraries are available:

- `flow`

  This tag library provided tags that allow you to retrieve flow information from OVBPI. You can get lists of flows, lists of flow instances, flow statistics, flow diagrams, etc..

- `i18n`

  This tag library is how the dashboard is able to use localizable text; see Localization on page 25 for more details about this taglib.

- `metrics`

  This tag library provided tags that allow you to retrieve metric information from OVBPI. You can list metric and threshold definitions, draw statistical graphs, draw metric dials, etc.

- `ovis`

  This tag library allows you to retrieve information from OV Internet Services (OVIS) and display statistics or diagrams.

- `ovsd`

  This tag library allows you to retrieve information from OV Service Desk (OVSD) and display statistics or diagrams.

- `ovsn`

  This tag library allows you to retrieve IT service information from OV Operations (OVO) and display statistics or diagrams (such as service hierarchy maps).

  The taglib is called “ovsn” as it refers to the part of OVO known as “OV Service Navigator”.

- `service`

  This tag library allows you to retrieve specific OVBPI information about the IT services within a flow.

- `util`

  This tag library contains a set of handy utility mechanism - such as error handling. It also provides some utility mechanisms you can use when calling the other tag libraries.

Documentation (Javadocs)

Full documentation for the Taglibs is provided on the OVBPI product CD - under the `docs` directory.

You can read the documentation by using a Web browser and opening the file:

```
docs\html\OVBPITagLibs\index.html
```

This documentation helps you find out the available list of tags and their syntax. However, the best way to really learn about how these tags are used is by reading the JSP code directly, and by referring to the worked examples later in this training guide.

Basic Operation

The taglibs are acting as a layer between you and the database. You specify the tag, and it makes the necessary calls to the OVBPI database (using back-end Java beans) to retrieve the information.

But how does it give you back this information? Most of the tags return their information by passing back a Java bean which contains the information gathered from the OVBPI database.

The javadoc for each tag says what type of Java bean it returns, and there is a separate set of javadocs that describes all these return beans; see Java Beans on page 50 for more details.

Let’s consider an example:

The `<flow:flow>` tag returns details of a given flow. You pass in the specific flow ID that you are interested in, and you are returned a Java bean that contains the flow details for that flow.

You call the tag like this:

```
<flow:flow flowId="<%= flowId %>" var="myFlowBean" />
```

where:

— You pass in the Java variable `flowId` which you have pre-set to be the flow ID of the flow you want

— You specify the `var=` attribute to provide the name for a Java bean

This Java bean is created by the `flow` tag and returned to you. Within your JSP you can then refer to the variable called `myFlowBean`, and call methods on it to obtain the flow specific information that is returned by the tag.

The javadoc for the `flow` tag tells you that the bean returned to you is of type `FlowBean`. And the javadoc for the `FlowBean` tells you that you have methods available such as: `getActiveCount()`, `getNodes()`, etc.

Two Types of Tags

Generally speaking, within most of the tag libraries, you find two types of tags:

1. List tags
2. Specific tags

List Tags

Some examples of “list” tags are:

- `<flow:flowOutlineList>`
- `<flow:flowInstanceOutlineListCount>`
- `<ovis:serviceObjectiveList>`

These are tags that return “lists” of items.

When calling a “list” tag, be aware that the tag actually sets up a “loop” within your JSP code.

Let’s explain by an example:

> Suppose you want to loop through all the flows currently active in your OVBPI system, displaying each flow name. You could use the following code segment:

```
<table>
   <flow:flowOutlineList var="flowOutlineBean">
     <tr>
         <td><%= flowOutlineBean.getName() %></td>
     </tr>
   </flow:flowOutlineList>
</table>
```

> The above code segment produces code that basically loops through your flow definitions, displaying the name of each flow.

So a “list” tag typically spreads over a few lines, and expands into a code-loop that loops through the returned results.

Specific Tags

A specific tag is a “non-list” tag. It returns a single object or value.

For example:

The `<flow:flow>` tag is a “non-list” tag. Suppose you have the flow ID for a particular flow definition. You can use the `<flow:flow>` tag to get the details for the given flow.

The code looks like this (assuming the flow ID is held in a Java variable called `flowId`):

```
<flow:flow flowId="<%= flowId %>" var="flowBean" />
```

where:

— This tag does not produce any loops. It simply retrieves the data for the requested flow ID and returns a bean containing the details for this flow

— The tag is executed, the `flowBean` variable is loaded up with data, and control then moves on to the next line of code in the JSP

Error Handling

There are utility tags for handling errors that may be returned from other tags. For example, `<util:onerror ...>` can be placed after a tag call to catch any error that might get thrown.

For example, consider the following JSP code segment:

```
<jsp:useBean id="errorInfoBean"
   class="com.hp.ov.bia.views.taglibs.util.ErrorInfoBean" scope="session" />

...

<flow:flowInstance flowInstanceId="<%= flowInstId %>"
                  var="flowInstanceBean" errorInfoBean="<%= errorInfoBean %>"
/>

<util:onerror errorInfoBean="<%= errorInfoBean %>">
     <jsp:forward page="../gen/alertPage.jsp">
         <jsp:param name="alert" value="It all went wrong!"/>
     </jsp:forward>
</util:onerror>

<flow:flowInstanceImage flowInstanceBean="<%= flowInstanceBean %>" />
```

where:

- The `<flow:flowInstance>` tag gets the instance details for the given flow instance ID. The result is returned in a `FlowInstanceBean`
- Because the `<flow:flowInstance>` tag has specified an `errorInfoBean=` parameter, any error information is returned to the calling JSP code
- If the `<flow:flowInstance>` tag throws an error then the code between the `<util:onerror>` mark-up is executed. As this example forwards to another JSP, it terminates this page

  A possible error might be if the value in the variable `flowInstId` variable does not match any flow instance ID within the OVBPI Engine database.
- If the `<flow:flowInstance>` tag has no errors then the `<util:onerror>` code is skipped and the JSP moves on to the next line (which in this case is the `<flow:flowInstanceImage>` tag)
- The `errorInfoBean` must be declared as a session wide bean, hence the `<jsp:useBean:` tag

The most obvious error that you can expect to see is when you ask for an object (flow, flow instance, node, node instance, etc.) that doesn't exist. For example, you ask for a specific flow but have not supplied the correct flow ID.

List Tags

Be aware that when using a "list" tag, you may not always get errors thrown. In particular, the "list" tags tend not to throw any error if no objects are found.

For example:

```
<flow:flowOutlineList var="flowOutlineBean" nameFilter="Order Flow" />
```

The `flowOutlineList` tag sets up a loop and finds all flow definitions that match any given criteria. In this case, you have asked to retrieve only flows whose name matches the given flow name: `Order Flow`.

You might expect to be able to place some `<util:onerror>` code after this call to catch the condition where the flow name was not found....wrong!

The `flowOutlineList` tag in the above example loops through the flow definitions trying to match them against the given filter. If there are no matches then that is a valid outcome!

In other words, a “list” tag does not consider “no return values” to be an error.

Java Beans

The OVBPI Dashboard provides a number of back-end Java beans for accessing the OVBPI data. Some of these work in conjunction with the taglibs, and others are there as helpers if you need to start writing bespoke SQL and issuing this against the OVBPI database.

Documentation (Javadocs)

Full documentation for all the Dashboard Java beans is provided on the OVBPI product CD - under the `docs` directory.

You can read the documentation by using a Web browser and opening the file:

```
docs\html\OVBPIJavadoc\index.html
```

You can broadly categorize the beans into two main groups:

- Taglib Beans

  ```
  com.hp.ov.bia.views.taglibs.*
  ```

  These beans are the ones you need to know about when using the taglibs.

- Helper Beans

  ```
  com.hp.ov.bia.views.*
  com.hp.ov.bia.views.util.*
  ```

  These are the additional beans that might come in handy as/when you start doing Dashboard customization that cannot be performed within a taglib. The most obvious example of this would be if you want to issue custom SQL statements directly against the OVBPI database; see Chapter 6, Direct OVBPI Database Access.

The Next Step

With the Dashboard JSP code as an example, and all the Taglib and Java bean documentation, it is pretty easy to understand how everything works.

To help your understanding, the remaining chapters of this guide lead you through many worked examples.

3

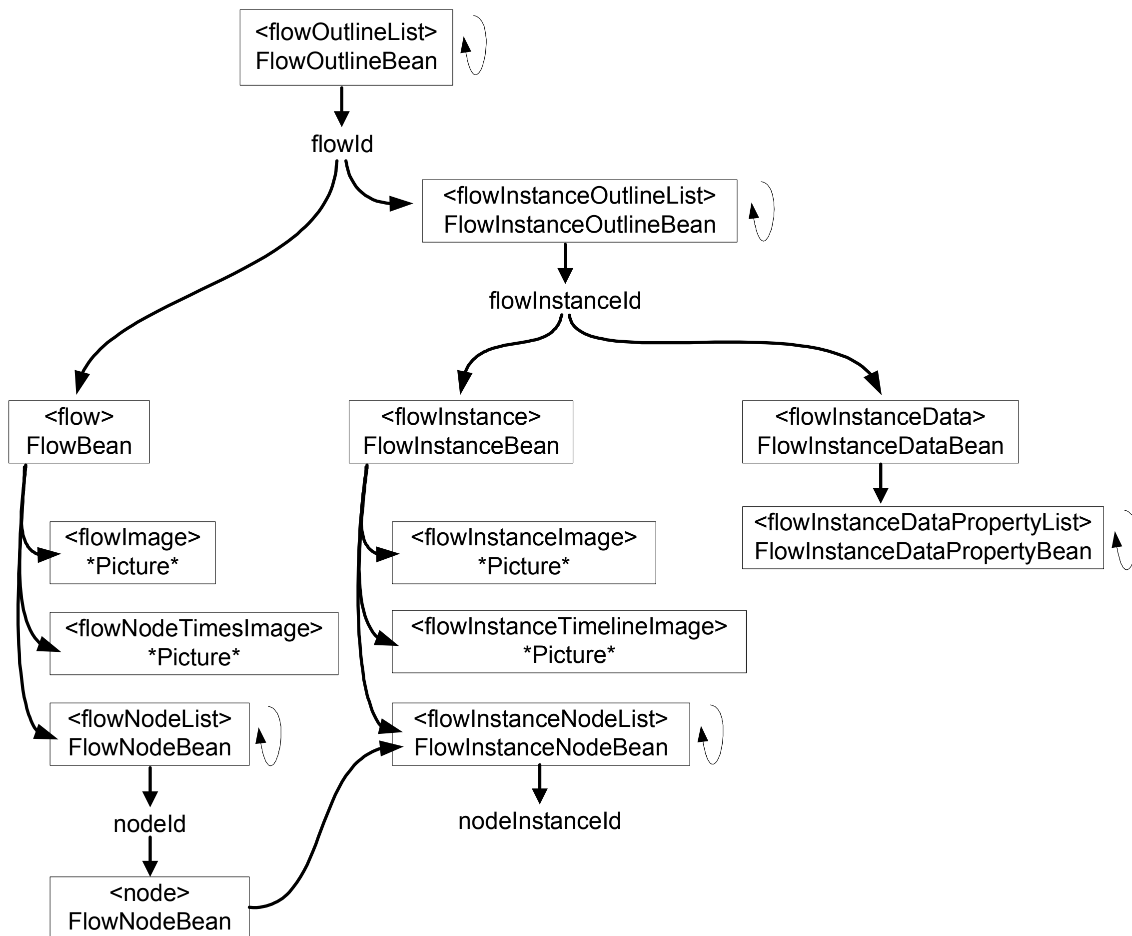
Working With Flows

This chapter looks at the <flow> tag library. It provides an overview of the typical calling sequence, and then takes you through a series of worked examples.

<flow> Tag Hierarchy

The following diagram lists the main <flow> tags, and shows how they relate together:

Figure 1 <flow> Tag Hierarchy



On the diagram:

- Each box lists the tag name, and underneath that is the type of Java bean that the tag returns. For those tags that do not return a Java bean it lists what the tag produces.

- There is a little "loop" symbol to the right of each "list" tag to indicate that this tag typically loops through the return values.

The idea of the diagram is to help you see the typical calling sequence for the tags. For example:

- You typically start with the `flowOutlineList` tag to loop through the available flows and pull out the `flowId`
- Once you have the `flowId`, you might then call the `flow` tag, passing it this `flowId`. The `flow` tag returns to you a `FlowBean`
- You can then pass this `FlowBean` to the `flowImage` tag and have it draw you a flow diagram

Alternatively, if you wanted to drill down into the flow instance details:

- Once you have the `flowId`, you can call the `flowInstanceOutlineList` tag. This allows you to loop through the flow instances for this `flowId`.

  For each flow instance you get a `FlowInstanceOutlineBean`, and from this, you can extract the `flowInstanceId`
- You can then pass this `flowInstanceId` to the `flowInstance` tag, and it returns to you a `FlowInstanceBean`
- You can then pass this `FlowInstanceBean` to the `flowInstanceImage` tag and it draws you a flow instance diagram

Accessing Flow Details

Listing Flow Definitions

To simply loop through your flow definitions and list information about each, you use the `flowOutlineList` tag.

In its simplest form, you call the tag and just specify the variable name for the return results. By default the tag returns all non-superseded flow definitions:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Simple Flow Listing</title></head>
  <body>
     <table cellSpacing=0 border=1>
        <th>Flow List</th>
        <flow:flowOutlineList var="flowOutlineBean">
           <tr>
                <td><%= flowOutlineBean.getName() %></td>
           </tr>
        </flow:flowOutlineList>
     </table>
  </body>
</html>
```

where:

- You declare the taglib:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>
```

  This tells the JSP that you want to use the Java taglib `com.hp.ov.bia.views.taglibs.flow`. The `prefix` option says that you will refer to this taglib within the rest of the JSP by the tag name `flow`.

  You can now start your flow tags with: `<flow:`*`whatever`*`>`, and end the tags with `</flow:`*`whatever`*`>`.

The above example code simply loops through, outputting the name of each flow definition. The javadocs for the return bean type (`FlowOutlineBean`) tells you that you can output lots of information about the flow using methods such as:

```
getActiveCount()
getBlockedWeight()
getAverageDuration()
getFlowId()
getItStatus()
getStatus()
```

and many more.

Filtering Flows By Name

You can specify that the `flowOutlineList` tag only retrieve flow definitions where the flow name matches a certain name, or list of names.

To match on a single flow name:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Flow List - simple filter</title></head>
  <body>
     <table cellSpacing=0 border=1>
        <th>Flow Name</th>
        <th>Active</th>
        <th>Business Health</th>
        <flow:flowOutlineList var="flowOutlineBean" nameFilter="Order Flow" >
           <tr>
                <td><%= flowOutlineBean.getName() %></td>
                <td><%= flowOutlineBean.getActiveCount() %></td>
                <td><%= flowOutlineBean.getStatus() %></td>
           </tr>
        </flow:flowOutlineList>
     </table>
  </body>
</html>
```

To specify more than one flow name, you need to make use of the `<util:args>` tag...

Multiple Flow Names (<util:args>)

To specify more than one flow name, you need to build up an `ArgsBean` which can then be passed to the `flowOutlineList` tag. You can do this by using the `<util:args>` tag:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.util" prefix="util" %>

<html>
  <head><title>Flow List - util:args filter</title></head>
  <body>

     <util:args var="flowNameArgsBean">
         <util:arg name="Order Flow"/>
         <util:arg name="Insurance Claim"/>
     </util:args>

     <table cellSpacing=0 border=1>
        <th>Flow Name</th>
        <th>Active</th>
        <th>Business Health</th>
        <flow:flowOutlineList var="flowOutlineBean"
                              nameFilter="<%= flowNameArgsBean %>" >
           <tr>
                <td><%= flowOutlineBean.getName() %></td>
                <td><%= flowOutlineBean.getActiveCount() %></td>
                <td><%= flowOutlineBean.getStatus() %></td>
           </tr>
        </flow:flowOutlineList>
     </table>
  </body>
</html>
```

Filtering By Flow Status

The `flowOutlineList` tag also lets you filter the flow list by the current state of the flow definition.

Valid states are listed in the javadoc, and they are as follows:

- `ACTIVE`

  This matches flows that are "healthy"; see Flow Status on page 15

- `IMPEDED`

  This matches flows that are "at risk"; see Flow Status on page 15

- `BLOCKED`

  This matches flows that are "blocked"; see Flow Status on page 15

- `DELETED`

  This matches flows that have been superseded by a newer version. Internally, within OVBPI, the flow is referred to as being marked for deletion (hence the status `DELETED`), but within the Dashboard the flow is referred to as a "superseded" flow.

- `NOT DELETED`

  This is the default. It matches flows that are in any state other than `DELETED`.

  Using this filter is the same as a filter of `ACTIVE|IMPEDED|BLOCKED`.

- `ALL`

  Simply matches all flows in all states.

Here is an example that lists only the superseded flow definitions:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Flow List - by flow status</title></head>
  <body>
     <table cellSpacing=0 border=1>
        <th>Flow Name</th>
        <th>Active</th>
        <th>Business Health</th>
        <flow:flowOutlineList var="flowOutlineBean"
                              nameFilter="Order Flow"
                              statusFilter="DELETED" >
           <tr>
                <td><%= flowOutlineBean.getName() %></td>
                <td><%= flowOutlineBean.getActiveCount() %></td>
                <td><%= flowOutlineBean.getStatus() %></td>
           </tr>
        </flow:flowOutlineList>
     </table>
  </body>
</html>
```

Getting the Flow ID

To convert a flow name into a flow ID, you can use the `flowOutlineList` tag.

Specify the name of the flow (`nameFilter=`) and `flowOutlineList` returns the active version of that flow definition. You can then pull out the flow ID from the returned `FlowOutlineBean`.

For example:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Flows - get the flow ID</title></head>
  <body>
        <%
            String flowId = "";
        %>

        <flow:flowOutlineList var="flowOutlineBean" nameFilter="Order Flow" />

        <%
            if (flowOutlineBean != null)
            {
                flowId = flowOutlineBean.getFlowId();
            }
        %>

        <!-- Display the result -->

        Flow ID = [<%= flowId %>]

  </body>
</html>
```

where:

- After the `flowOutlineList` tag, you check to see if `flowOutlineBean` is null.

  You do this because, if the `flowOutlineList` tag cannot find any flows that match the given criteria, it does not allocate a return bean and hence when you go to access `flowOutlineList`, it might be null.

Listing Node Instance Details

Let’s look at an example where you have found the flow ID and you now want to loop through all the flow instances for this flow ID. For each flow instance you then want to loop through any node instances.

The basic flow of tags is as follows:

- Pass the flow ID to the `flowInstanceOutlineList` tag and set up a loop to go through all the returned flow instances
- Within this loop, you can display some overall flow instance details, and then call the `flowInstance` tag to get further details about each flow instance
- You can then pass this flow instance information into the `flowInstanceNodeList` tag and set up a loop to go through each of the node instances

The JSP code segment might look something like this:

```
<%-- Loop through the flow instance details --%>

<flow:flowInstanceOutlineList var="flowInstanceOutlineBean"
                              idFilter="<%= flowId %>"
                              maxInstances="100"  >
    <%
      // Pull out the flow instance Id
      String flowInstId = flowInstanceOutlineBean.getFlowInstanceId();
    %>

    <%-- Print some basic flow instance details --%>

    <h1>Flow Instance:</h1>

    Flow InstId: <%= flowInstId %>     <br />
    Identifier : <%= flowInstanceOutlineBean.getIdentifier() %> <br />


    <%-- Now loop through and list the node instance details --%>

    <h2>Node Instances:</h2>

    <%-- Get the flow instance object --%>

    <flow:flowInstance flowInstanceId="<%= flowInstId %>"
                       var="flowInstanceBean" />

    <%-- Now get the list of node instances                --%>
    <%-- and loop through displaying the node instance ids --%>

    <flow:flowInstanceNodeList flowInstanceBean="<%= flowInstanceBean %>"
                               var="flowInstanceNodeBean" >
        Node Name    : <%= flowInstanceNodeBean.getName() %>   <br />
        Node InstId  : <%= flowInstanceNodeBean.getNodeInstanceId() %><br />
        Node Duration: <%= flowInstanceNodeBean.getOverallDuration() %><p />
    </flow:flowInstanceNodeList>

</flow:flowInstanceOutlineList>
```

Drawing Flow Diagrams

There are three main types of "flow" diagram available to you, and a corresponding taglib for each of these. They are:

- Flow Diagram: <flowImage>

  This shows the overall flow diagram, showing the number of flow instances active at each node, and any IT service impact that might be impacting the flow.

- Flow Instance Diagram: <flowInstanceImage>

  This shows where a particular flow instance currently is in the flow, showing which nodes have been completed, which are active, and whether any are currently impacted by an IT service impact.

- Flow Instance Timeline: <flowInstanceTimelineImage>

  This shows the flow instance details as a time line, so you can easily see the order in which nodes are being executed, and visually see the length of time spent in each node.

Flow Diagram <flowImage>

To draw an overall flow diagram is very simple.

The basic steps are:

- Get the flow ID
- Use the `flow` tag to get a `FlowBean` for your flow
- Pass this `FlowBean` to the `flowImage` tag

For example:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Flow Diagram Page</title></head>
  <body>
<%
    String flowName = "Order Flow";
    String flowId = "";
%>
    <%-- Get the flowId --%>

    <flow:flowOutlineList var="flowOutlineBean"
                          nameFilter="<%= flowName %>" />
<%
    // Grab the flowId
    if (flowOutlineBean == null)
    {
        String errorText = "Unable to find flow [" + flowName + "]";
        %>
           <jsp:forward page="../gen/alertPage.jsp">
              <jsp:param name="alert" value="<%= errorText %>"/>
           </jsp:forward>
        <%
    }
    flowId = flowOutlineBean.getFlowId();
%>

    <%-- Now you have the flowId, you can get the flow --%>

    <flow:flow flowId="<%= flowId %>" var="flowBean" />

    <%-- Now draw the flow --%>

    <flow:flowImage flowBean="<%= flowBean %>" />

  </body>
</html>
```

Flow Instance Diagram <flowInstanceImage>

To draw a specific flow instance image is very simple.

The basic steps are:

- Get the flow ID
- Use the `flowInstanceOutlineList` tag to return the flow instances
- Find the flow instance that you want
- Use the `flowInstance` tag to get the `FlowInstanceBean` for this instance
- Pass this `FlowInstanceBean` to the `flowInstanceImage` tag

Here is an example that gets the first (up to) 100 flow instances and displays flow instance diagrams for each one:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Flow Instance Diagram Page</title></head>
  <body>
<%
     String flowName = "Order Flow";
     String flowId = "";
%>

     <%-- Get the flowId --%>

     <flow:flowOutlineList var="flowOutlineBean"
                           nameFilter="<%= flowName %>" />
<%
         ...test that flowOutlineBean is not null...

     flowId = flowOutlineBean.getFlowId();
%>
     <%-- Now you have the flowId, you can get the list of flow instances --%>

     <flow:flowInstanceOutlineList var="flowInstanceOutlineBean"
                                   idFilter="<%= flowId %>"
                                   maxInstances="100"  >
<%
         // Pull out the flow instance Id
         String flowInstId = flowInstanceOutlineBean.getFlowInstanceId();
%>
```

```
        <%-- Print some basic flow instance details --%>

        <h1>Flow Instance:</h1>

          Identifier: <%= flowInstanceOutlineBean.getIdentifier() %> <p />

          <%-- Get the flow instance object --%>
          <flow:flowInstance flowInstanceId="<%= flowInstId %>"
                             var="flowInstanceBean" />

          <%-- Now draw the flow instance diagram --%>
        <flow:flowInstanceImage flowInstanceBean="<%= flowInstanceBean %>" />

    </flow:flowInstanceOutlineList>

  </body>
</html>
```

Flow Instance Timeline <flowInstanceTimelineImage>

To produce a flow instance timeline, you follow the same basic code steps as for a flow instance diagram (see Flow Instance Diagram <flowInstanceImage> on page 66), however, once you have the flow instance details you use the `flowInstanceTimelineImage` tag to produce the actual picture.

The basic steps are:

- Get the flow ID
- Use the `flowInstanceOutlineList` tag to return the flow instances
- Find the flow instance that you want
- Use the `flowInstance` tag to get the `FlowInstanceBean` for this instance
- Pass this `FlowInstanceBean` to the `flowInstanceTimelineImage` tag

For example, once you have the flow instance ID, you produce a flow instance timeline with the following code:

```
<%-- Get the flow instance object --%>

<flow:flowInstance flowInstanceId="<%= flowInstId %>"
                   var="flowInstanceBean" />

<%-- Now draw the flow instance timeline --%>

<flow:flowInstanceTimelineImage flowInstanceBean="<%= flowInstanceBean %>"
                                   axisWidth="500"  />
```

Setting Background Color

You can alter the background color for your flow diagrams by using the `bgColor` parameter.

If this parameter is not passed into the tag then the background defaults to the value as specified in the `DashboardConfig.properties` file. The property that sets the default background color is `FlowBackgroundColor`.

The `bgColor` parameter accepts either HTML color values (specified as hexidecimal strings preceded by the '#' character) or color names as defined within the `java.awt.Color` class.

Some examples of color names known to `java.awt.Color` are: `red`, `white`, `black`, `cyan`. Refer to the Java SDK documentation for more details.

Some examples of HTML colors are `#ff0000`, `#E4E9EE`, etc., where the hexidecimal numbers represent the shades of red, green and blue (`#rrggbb`).

Here are some examples setting the background for a flow diagram:

Named Color

```
<flow:flowImage flowBean="<%= flowBean %>" bgColor="purple" />
```

HTML (Hex) Color

```
<flow:flowImage flowBean="<%= flowBean %>" bgColor="#ddeeaa" />
```

Activating Node URLs

If you want the nodes in your flow diagrams to be hyperlinks, then you can use the `nodeBaseUrl` parameter. This is available when drawing overall flow diagrams or instance diagrams/timelines.

The parameter is called `nodeBaseURL` because it is the “base” for the resultant URL that is built for each node within the diagram. This is because the node ID might be added to the URL. Let’s explain...

Fixed URL

If you supply a simple URL string with no parameters, then that URL string is used for each node.

For example, if your tag is:

```
<flow:flowImage flowBean="<%= flowBean %>" nodeBaseUrl="myPage.html" />
```

the flow diagram is drawn and when you click on any of the nodes within the flow diagram, the Web browser redirects to the page `myPage.html`.

Dynamic URL

If your URL contains at least one HTML parameter, then the flow diagram appends the node ID to the resultant URL. This means that your URL contains the node ID of the node the user clicked on.

All that gets appended to your base URL string is the value of the node ID. So for the resultant URL to be correctly formed you should end your base URL with the name of the node ID parameter and an equals (=) sign.

For example, if you want the resultant URL to contain the node ID, you could specify the following tag:

```
<flow:flowImage flowBean="<%= flowBean %>" nodeBaseUrl="myPage.jsp?nodeid="/>
```

When the user clicks on a node within the flow diagram, the Web browser redirects to `myPage.jsp` passing in the parameter `nodeid` set to the value for the node the user clicked on. This allows `myPage.jsp` to know which node the user clicked on, and behave accordingly.

Multiple Parameters on the Base URL

If you want to pass in other parameters on the base URL, you can, but there is something that is worth knowing about.

You can construct the URL as follows:

```
<flow:flowImage flowBean="<%= flowBean %>"
        nodeBaseUrl="myPage.jsp?flowid=12345&nodeid=" />
```

and this works fine. It passes in the two parameters (`flowid` and `nodeid`) to `myPage.jsp`.

However, it is more typical within a JSP, to want to pass in parameters and set the values to Java variables. For example, you might have the flow ID in a variable called `flowId`, and you want to pass this value into the node base URL. The obvious thing to try is this:

```
<flow:flowImage flowBean="<%= flowBean %>"
        nodeBaseUrl="myPage.jsp?flowid=<%= flowId %>&nodeid=" />
```

For some reason...this does **not** work :-(It seems that JSPs are unable to understand this substitution within parameters of tags.

So...if you need to build a compound URL, you need to build the URL first, within Java, and then pass the result as one variable into the tag.

It looks like this:

```
<%
    String theURL = "myPage.jsp?flowid=" + flowId + "&nodeid=";
%>
<flow:flowImage flowBean="<%= flowBean %>" nodeBaseUrl="<%= theURL %>" />
```

Showing Metric Flags

The `flowImage` tag lets you draw a flow diagram and specify whether you want it to display metric flags on any particular nodes.

You do not need to have any actual business metrics defined for your flow. You are able to have the flow drawn with metric flags wherever you decide.

The parameters are `startMetric` and `endMetric`.

You need to specify both parameters for the diagram to show metrics.

For each parameter, you specify the name of the node on which to place the metric flag. If either one of the specified node names does not exist then none of the metric flags are shown.

Here is an example where you draw the metric flags between the `Process Order` and `Ship Order` nodes:

```
<flow:flowImage flowBean="<%= flowBean %>"
                   startMetric="Process Order"
                   endMetric="Ship Order" />
```

Further Customization

You have seen that it is easy to customize things such as background color, node URLs and metric flags. But what if you wanted to change the color of the text that shows the name of each node? Why can't you set that from within the tag? Where is the “nodeTextColor” parameter?

The reason there is no “nodeTextColor” parameter on the `flowImage` tag is because the tags need to allow people to not just change/set the text color for all nodes, but also allow people to set the text color for each node individually - if they so desired. That is, there are loads of additional customizations that you might want to make, but they are actually qualities of each node within the flow diagram and therefore not simply a “diagram wide” option.

The way that you specify node specific customizations is by using an “annotation”. (see Flow Annotations on page 73)

Flow Annotations

When calling a tag to draw a flow diagram, you can set the `annotationBean` parameter. This parameter allows you to pass in a Java bean that implements the `com.hp.ov.bia.views.taglibs.flow.FlowAnnotation` interface.

So what does that mean?

When a flow diagram is being drawn, the code drawing the flow can make call-outs to find out things like "what color to use for this node name?", or "what image to use when displaying this node?". You are able to supply a Java bean which can provide the answers to these questions. This bean that you supply is called "an annotation".

To write an annotation, your Java bean must implement the `FlowAnnotation` interface. This just means that your bean needs to provide a particular set of methods as defined in the `FlowAnnotation` interface.

So what kind of methods are defined in this `FlowAnnotation` interface? It basically defines methods such as:

```
getNodeText()
getNodeTextColor()
getArcColor()
getMetricArcColor()
getNodeTooltip()
etc...
```

So when the flow is being drawn, as the flow drawer draws each node it calls the method `getNodeText()` to get the actual text to display for this node. The default is the actual name of the node. The drawer then calls the method `getNodeTextColor()` to see what color this node's text should be. It continues through the various methods as it gathers all the necessary information it needs to draw each node.

This mechanism of calling out to an annotation means that the flow drawer is highly configurable.

DefaultFlowAnnotationBean

What if you are not a Java guru! How are you supposed to write a Java bean that implements the `FlowAnnotation` interface?

Well...you don't. There is one already written for you, and it is called:

```
com.hp.ov.bia.views.taglibs.flow.DefaultFlowAnnotationBean
```

This Java bean allows you to set things such as the color of the node text, and the color of the arcs within the diagram, etc..

Your JSP needs to instantiate this bean - within Java code...and that is not too difficult if you have an example to follow.

Example 1

To use the `DefaultFlowAnnotationBean` bean you need to import its definition at the top of the JSP.

You also need to import `java.awt.Color`. This is because the methods within `DefaultFlowAnnotationBean`, that allow you to set colors, all expect the color to be defined as a Java color.

So at the top of the JSP you have the import line:

```
<%@ page import="com.hp.ov.bia.views.taglibs.flow.DefaultFlowAnnotationBean,
                java.awt.Color"%>
```

When you are about to display your flow diagram, you need some Java code to instantiate a `DefaultFlowAnnotationBean`, and then set various colors.

You then call the `flowImage` tag, passing it your instantiation of the default annotation bean.

Your code might look something like this:

```
<%-- Assuming you have the flowId set, you can get the flow --%>

<flow:flow flowId="<%= flowId %>" var="flowBean" />

<%-- Set up the default annotation --%>
<%
    DefaultFlowAnnotationBean defAnno = new DefaultFlowAnnotationBean();

    defAnno.setArcColor(Color.pink);
    defAnno.setNodeTextColor(Color.decode("#ddeeaa"));
%>

<flow:flowImage flowBean="<%= flowBean %>"
                annotationBean="<%= defAnno %>" />
```

where this example draws the arcs of the flow in pink, and the color of the node labels is set to “#ddeeaa” - which is a nice yellow’ish-green’ish color.

Example 2

This example code segment uses the default annotation bean to set more colors. The example also sets the additional `flowImage` tag parameters to alter the background color and show metric flags:

```
<%
    DefaultFlowAnnotationBean defAnno = new DefaultFlowAnnotationBean();

    defAnno.setArcColor(Color.pink);
    defAnno.setMetricArcColor(Color.white);
    defAnno.setNodeLeftTextColor(Color.orange);
    defAnno.setNodeRightTextColor(Color.blue);
    defAnno.setNodeTextColor(Color.decode("#ddeeaa"));
%>

<flow:flowImage flowBean="<%= flowBean %>"
                bgColor="#aaaaaa"
                startMetric="Process Order"
                endMetric="Ship Order"
                annotationBean="<%= defAnno %>" />
```

Developing Your Own Annotation

To do further customization, such as:

- Specifying what the left and/or right text should be for a node
- Setting different colors for the text on different nodes
- Setting different images for each node
- etc..

you need to supply your own annotation.

As stated earlier, you could do this by writing a Java bean that implements the `FlowAnnotation` interface. However, an easier way is to write a Java bean that extends the default annotation bean: `DefaultFlowAnnotationBean`.

By extending `DefaultFlowAnnotationBean`, you only need to provide methods for the ones that you wish to alter. That is, you override the methods within `DefaultFlowAnnotationBean` that you want to change.

So if the easiest way to provide an annotation is to override methods within `DefaultFlowAnnotationBean`, what methods does this bean offer and which ones are typically worth overriding?

The full list of methods is described in the javadocs (see Documentation (Javadocs) on page 50).

Typical Methods To Override

The methods that you might typically want to override are:

- `getNodeLeftText()`
  `getNodeRightText()`
  `getNodeLeftTextColor()`
  `getNodeRightTextColor()`

  These allow you to specify what is displayed as left and right text above each node, and the individual colors. For example, you could specify different values and/or colors for each node.

- `getNodeText()`
  `getNodeTextColor()`

  These allow you to alter the text that is displayed underneath each node as its label, and the color to use. For example, you might specify different colors for each node label.

- `getNodeTooltip()`

  This allows you to specify what data/text is put into the tooltip that appears when the user moves the cursor over each node.

- `getNodeTypeImage()`

  This allows you to specify the actual image used when displaying a node. For example, you might set an image based on the type of node, or set the image differently for each node.

Method Parameters

All these methods are passed three parameters:

1. The current node (`FlowNode`)

   This is passed as a `FlowNode` object.

   The method is able to access all the node details from within this object. Details such as the node name, node type, etc.

   The method is also able to derive from this `FlowNode` object whether this node is part of an overall flow diagram or part of a flow instance diagram.

2. The current JSP page context (`PageContext`)

   This passes the current Web context to the method.

3. The current Web browser locale (`Locale`)

   This allows the method to determine things such as whether to use dollar($) signs or pound(£) sign...etc..

FlowNode Parameter

As mentioned above, the node that is being drawn, is passed to the method as a `FlowNode` object. You can think of this object definition (class) as being a general definition for any node.

When a method is passed this `FlowNode` object, the method is able to ask whether the node is an instance of a `FlowNodeBean` class or a `FlowInstanceNodeBean` class. This allows the method to know whether it is drawing an overall flow diagram or a flow instance diagram. It also allows the method to know what kind of node information the bean contains.

Hence you can write your methods to behave differently depending on whether they are drawing a flow diagram or a flow instance diagram.

Within the method you can construct your code as follows (assuming that the `FlowNode` object is passed in a variable called `node`:

```
if (node instanceof FlowNodeBean)
{
     // You are drawing a flow diagram

     FlowNodeBean fNode = (FlowNodeBean)node;

     ...your code here...referencing the fNode...
}
else
{
     // You are drawing a flow instance diagram

     FlowInstanceNodeBean fiNode = (FlowInstanceNodeBean)node;

     ...your code here...referencing fiNode...
}
```

If you are writing an annotation that you intend to use only when calling a flow diagram (`<flowImage>`), then you do not need to worry about testing the node type. You could just assume that it is a `FlowNodeBean`. Indeed, typically you write an annotation for a specific flow diagram. But at least you have the ability to write more generic/reusable annotations if you wish.

How to Write an Annotation

There are two main ways that you can write an annotation:

1. As a separate Java bean, compiled by you, and placed underneath `WEB-INF\classes` or within a JAR file in `WEB-INF\lib`
2. As code within a JSP page

As a Separate Java Bean

Let's work through an example where you create an external Java annotation bean:

Create a Subdirectory

Create a subdirectory under `WEB-INF\classes` and give this new directory the name `extra`

Create a Source File

Create your new Java source file in this `WEB-INF\classes\extra` directory and make sure that you specify the package name to be `extra`

The first line of your file should be:

```
package extra;
```

For this example, your Java source file is called: `AnnotateCountAndValue.java`

Write the Code

The Java class that you write must have the same name as the basename of the source file, so you define your Java class to be called `AnnotateCountAndValue` and have it extend `DefaultFlowAnnotationBean`

You source code looks something like this:

```
package extra;

public class AnnotateCountAndValue extends DefaultFlowAnnotationBean
{

}
```

You now write the method(s) that you want to override.

In this example, you might provide the following methods:

- `getNodeLeftText()`

  Having it return the active value (weight) in each node.

- `getNodeRightText()`

  Having it return the active flow instance count in each node.

The code might look something like this:

```
package extra;

import com.hp.ov.bia.views.taglibs.flow.DefaultFlowAnnotationBean;
import com.hp.ov.bia.views.taglibs.flow.FlowNode;
import com.hp.ov.bia.views.taglibs.flow.FlowNodeBean;
import com.hp.ov.bia.views.util.FlowNodeType;
import javax.servlet.jsp.PageContext;
import java.text.NumberFormat;
import java.util.Locale;

public class AnnotateCountAndValue extends DefaultFlowAnnotationBean
{
     public String getNodeLeftText(FlowNode node,
                                    PageContext pageContext, Locale locale)
    {
        String text = "";
        NumberFormat nf = NumberFormat.getCurrencyInstance(locale);

        FlowNodeBean fNode = (FlowNodeBean)node;
        if ( (FlowNodeType.END_NODE).equals(fNode.getType()))
        {
          text = "T = " + nf.format(fNode.getTotalWeight() / 1000000) + "M";
        }
        else
        {
            text = nf.format(fNode.getActiveWeight() / 1000000) + "M";
        }

        return text;
    }

    public String getNodeRightText(FlowNode node,
                                    PageContext pageContext, Locale locale)
    {
        String text = "";

        FlowNodeBean fNode = (FlowNodeBean)node;
        if (fNode.getType().equals(FlowNodeType.END_NODE))
        {
            text = "T = " + fNode.getTotalCount();
        }
        else
        {
            text = new Long(fNode.getActiveCount()).toString();
        }
        return text;
    }
}
```

where:

- The `getNodeLeftText()` method returns a different value depending on whether it is an end node or not

  For an end node, the method returns the total weight that has passed through this node. For all other nodes it returns the active weight at the node.

- `getNodeRightText()` returns active or total instance counts at each node, depending on whether it is an end node or not
- These methods assume that the node being drawn is part of a flow diagram. That is why you see the lines:

  ```
  FlowNodeBean fNode = (FlowNodeBean)node;
  ```

  The method assumes that it is a flow node and not a flow instance node.

Compile Your Annotation Code

To do this, you need to be in the `WEB-INF\classes\extra` directory.

You need to compile your code with the following JAR files in your classpath:

- OVBPI Jar files from the `WEB-INF\lib` directory:

  ```
  bia-views.jar
  bia-flowviewer.jar
  ```

- Tomcat libraries from the `jakarta-tomcat-5.0.19\common\lib` directory:

  ```
  jsp-api.jar
  servlet-api.jar
  ```

Your compilation looks something like this (for MS Windows):

```
set CP=..
set CP=%CP%;../../lib/bia-views.jar
set CP=%CP%;../../../../../common/lib/jsp-api.jar
set CP=%CP%;../../lib/bia-flowviewer.jar
set CP=%CP%;../../../../../common/lib/servlet-api.jar

javac -classpath %CP% AnnotateCountAndValue.java
```

Use the Annotation from a JSP

You can now call this annotation from a JSP when displaying a flow diagram

```
    <h1>Flow Diagram - calling external annotation</h1>
<%
    extra.AnnotateCountAndValue myAnno = new extra.AnnotateCountAndValue();
%>
    <flow:flowImage flowBean="<%= flowBean %>"
                    annotationBean="<%= myAnno %>" />
```

Issues With This Technique

This is probably the most “standard Java” way to write an annotation. However, be aware that once you run your JSP, and it successfully loads and uses your annotation, if you make any further modifications to your annotation (and recompile your Java code) the Servlet Engine (Tomcat) does **not** reload the annotation code! It keeps using the currently loaded version of your annotation.

You need to restart the Servlet Engine (Tomcat) to force it to reload the new version of the annotation.

This behavior is OK for a production system, but can be frustrating when you are in development mode and trying to fine-tune your annotation code.

Also, writing your annotation this way means that you have to manually recompile the annotation whenever you make changes.

There is a “slightly easier” alternative that you might wish to consider when developing annotations...

An Annotation Within a JSP

When writing your annotation code within a JSP, you write the same basic code, however, there are two things that make this a good way to work:

- Your annotation code is automatically compiled when the Web browser needs to run it.
- If you make any changes, the annotation is recompiled and the new version is automatically picked up by the Web browser.

Code Structure

You can write the annotation directly in the calling JSP or in a separate JSP of its own. It's a good idea to write the annotation in a separate JSP as this allows you to reuse the annotation amongst many JSPs - you simply need to include the file in the other JSPs and they are then able to reference the annotation.

The typical structure of the annotation JSP is as follows:

```
<%@ page import=" ...import the Java classes needed for your annotation... "
%>

<%!
public class AnnoLeftRightText extends DefaultFlowAnnotationBean
{

   ...your annotation methods in here

}
%>
```

where:

- At the top of the annotation JSP file you have a line to import all the necessary Java classes.
- The actual class definition is then placed between the tags `<%!` and `%>`.

  These mark the code as global/reusable, hence you are able to define your class.

So an annotation that provided methods to set the left and right text color looks something like this:

```
<%@ page import="com.hp.ov.bia.views.taglibs.flow.DefaultFlowAnnotationBean,
                 com.hp.ov.bia.views.taglibs.flow.FlowNode,
                 com.hp.ov.bia.views.taglibs.flow.FlowNodeBean,
                 com.hp.ov.bia.views.util.FlowNodeType,
                 java.util.Locale,
                 java.text.NumberFormat"
%>

<%!

public class AnnoNodeLeftRightText extends DefaultFlowAnnotationBean
{
    public String getNodeLeftText(FlowNode node,
                                PageContext pageContext, Locale locale)
    {
        String text = "";

        NumberFormat nf = NumberFormat.getCurrencyInstance(locale);

        FlowNodeBean fNode = (FlowNodeBean)node;
        if ( (FlowNodeType.END_NODE).equals(fNode.getType()))
        {
           text = "T=" + nf.format(fNode.getTotalWeight() / 1000000) + "M";
        }
        else
        {
            text = nf.format(fNode.getActiveWeight() / 1000000) + "M";
        }

        return text;
    }

    public String getNodeRightText(FlowNode node,
                                  PageContext pageContext, Locale locale)
    {
         ...your code in here...
    }
}
%>
```

where:

- The `getNodeLeftText()` method returns a different value depending on whether it is an end node or not.

  For an end node, the method returns the total weight that has passed through this node. For all other nodes it returns the active weight at each node.

- `getNodeRightText()` does whatever you require it to do.
- These methods assume that the node being drawn is part of a flow diagram. That is why you see the lines:

  `FlowNodeBean fNode = (FlowNodeBean)node;`

  The method assumes that it is a flow node and not a flow instance node.

Use the Annotation from a JSP

You can now include the JSP, that defines the annotation, within a JSP that wants to draw a flow using the annotation, as follows:

```
<%@ include file="ex_anno_nodeLeftRightText_flowOnly.jsp" %>

<h1>Flow Diagram - calling JSP annotation</h1>
<%
 AnnoNodeLeftRightTextFlowOnly myAnno = new AnnoNodeLeftRightTextFlowOnly();
%>
 <flow:flowImage flowBean="<%= flowBean %>"
                 annotationBean="<%= myAnno %>" />
```

where:

- You use the `<%@ include` markup to include the JSP that contains the code for the annotation.
- You then instantiate one of these annotations.
- You then pass this annotation into the `<flow:flowImage>` tag.

Setting Left Text

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeLeftText()` you are able to provide the code that is called as the flow drawer draws each node. If you return a text string, this text is placed above-and-to-the-left of that node within the flow diagram.

In the following example:

- The `weight` field of the flow is assumed to be a currency value
- The method behaves differently depending on whether it is drawing a flow diagram or a flow instance diagram

  For a flow diagram, it returns:

  — The total weight that has been through the node - if the node is an end node

  — The current weight at a node - for all other node types

  For a flow instance diagram it simply returns whatever the default is.
- The weight value is formatted as currency, using the local settings, and the value is shown in millions - hence it is divided by 1000000 and displayed with the letter “M” appended.

Here is the code:

```
public String getNodeLeftText(FlowNode node,
                                   PageContext pageContext, Locale locale)
{
    String text = "";

    NumberFormat nf = NumberFormat.getCurrencyInstance(locale);

    if (node instanceof FlowNodeBean)
    {
        // It is a flow diagram

        FlowNodeBean fNode = (FlowNodeBean)node;
        if ( (FlowNodeType.END_NODE).equals(fNode.getType()))
        {
            text = "T=" + nf.format(fNode.getTotalWeight() / 1000000) + "M";
        }
        else
        {
            text = nf.format(fNode.getActiveWeight() / 1000000) + "M";
        }
    }
    else
    {
        // It is for a flow instance

        // Just set it to the default
        text = super.getNodeLeftText(node, pageContext, locale);
    }

    return text;
}
```

Setting Right Text

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeRightText()` you are able to provide the code that is called as the flow drawer draws each node. If you return a text string, this text is placed above-and-to-the-right of that node within the flow diagram.

In the following example:

- The method behaves differently depending on whether it is drawing a flow diagram or a flow instance diagram

  For a flow diagram, it returns:

  — The total flow instance count that has been through the node - if the node is an end node

  — The current flow instance count at a node - for all other node types

  For a flow instance diagram it simply returns whatever the default is.

Here is the code:

```
public String getNodeRightText(FlowNode node,
                                    PageContext pageContext, Locale locale)
{
    String text = "";

    if (node instanceof FlowNodeBean)
    {
        // It is for a flow diagram

        FlowNodeBean fNode = (FlowNodeBean)node;
        if (fNode.getType().equals(FlowNodeType.END_NODE))
        {
            text = "T=" + fNode.getTotalCount();
        }
        else
        {
            text = new Long(fNode.getActiveCount()).toString();
        }
    }
    else
    {
        // It is for a flow instance

        // Just set it to the default
        text = super.getNodeRightText(node, pageContext, locale);
    }

    return text;
}
```

Setting Left Text Color

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeLeftTextColor()` you are able to provide the code that is called as the flow drawer draws each node. If you return a color, this color is used to display the left node text.

In the following example:

- You return a different color depending on the name of the node
  - — If the node being displayed has the name `Process Order`, then you return a color of green
  - — If the node being displayed has the name `End`, then you return a color of white
  - — For all other nodes, you return the default color
- The color that you return must be of type `java.awt.Color`

  `java.awt.Color` lets you chose from a few “known” color names, or you can specify the color by setting the RGB values (in Hex)

Here is the code:

```
public Color getNodeLeftTextColor(FlowNode node,
                                     PageContext pageContext, Locale locale)
{
    // Default the color to what it would normally be

    Color theColor = super.getNodeLeftTextColor(node, pageContext, locale);

    // Here is specific color handling based on the name of the node

    if (node.getName().equals("Process Order"))
    {
        // If it is the process order node - make it green
        theColor = Color.green;
    }
    else if (node.getName().equals("End"))
    {
        // If it is the End node - make it white
        theColor = Color.decode("#ffffff");
    }
    return theColor;
}
```

Setting Right Text Color

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeRightTextColor()` you are able to provide the code that is called as the flow drawer draws each node. If you return a color, this color is used to display the right node text.

In the following example:

- The method returns a different color depending on the name of the node
  - If the node being displayed has the name `New Order`, then it returns a color of pink
  - If the node being displayed has the name `Process Order`, then it returns a color of yellow
  - For all other nodes, it return the default color
- The color that you return must be of type `java.awt.Color`

  `java.awt.Color` lets you chose from a few “known” color names, or you can specify the color by setting the RGB values (in Hex)

Here is the code:

```
public Color getNodeRightTextColor(FlowNode node,
                                      PageContext pageContext, Locale locale)
{
    // Default the color to what it would normally be

    Color theColor = super.getNodeRightTextColor(node, pageContext, locale);

    // Here is specific color handling based on the name of the node

    if (node.getName().equals("New Order"))
    {
        // If it is the new order node - make it pink
        theColor = Color.pink;
    }
    else if (node.getName().equals("Process Order"))
    {
        // If it is the process order node - make it yellow
        theColor = Color.yellow;
    }
    return theColor;
}
```

Setting Node Label Text

Your annotation must extend DefaultFlowAnnotationBean.

By overriding the method getNodeText() you are able to provide the code that is called as the flow drawer draws each node. If you return a text string, this text is placed below the node, as its label (or “node text”) within the flow diagram.

In the following example:

- If the name of the node is Process Order, the method returns the text The Order is Processed! For all other nodes, it simply returns the default text (which happens to be the name of the node).

Here is the code:

```
public String getNodeText(FlowNode node,
                              PageContext pageContext, Locale locale)
{
    // If the node has a certain name, you provide substitute text...else you
    // return the default text.

    if (node.getName().equals("Process Order"))
    {
        return "The Order is Processed!";
    }
    else
    {
        return super.getNodeText(node, pageContext, locale);
    }
}
```

Setting Node Label Text Color

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeTextColor()` you are able to provide the code that is called as the flow drawer draws each node. If you return a color, this color is used to display the label text for the node.

In the following example:

- You return a different color depending on the name of the node
  - If the node being displayed has the name `Process Order`, then you return a color of orange
  - For all other nodes, you return the default color
- The color that you return must be of type `java.awt.Color`

  `java.awt.Color` lets you chose from a few “known” color names, or you can specify the color by setting the RGB values (in Hex)

Here is the code:

```
public Color getNodeTextColor(FlowNode node,
                                 PageContext pageContext, Locale locale)
{
    // If the node has a certain name, you provide a substitute color...
    // else you return the default color.

    if (node.getName().equals("Process Order"))
    {
        return Color.ORANGE;
    }
    else
    {
        return super.getNodeTextColor(node, pageContext, locale);
    }
}
```

Setting Node Tooltip

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeTooltip()` you are able to provide the code that is called as the flow drawer draws each node. If you return a text string, this text becomes the tooltip that appears when the user moves their cursor over that node within the flow diagram.

In the following example:

- If the method is called while drawing a flow instance, it simply returns the default tooltip
- If the method is called while drawing a flow diagram then it builds a tooltip that contains:
  - The type of the node
  - The name of the node
  - The total flow instances that have been through this node
  - If the node is not an End node, it adds the number of active flow instances currently at this node
- Notice that each line of the tooltip is separated by a “\n” character

Here is the code:

```
public String getNodeTooltip(FlowNode node,
                                PageContext pageContext, Locale locale)
{
    // Set the tooltip to include the standard Node type and Node name.

    String tooltip = "Type: " +
                      getLocalizedNodeType(node.getType(),pageContext,locale)
                      + "\n" +
                     "Name: " + node.getName();

    if (node instanceof FlowNodeBean)
    {
        FlowNodeBean fNode = (FlowNodeBean)node;
        if (! FlowNodeType.END_NODE.equals(fNode.getType()))
        {
            // Only add the active count for a non-End node

            tooltip += "\nActive Count: " +
                          getNumberFormat(locale).format(fNode.getActiveCount());
        }
        tooltip += "\nTotal Count: " +
                      getNumberFormat(locale).format(fNode.getTotalCount());
    }
    else
    {
        // It's a flow instance - just leave it as the default tooltip

        tooltip = super.getNodeTooltip(node, pageContext, locale);
    }

    return tooltip;
}
```

Setting Node Image

Your annotation must extend `DefaultFlowAnnotationBean`.

By overriding the method `getNodeTypeImage()` you are able to provide the code that is called as the flow drawer draws each node. If you return an image, this image is used to draw that node within the flow diagram.

The image is automatically scaled to fit the size of the standard node image.

In the following example:

- If you are drawing a flow instance image, you just return the default image for the node.
- If drawing a flow diagram:
  - If the node is called `New Order`, you return the image `phone_receiver.gif`
  - If the node is called `Process Order`, you return the image `engineer.gif`
  - etc. for the different node names.
- You could obviously decide on your images based on something other than just the node name.
- This method must return an actual `java.awt.Image` object and this is achieved by calling the method: `PictureGenerator.getImage()`, passing in the name of the image to be returned.

  The full name for the `PictureGenerator` class is: `com.hp.ov.bia.common.picturegenerator.PictureGenerator`
- The image name that you return must be available to the `PictureGenerator` class.

  This means that the image file you return must be on the classpath of your calling JSP. To achieve this, place your image in the `WEB-INF\classes` directory.

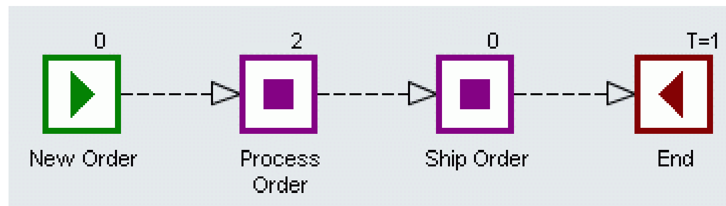
Here is the code:

```
protected Image getNodeTypeImage(FlowNode node,
                                   PageContext pageContext, Locale locale)
{
    // Initialise to the default image

    Image theImage = super.getNodeTypeImage(node, pageContext, locale);

    if (node instanceof FlowNodeBean)
    {
       if (node.getName().equals("New Order"))
       {
           ImageIcon image = PictureGenerator.getImage("phone_receiver.gif");
           theImage = image.getImage();
       }
       else if (node.getName().equals("Process Order"))
       {
           ImageIcon image = PictureGenerator.getImage("engineer.gif");
           theImage = image.getImage();
       }
       else if (node.getName().equals("Ship Order"))
       {
           ImageIcon image = PictureGenerator.getImage("ship_order.gif");
           theImage = image.getImage();
       }
       else if (node.getName().equals("End"))
       {
           ImageIcon image = PictureGenerator.getImage("success.gif");
           theImage = image.getImage();
       }
    }

    return theImage;
}
```

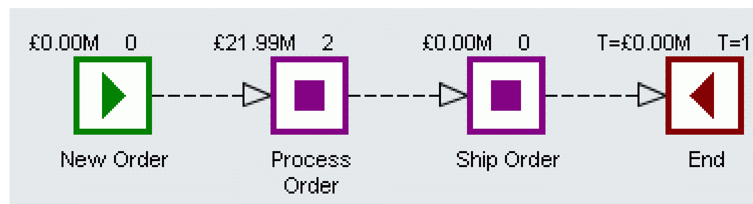
Example Flow Diagrams

To give you an idea of the way annotations can be used to alter a flow diagram, here are some examples for a flow called `Order Flow`:

Default Flow Diagram



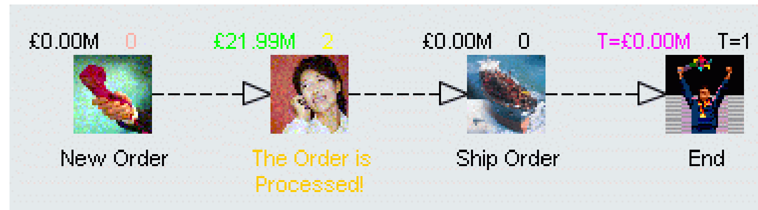
With Left/Right Text



where:

- The annotation sets:
  - — The left text to be the amount of orders currently in each node - except for the end node which shows the total value of orders that have been processed
  - — The right text shows the current number of orders active in each node - except for the end node which lists the total number of orders that have been processed. (This is the same as for the default flow diagram.)

With Different Images



where:

- The node label text for the `Process Order` node has been changed to show the text: `The Order is Processed!`
- The images for each node have been changed
- The left and right node text are displayed using individual colors

The OVBPI Dashboard

Now that you know how to draw flow diagrams with annotations, let's look at how easy it is to customize the flow diagrams within the OVBPI Dashboard.

Custom Flow Drawing

The Dashboard allows you to provide custom JSPs for drawing flow diagrams.

This means that you can write a standalone JSP that draws the flow diagram, using your own custom annotation, and the Dashboard is able to embed this within the standard Dashboard Web pages.

The Dashboard looks in the directory:

`ovbpidashboard2-0\customFlowImageDrawer`

for all custom flow drawing JSPs. If you are adding your own custom flow drawing JSPs then you need to create the `customFlowImageDrawer` directory.

Flow Drawing

To provide a custom flow drawer JSP for a flow diagram, the file must be named:

`flowname``.jsp`

where *`flowname`* must match the name of the flow to be drawn. (It is case-sensitive.)

The *`flowname`*`.jsp` page is passed the following parameters:

- `flowid`

  The flow ID of the flow to be drawn.

- `nodebaseurl` (optional)

  The base URL for each node within the flow diagram,

- `metricstartnode` (optional)

  The name of the node from which the metric starts.

- `metricendnode` (optional)

  The name of the node at which the metric ends.

- `callingpagename` (optional)

  The name of the page that invoked this JSP.

You are able to write a standalone JSP that uses some, or all, of these parameters to locate the flow, and then produce the flow diagram.

Flow Instance Drawing

To provide a custom flow drawer JSP for a flow instance diagram, the file must be named:

`*flowname*-instance.jsp`

where `*flowname*` must match the name of the flow to be drawn. (It is case-sensitive.)

The `*flowname*-instance.jsp` page is passed the following parameters:

- `flowinstanceid`

  The flow instance ID of the flow instance to be drawn.

- `view`

  The view to be drawn. Possible values are: `timeline` or `flowDiagram`.

- `nodebaseurl` (optional)

  The base URL for each node within the flow diagram,

- `axiswidth` (optional)

  The width (in pixels) of the timeline drawing.

- `callingpagename` (optional)

  The name of the page that invoked this JSP.

You are able to write a standalone JSP that uses some, or all, of these parameters to locate the flow instance and then produce the flow instance diagram or timeline.

Example - Flow Diagram

If your flow is called `Order Flow`, then creating the following and saving it in the file:

`ovbpidashboard2-0\customFlowImageDrawer\Order Flow.jsp`

provides a custom flow diagram for the Order Flow when displayed within the OVBPI Dashboard:

```
<%@ page errorPage="/gen/errorPage.jsp" %>

<%
  // Get parameters

  String flowId      = request.getParameter("flowid");
  String nodeBaseUrl = request.getParameter("nodebaseurl");
%>

<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

  <!-- Get the flow bean for this flow ID -->

  <flow:flow flowId="<%= flowId %>" var="flowBean" />

  <!-- Now draw the customized flow image -->

<%@ include file="ex_anno_theLot.jsp" %>
<%
  AnnotateTheLot myAnno = new AnnotateTheLot();
%>
  <flow:flowImage flowBean="<%= flowBean %>"
                    annotationBean="<%= myAnno %>"
                       nodeBaseUrl="<%= nodeBaseUrl %>" />
```

Example - Flow Instance Diagram

If your flow is called `Order Flow`, then creating the following and saving it in the file:

`ovbpidashboard2-0\customFlowImageDrawer\Order Flow-instance.jsp`

provides a custom flow instance diagram for the Order Flow when displayed within the OVBPI Dashboard:

```
<%@ page errorPage="/gen/errorPage.jsp" %>
<%
  // Get parameters

  String flowInstId       = request.getParameter("flowinstanceid");

  String view             = request.getParameter("view");
  String nodeBaseUrl      = request.getParameter("nodebaseurl");
  String axisWidthText    = request.getParameter("axiswidth");
  Integer axisWidth       = (axisWidthText != null && axisWidthText != "" ?
                              Integer.valueOf(axisWidthText) :
                                new Integer(500));  // Default 500 pixels.
%>

<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

  <!-- Get the flow instance bean for this flow instance ID -->

  <flow:flowInstance flowInstanceId="<%= flowInstId %>"
                                        var="flowInstanceBean" />

  <!-- Now draw the customized flow instance image -->

<%@ include file="ex_anno_theLot.jsp" %>

<%
    AnnotateTheLot myAnno = new AnnotateTheLot();

    if (view != null && view.equalsIgnoreCase("timeline"))
    {
%>
      <!-- Draw the flow instance timeline image -->
      <flow:flowInstanceTimelineImage flowInstanceBean="${flowInstanceBean}"
                                      nodeBaseUrl="<%= nodeBaseUrl %>"
                                      axisWidth="<%= axisWidth %>"
                                      annotationBean="<%= myAnno %>" />
<%
    }
    else
```

```
    {
%>
      <!-- Draw the flow instance image -->
      <flow:flowInstanceImage flowInstanceBean="${flowInstanceBean}"
                              nodeBaseUrl="<%= nodeBaseUrl %>"
                              annotationBean="<%= myAnno %>" />
<%
    }
%>
```

Lab - Drawing Flows

The purpose of this lab is to give you some practise drawing flow diagram and to help you write annotations.

Basic Flow Settings

- Create a new (empty) JSP file
- Write the code to display a flow diagram for the flow: `Order Flow` (that you developed during the *OVBPI Integration Training Guide - Modeling Flows*)

  Display the `Order Flow` using the `flowImage` tag, letting everything default. (Refer to Figure 1 on page 54 for help on the correct sequence of tags to call.)
- Now change your JSP so that the flow displays with a “purple” background
- Now show Metric flags between the nodes `Process Order` and `Ship Order`
- Change the color of the arcs (that join the nodes) to be “red”
- Change the color of the right-node-text to be “blue”

Custom Left/Right Node Text

- In the `labs` directory, locate the file: `myAnnotation.jsp`
- Your mission is to read the comments for the methods inside, and write the code. This should give your flow diagram left and right node text

  Good luck :-)

Custom Colors

- Add a new method to your myAnnotation.jsp, such that you set different colors for the left-node-text for each of the nodes. Try something like:
  - New Order - yellow
  - Process Order - blue
  - Ship Order - pink
  - End - orange
- Add the necessary methods to your myAnnotation.jsp, such that the node label for the End node says: The End at Last!!! and that this text is shown in red.
- In the labs directory, locate the file: phone_receiver.gif

  Add the necessary method(s) to your myAnnotation.jsp, such that the New Order node displays this phone_receiver.gif image instead of the standard start node image.

OVBPI Dashboard

Now that you have a standalone JSP that displays the Order Flow flow, let's use it within the OVBPI Dashboard:

- Locate the directory: webapps\ovbpidashboard2-0
- Create a subdirectory called: customFlowImageDrawer
- Now copy your standalone JSP page (that draws the Order Flow) into this webapps\ovbpidashboard2-0\customFlowImageDrawer directory.
- Rename your JSP page to be: Order Flow.jsp
- Edit this Order Flow.jsp page as follows:
  - Remove any html heading tags such as: <html>,<head>,<body> and the closing tags at the end of the page.
  - Make sure that any reference to any external files (such as the annotation) are now correct, given the new location of this file.
- Run the OVBPI Dashboard and drill into the Order Flow flow. When you display the Business Flow & Resource Summary page you should see your custom Order Flow flow diagram.

The standalone JSP that you have written to display the `Order Flow` flow is hard coded to display that flow...which is fine. You could alter your page now to use the `flowid`, and other parameters, passed into the page from the Dashboard. But as you have seen, you don't even need to do that.

- In the Dashboard, go back to the home page and drill into another flow.

  Notice that all other flows are displayed using the default flow drawer built into the Dashboard. By providing your `Order Flow.jsp` file you have provided a custom flow image drawer just for the `Order Flow` flow.

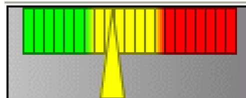
Well done! You have reached the end of the lab.

4

Working With Sliders

The Dashboard provided with OVBPI is a generic set of JSP pages that display flow information within a Web browser. The way the various pieces of data are rendered on each screen is completely customizable...assuming you have the JSP and HTML skills to do so.

To assist with dashboard customization, there is an additional Java bean that allows you to display a value as a “slider”. For example:



You can configure the colors, the range of each color, and (obviously) set the actual value to be displayed.

The “slider” is a simple bean offered to assist with making your dashboards a bit more interesting. It is just an example to help you with your dashboard customization. A full-time Web designer may well have their preferred way of displaying results graphically - that’s fine. But if you don’t have access to any drawing utilities, the simple “slider” might be of use.

Let’s look at some examples.

The Slider Bean

The slider bean is:

```
com.hp.ov.bia.views.SliderPictureGenerator
```

This is a Java bean that can draw sliders.

These sliders are created as pictures (`.png` files), which you can then display within your `HTML` as you desire.

Unfortunately, there are no taglibs for using this slider bean, you must do it all yourself in Java code. Actually, this does involve a bit of setup code, but once you have that in place it is pretty easy to use.

Javadocs

The full javadoc for the `SliderPictureGenerator` bean can be found in *`OVBPI-CD`*`\docs\html\OVBPIJavadoc\index.html`.

Let's now look at a worked example of using a slider...

Setting Up The Slider

To initialize the slider, you need to:

- Set up variables that point to the directory in which you want the slider to save its .png picture files.
- Set the size of the slider (width/height).
- Set up the range of the slider - Overall maximum, and the two mid points.
- Set up the colors for each sub-range (Default is green/yellow/red).
- Set up some variables that are used to build each picture.

This set-up code is pretty much "generic" code that is the same for all dashboards.

The set-up code looks like this:

```
<%
   // Set up a slider

   // Locate the directory where the JSP is running
   String myServletPath = request.getServletPath();
   File myJspDir = new File(application.getRealPath(myServletPath));
   myJspDir = new File(myJspDir.getParent());

   // Set up the variables for the slider
   String sliderUniquePrefix = session.getId();
   String sliderRelPath = "../images/generated";

   // Open the images directory
   File sliderDir = new File(myJspDir, sliderRelPath);

   // Set the absolute path
   String sliderAbsPath = sliderDir.getAbsolutePath();

   // Set up a slider picture generator (Width, Height)
   SliderPictureGenerator gSlider = new SliderPictureGenerator(120, 45);
   gSlider.setRange(0, 33, 66, 100);

   // You now have a slider ready for generating pictures

   // These variables are used to build each picture
   String sliderFNameBase = sliderUniquePrefix + "-slider-";
   File   sliderFile;      // Points to the picture file
   String sliderFName;     // Holds the file name for this picture
%>
```

This set-up code gives you the following variables:

- `gSlider` - This is used to generate the slider pictures
- `sliderRelPath` - The relative path to where the pictures is saved
- `sliderAbsPath` - The absolute path to where the pictures is saved
- `sliderFile` and `sliderFName` - Used to build each picture file
- `sliderFNameBase` - The base name to be used when you create each picture file name. This prefixes every picture file with the session ID of the Web Browser so that the pictures do not conflict with other Web users.

Notice that in this example, the slider ranges are `0`, `33`, `66`, `100`. This suits showing a percentage value, or values where anything over `100` is simply considered too high. If you display a value greater than the maximum, the slider simply shows it at the very top (right) of the scale - a bit like a speedometer in a car.

Producing a Slider Picture

Now that the slider is set up and ready to go, you can use it to generate pictures. You can display any value you like - an actual number, an average, a percentage - whatever you have calculated and wish to show as a slider.

Here is an example where the `flowOutlineList` tag is used to get the details for a flow. This example shows the active flow instance count within a slider. When the slider was set up earlier, the maximum was set to `100`, so if the active instance value is greater than `100` then it simply shows as being at the maximum of your slider.

This code pulls out the active flow instance count from the `flowOutlineBean`, and uses the previously defined slider picture generator (`gSlider`) to generate a picture:

```
<%
    // Build a picture for the Active instance Count

    long activeCount = flowOutlineBean.getActiveCount();

    // Set the slider to this value
    gSlider.setValue(activeCount);

    // Prepare a picture
    gSlider.preparePicture();

   // Build a unique file name for your picture
    sliderFName = sliderFNameBase + System.currentTimeMillis() + ".png";

    // Save the picture to this file
    sliderFile = new File(sliderAbsPath + "/" + sliderFName);
    sliderFile.mkdirs();
    gSlider.savePicture(sliderFile);

    // Assign a variable to hold the necessary HTML to display this picture
    String sliderImg_activeCount = "<img src=\"" + sliderRelPath + "/" +
                                    sliderFName + "\" width=\"" +
                                    gSlider.getWidth() +
                                    "\" height=\"" + gSlider.getHeight() +
                                    "\" border=\"1\" >";
%>
```

You now have your picture, and the variable `sliderImg_activeCount` holds the necessary HTML to display it.

Displaying a Slider

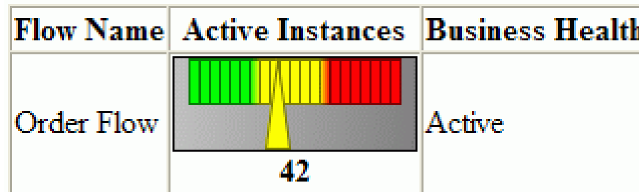
With the HTML for displaying the slider picture held in a Java variable (`sliderImg_activeCount`) you simply need to display this variable on the page.

In this code example, the slider is displayed as a column within a table. The actual value used within the slider (the active instance count) is also displayed underneath the slider:

```
<%@ page import="java.io.File,
                 com.hp.ov.bia.views.SliderPictureGenerator"%>

<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Slider - basic page</title></head>
  <body>
<%
      ...the code to set up the slider...
%>
  <table cellSpacing=0 border=1>
    <th>Flow Name</th>
    <th>Active Instances</th>
    <th>Business Health</th>

    <flow:flowOutlineList var="flowOutlineBean" nameFilter="Order Flow" >

        <tr>
           <td><%= flowOutlineBean.getName() %></td>

            <%
                 ...code to generate the picture...
                 ...and set up sliderImg with the HTML...
            %>

           <td align=center>
                         <%= sliderImg_activeCount %><br />
                         <b><%= activeCount %></b>
           </td>
        <td><%= flowOutlineBean.getStatus() %></td>
      </tr>

    </flow:flowOutlineList>

  </table>

  </body>
</html>
```

Resulting HTML Page

For this example JSP, the result might look as follows:



Flow Name	Active Instances	Business Health
Order Flow	42	Active

So the slider might be of interest as an alternative way of displaying data within your dashboard.

Setting Range Colors

The slider is designed to show three ranges - denoted by colors. The default colors being green, yellow and red. You can set these to be whatever colors you require. Indeed, you may decide that for your needs you want all three colors to be the same.

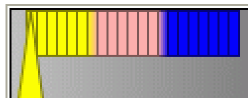
The method to set the range colors is: `setRangeColor()`

It takes three parameters, specifying the left, middle and right colors. Each color being a `java.awt.Color`.

So, for your slider generator called `gSlider`, you could set colors as follows:

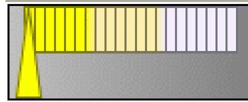
- To set the colors to be yellow/pink/blue:

```
gSlider.setRangeColor(java.awt.Color.yellow,
                      java.awt.Color.pink,
                      java.awt.Color.blue);
```



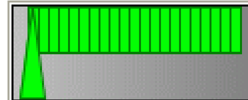
- You can also use the `decode()` method offered by `java.awt.Color`, to choose HTML colors using hexidecimal values. For example:

```
gSlider.setRangeColor(java.awt.Color.yellow,
                      java.awt.Color.decode("#fdeeaa"),
                      java.awt.Color.decode("#f0edff"));
```



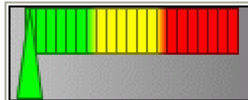
- To set the slider to have just the one color (for example: green):

```
gSlider.setRangeColor(java.awt.Color.green,
                      java.awt.Color.green,
                      java.awt.Color.green);
```



- To set the slider colors back to the default:

```
gSlider.setRangeColor(java.awt.Color.green,
                      java.awt.Color.yellow,
                      java.awt.Color.red);
```



Multiple Sliders on a Page

Once you have a slider generator defined within your JSP you can use it to produce all your slider pictures. You can also alter things such as the slider picture size, colors, ranges, etc..

Java Variables

Any Java variables you declare within a “list” tag (for example, the `flowOutlineList` tag) is only known within that “loop”.

If you want variables to be known within the loop and outside the loop, then declare them before the tag.

Unique File Names

Your Web server may be able to process the Web page fast enough that it can create more than one slider picture within one millisecond. So creating file names and using the time in milliseconds may not be enough to create a unique file name. You may wish to create the file name something like this:

```
int  sldCount = 1;

sliderFName = sliderFNameBase + System.currentTimeMillis()
                                              + "_" + sldCount++ + ".png";
```

where each file name created is unique even if the millisecond time is the same.

Adding a URL

You might want to make the slider “active” such that a user can click on the slider image and be taken to another Web page.

To do this, you simply need to alter the JSP such that the image is displayed within an HTML tag specifying the necessary HTML href.

For example:

```
<td align=center>
    <a href=show_more_details.jsp?count=<%= activeCount %> >
                                   <%= sliderImg_activeCount %></a><br />
    <b><%= activeCount %></b>
</td>
```

When the user clicks on the slider image they are taken to the JSP `show_more_details.jsp`, passing in the `count` parameter.

Lab - Drawing Sliders

The purpose of this lab is to get you using the slider bean, drawing sliders on your dashboard.

Basic Flow Definition List

- Create yourself a new JSP
- Now write the code to loop through all the OVBPI flows on your system, displaying an HTML table containing the following information for each flow:
  - Flow Name
  - Flow State
  - Active Count
  - Total Count

Adding a Slider

- Alter your JSP to display the `Total Count` as a slider, where:
  - The ranges of the slider are: `0`, `50`, `100`, `400`
  - The range colors are: `White`, `Blue`, `Pink`

Linking the Slider with a URL

- Alter your JSP so that if the user clicks on a slider, they are taken to the `gen/instances.jsp` page where they can see all the flow instances for that particular flow

Well done! You have reached the end of the lab.

5

Working With Metrics

This chapter looks at the <metrics> tag library.

Overall, the tag library can be divided into five main groups:

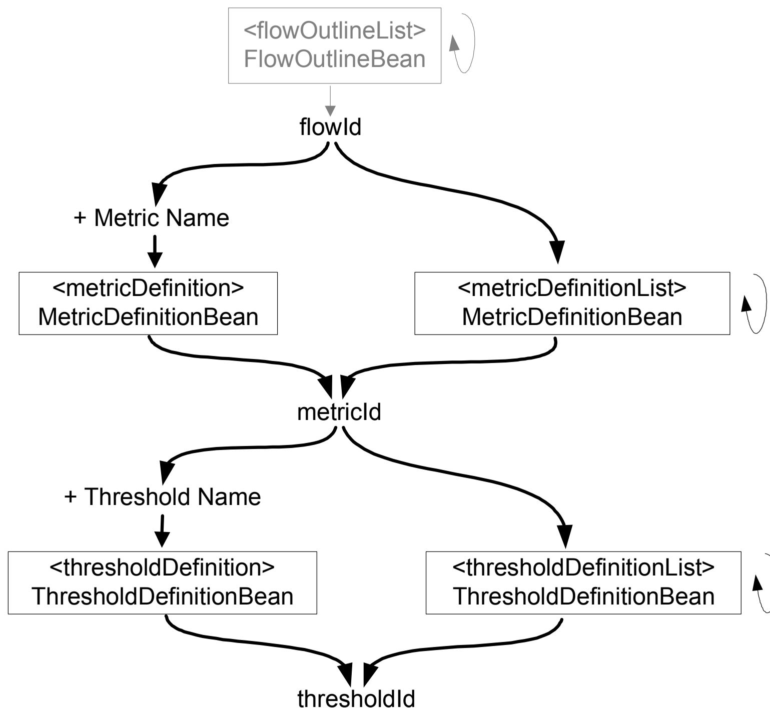
- Definitions
- Statistical data and graphs
- Instance values
- Dials
- Alerts

This chapter looks at each group, and provides worked examples showing how to call the tags.

Definitions

The following diagram lists the <metrics> tags used for accessing metric and threshold definitions:

Figure 1 <metrics> Tags - Definitions



On the diagram:

- Each box lists the tag name, and underneath that is the type of Java bean that the tag returns.
- There is a little “loop” symbol to the right of each “list” tag to indicate that this tag typically loops through the return values.

The goal of the diagram is to help you see the typical calling sequence for the tags. For example:

- All metric definitions refer to a specific flow, thus you need to start with a `flowId`. You can use the `flowOutlineList` tag to loop through the available flows and get the `flowId`.

- Once you have the `flowId`, you might then call the `metricDefinitionList` tag, passing in this `flowId`. The `metricDefinitionList` loops through all the metrics defined for this `flowId` and, within each loop, returns to you a `MetricDefinitionBean`.

  Alternatively, once you have the `flowId`, if you know the name of the particular metric you are interested in, you can pass the `flowId` and the name of the metric to the `metricDefinition` tag. This returns a `MetricDefinitionBean` for your metric.

- Once you have the `MetricDefinitionBean` you are able to get the `metricId`.

- Once you have the `metricId`, you can then use the `thresholdDefinitionList` tag to loop through the thresholds defined for this metric - passing in the `metricId`.

  Alternatively, if you know the name of the threshold you are interested in, you can pass the `metricId` and the name of the threshold to the `thresholdDefinition` tag.

Code Examples

Here are some example JSPs that use the <metrics> tags to display details of the metric and threshold definitions.

Accessing Definitions By Name

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - Metric/Threshold definitions by name</title></head>
  <body>
  <!-- 1 -->
  <%
      String flowId = "";
      String flowName = "Call System";
  %>

  <flow:flowOutlineList var="flowOutlineBean" nameFilter="<%= flowName %>" />

  <%
      if (flowOutlineBean != null)
      {
          flowId = flowOutlineBean.getFlowId();
      }
  %>

  <h1>Metric/Threshold Details - using names</h1>

  <!-- 2 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName="Call Assignment Time" />

  <%
     if (metricDefBean == null)
     {
       return;
     }
  %>

  Metric Name:  [Call Assignment Time] <br />
  Description:  [<%= metricDefBean.getMetricDescription() %>] <br />
  Type:         [<%= metricDefBean.getMetricType() %>] <p />
```

```
  <!-- 3 -->
  <metrics:thresholdDefinition var="thresholdDefBean"
                               metricId="<%= metricDefBean.getMetricId() %>"
                               thresholdName = "Call Assignment SLA" />

  Threshold:   [<%= thresholdDefBean.getThresholdName() %>]<br />
  Description: [<%= thresholdDefBean.getThresholdDescription() %>]<br />
  Type:        [<%= thresholdDefBean.getThresholdType() %>]<br />
  Alert:       [<%= thresholdDefBean.getCurrentAlertLevel() %>]<br />

  </body>
</html>
```

where:

- You declare the taglib:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
```

  This tells the JSP that you want to use the Java taglib `com.hp.ov.bia.views.taglibs.metrics`. The `prefix` option says that you will refer to this taglib within the rest of the JSP by the tag name `metrics`.

- Step **1** uses the `flowOutlineList` flow tag to determine the flowId for the flow `Call System`.
- Step **2** uses the `metricDefinition` tag, passing in the `flowId` and the name of a specific metric. The JSP then displays some data from the returned `MetricDefinitionBean`.
- Step **3** uses the `thresholdDefinition` tag, passing in the `metricId` and a specific threshold name. The JSP then displays some of the data available from the returned `ThresholdDefinitionBean`.

Accessing Definitions Using Lists

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - List Metric/Threshold definitions</title></head>
  <body>
  <!-- 1 -->
  <%
      String flowId = "";
      String flowName = "Call System";
  %>

  <flow:flowOutlineList var="flowOutlineBean" nameFilter="<%= flowName %>" />

  <%
      if (flowOutlineBean != null)
      {
          flowId = flowOutlineBean.getFlowId();
      }
  %>

  <h1>Metric/Threshold Details - using lists</h1>

  <table border=1>

    <tr>
        <th align="left">Name</th>
        <th align="left">Description</th>
        <th align="left">Type</th>
        <th align="left">Status</th>
    </tr>
```

```
    <!-- 2 -->
    <metrics:metricDefinitionList var="metricDefBean" flowId="<%= flowId %>">
      <tr>
         <td align="left"><%= metricDefBean.getMetricName() %></td>
         <td align="left"><%= metricDefBean.getMetricDescription() %></td>
         <td align="left"><%= metricDefBean.getMetricType() %></td>
         <td> </td>
      </tr>

      <!-- 3 -->
      <metrics:thresholdDefinitionList var="thresholdDefBean"
                    metricId="<%= metricDefBean.getMetricId() %>">
        <tr>
         <td align="right"><%= thresholdDefBean.getThresholdName() %></td>
        <td align="left"><%=thresholdDefBean.getThresholdDescription()%></td>
         <td align="left"><%= thresholdDefBean.getThresholdType() %></td>
         <td align="left"><%= thresholdDefBean.getCurrentAlertLevel() %></td>
        </tr>
      </metrics:thresholdDefinitionList>

    </metrics:metricDefinitionList>

  </table>

  </body>
</html>
```

where:

- You declare the taglib:

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
```

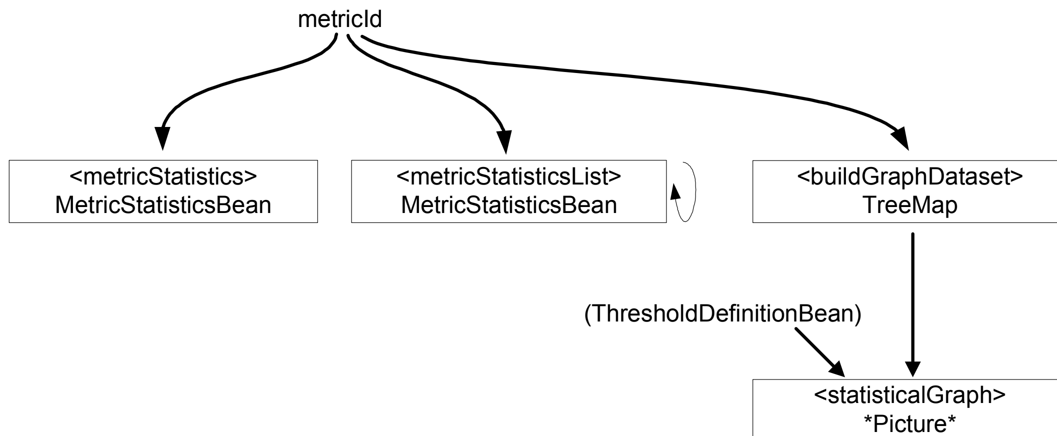
This tells the JSP that you want to use the Java taglib `com.hp.ov.bia.views.taglibs.metrics`. The `prefix` option says that you will refer to this taglib within the rest of the JSP by the tag name `metrics`.

- Step **1** uses the `flowOutlineList` flow tag to determine the flowId for the flow `Call System`.
- Step **2** uses the `metricDefinitionList` tag to loop through all the defined metrics for the given `flowId`.
- Step **3** uses the `thresholdDefinitionList` tag to loop through all the thresholds that are defined for each metric.

Statistical Data and Graphs

The following diagram lists the <metrics> tags used for accessing metric statistics, and for producing historical graphs:

Figure 2 <metrics> Tags - Statistical Data and Graphs



On the diagram:

- Each box lists the tag name, and underneath that is the type of Java bean that the tag returns. For the `statisticalGraph` tag, that does not return a Java bean, it produces an actual picture.
- There is a little "loop" symbol to the right of the `metricStatisticsList` tag to indicate that this tag typically loops through the return values.

metricStatistics

The `metricStatistics` tag accesses the `metric_fact_statistics` table within the OVBPI database.

You can request the latest set of statistics and this gives you the statistics calculated over the most recent collection interval. You can also ask for the statistics from a specific collection interval.

You are also able to ask for the statistics over a longer period of time, for example, the statistics over the last hour. If you ask for the statistics over a time period, the `metricStatistics` tag gives you a single bean containing the results of the overall average, overall standard deviation, etc. calculated across the time period specified.

metricStatisticsList

The `metricStatisticsList` tag accesses the `metric_fact_statistics` table within the OVBPI database.

You can request the statistics over a specified period of time, for example, the statistics over the last hour. The `metricStatisticsList` tag enables you to iterate through each set of results across the specified time period.

buildGraphDataset

The `buildGraphDataset` tag accesses the `metric_fact_statistics` table within the OVBPI database.

The `buildGraphDataset` tag provides an easy way to get the statistical metric data over a period of time, and build a Java bean that is ready to be passed in to the `statisticalGraph` tag. Basically, the result of the `buildGraphDataset` tag is a sorted list (a Java TreeMap) of statistical values over the time period specified. When you call the `buildGraphDataset` tag you specify the time period and the particular field, or fields, of data that you wish to retrieve.

statisticalGraph

The `statisticalGraph` tag draws a time-series graph based on the data as passed in from the `buildGraphDataset` tag.

If you wish the graph to have a legend showing the threshold settings then you can also pass in a `ThresholdDefinitionBean`.

Code Examples

Here are some example JSPs that use the <metrics> tags to display the statistical metric data, and to display statistical graphs.

Listing Metric Statistics

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - List the statistics</title></head>
  <body>
  <%
      String flowId = "";
      String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Calls Resolved ON Contract" />

  <h1>Statistics - over last hour</h1>

  <table border=1>
    <tr>
      <th>Time Recorded</th>
      <th>Average</th>
      <th>Standard Deviation</th>
      <th>Sample Count</th>
    </tr>
    <!-- 2 -->
    <metrics:metricStatisticsList var="metricStatsBean"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            infoType="Completed"
                            timeFrom="-1 H" >
    <tr>
      <td><%= metricStatsBean.getTimeOfLastUpdate() %></td>
      <td><%= metricStatsBean.getAverage() %></td>
      <td><%= metricStatsBean.getStandardDeviation() %></td>
      <td><%= metricStatsBean.getCount() %></td>
    </tr>

    </metrics:metricStatisticsList>

  </table>
```

```
  </body>
</html>
```

where:

- Step **1** uses the flowId to look up the metric definition for the Calls Resolved ON Contract metric.
- Step **2** uses this metricId to call the metricStatisticsList tag. This tag loops through all the metric statistics returned.

  The metricStatisticsList tag accesses the metric_fact_statistics table (within the OVBPI database). At the end of each collection interval three records are written to this table: Active, Completed and Total. The infoType parameter allows you to specify whether you want the Active, Completed or Total statistics record.

  There are two parameters that allow you to specify the time period for which you want the statistical data, timeFrom and timeTo.

  — The timeFrom parameter allows you to specify the start time. This can be an actual Java date/time object or a relative time from now. The above example shows the use of a relative time, -1 H, which means "The previous one hour".

  — If no timeTo parameter is supplied then the tag assumes that you want everything up to now.

Graphing Statistical Data - Example 1

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - Show a bar/line graph</title></head>
  <body>
  <%
     String flowId = "";
     String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Calls Resolved ON Contract" />

  <!-- 2 -->
  <metrics:thresholdDefinition var="thresholdDefBean"
                           metricId="<%= metricDefBean.getMetricId() %>"
                           thresholdName = "ON Contract Calls Resolved" />

  <h1>Bar Graph</h1>

  <!-- 3 -->
  <metrics:buildGraphDataset var="graphDataset"
                             metricId="<%= metricDefBean.getMetricId() %>"
                             infoType="Completed"
                             timeFrom="-1 H"
                             fields="avg"/>

  <!-- 4 -->
  <metrics:statisticalGraph imageBackgroundColor="#FFFFFF"
                      data="<%= graphDataset %>"
                      XAxisTitle="Time"
                      YAxisTitle="Percent (%)"
                      graphTitle="<%= thresholdDefBean.getThresholdName() %>"
                      thresholdDefinitionBean="<%= thresholdDefBean %>"
                      graphType="bar"
                      timeFrom="-1 H"
            collectionInterval="<%= metricDefBean.getCollectionInterval() %>"
                      width="700"
                      height="400"/>

  </body>
</html>
```

where:

- Step **1** uses the flowId to look up the metric definition for the Calls Resolved ON Contract metric.
- Step **2** gets a ThresholdDefinitionBean for the threshold ON Contract Calls Resolved. The only reason for getting this threshold bean is so you can pass it into the statisticalGraph tag and have it draw the threshold legend on the graph.
- Step **3** builds the data to be graphed.

  The infoType parameter allows you to specify which statistics you are retrieving - Active, Completed or Total.

  The time period for the data retrieved is specified with the timeFrom and timeTo parameters. If no timeTo parameter is passed in then the tag assumes that you want the statistics up to now. The times can be specified as an actual Java date/time object or, as shown in this example, a relative time. Passing timeFrom=-1 H means “the previous one hour”.

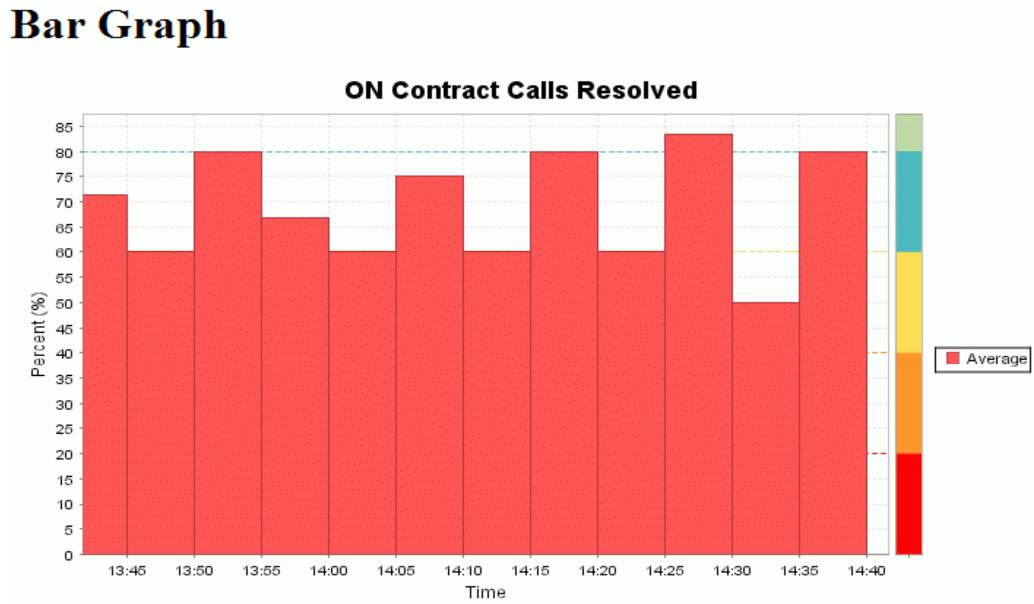
  The fields parameter specifies which statistical data attributes you wish the tag to retrieve from the database. For example, you could ask for avg|min|max and this would retrieve those three values for each set of statistics over the time period. The above example retrieves the average value (fields="avg").
- Step **4** passes the statistical data set to the statisticalGraph tag and draws this data as a bar graph.

  The statisticalGraph tag allows you to specify timeFrom and timeTo parameters. This allows you to draw only that period from the data set you are passing in. The above example displays the same time period as the graph data set contains (-1 H).

  The collectionInterval parameter is required so that the graph knows the time interval that each bar on the graph represents. In other words, it allows the bar graph to show the correct width of each bar.

When you run the JSP it might produce a graph as shown in Figure 3:

Figure 3 Statistical Bar Graph



Graphing Statistical Data - Example 2

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - Show a bar/line graph using groups</title></head>
  <body>
  <%
     String flowId = "";
     String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Call Processing Time" />

  <h1>Bar Graph - showing groups</h1>

  <!-- 2 -->
  <metrics:thresholdDefinition var="thresholdDefBean"
                               metricId="<%= metricDefBean.getMetricId() %>"
                               thresholdName = "Call Processing Speed" />
  <!-- 3 -->
  <metrics:buildGraphDataset var="graphDataset"
                             metricId="<%= metricDefBean.getMetricId() %>"
                             infoType="Completed"
                             getGroupInfo="true"
                             timeFrom="-1 H"
                             fields="avg"/>
  <!-- 4 -->
  <metrics:metricStatistics var="metricStatsBean"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            infoType="Total"
                            age="LATEST" />
<%
  String heading = thresholdDefBean.getThresholdName() + " grouped by: "
                                            + metricDefBean.getGroupName();
%>
```

```
  <!-- 5 -->
  <metrics:statisticalGraph imageBackgroundColor="#FFFFFF"
                  data="<%= graphDataset %>"
                  XAxisTitle="Time of day"
                  graphTitle="<%= heading %>"
                  autoScaleYAxis = "true"
                  thresholdDefinitionBean="<%= thresholdDefBean %>"
                  graphType="bar"
                  timeFrom="-2 H"
                  graphSeriesColors="blue,green,black"
         collectionInterval="<%= metricDefBean.getCollectionInterval() %>"

            average="<%= metricStatsBean.getAverage() %>"
            standardDeviation="<%= metricStatsBean.getStandardDeviation() %>"

                  width="700"
                  height="400"/>

  </body>
</html>
```

where:

- Step **1** uses the `flowId` to look up the metric definition for the `Call Processing Time` metric.
- Step **2** gets a `ThresholdDefinitionBean` for the threshold `Call Processing Speed`. The only reason for getting this threshold bean is so you can pass it into the `statisticalGraph` tag and have it draw the threshold legend on the graph.

  Be aware that this `Call Processing Speed` threshold is a `Relative` threshold, and this means that when you come to pass the statistical data to the `statisticalGraph` tag, you need to pass in the `average` and `standardDeviation` parameters (see step **4**).
- Step **3** builds the data to be graphed.

  The `getGroupInfo="true"` parameter tells the `buildGraphDataset` tag to retrieve the statistical data for each individual group.
- Step **4** is necessary to get the `MetricStatisticsBean`. You need to get the overall average and standard deviation for this metric so that these can be passed into the `statisticalGraph` tag. These are required purely so that the threshold (a `Relative` threshold) can be correctly displayed within the graph.

By calling the `metricStatistics` tag with `infoType="Total"` and `age="LATEST"`, you retrieve a `MetricStatisticsBean` that contains the details for the overall metric statistics - the statistics since the metric was first defined. This included the overall average and standard deviation for the metric.

- Step **5** is where you call the `statisticalGraph` tag to draw your graph.

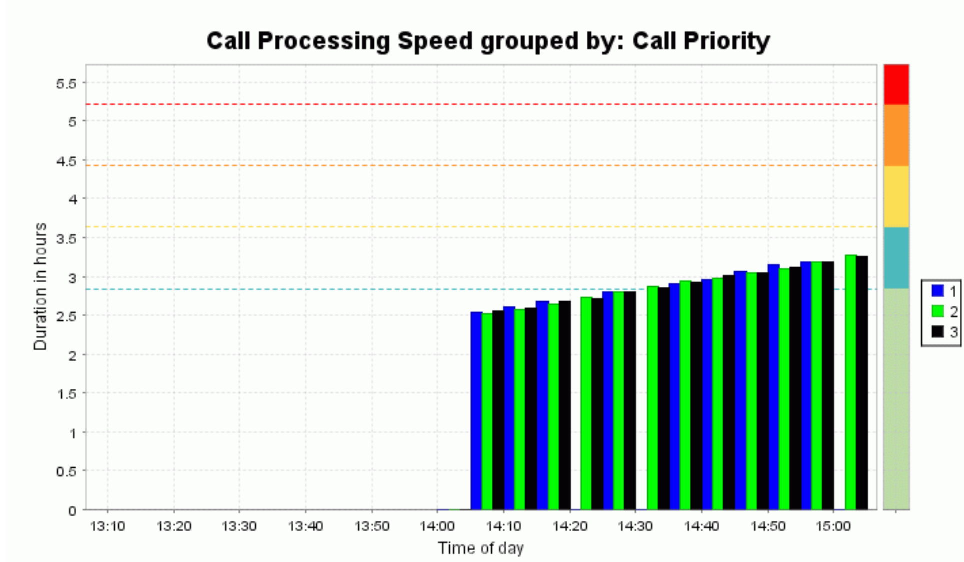
  You pass in the overall average and standard deviation because you are graphing a `Relative` threshold.

  Because the values being graphed are duration values, you can choose the `autoScaleYAxis` parameter. This means that the `statisticalGraph` tag automatically scales the Y axis to fit the data being displayed. It also means that you do not need to provide a Y axis title.

  This example also shows that the `timeFrom` parameter is `-2 H` whereas the time period retrieved by the `buildGraphDataset` tag is `-1 H`. This means that the graph shows a longer time period than the retrieved data, which means that the graph shows no values for the first hour. This is just shown here to show that the time periods of the `statisticalGraph` tag and the `buildGraphDataset` tag can be different.

When you run the JSP it might produce a graph as shown in Figure 4:

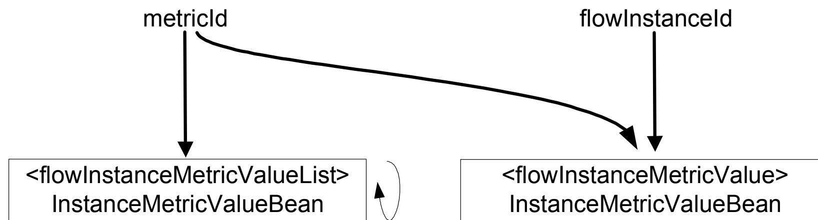
Figure 4 Statistical Bar Graph - showing groups



Instance Values

The following diagram lists the <metrics> tags used for accessing metric instance values for individual flow instances:

Figure 5 <metrics> Tags - Instance Values



where:

- Each box lists the tag name, and underneath that is the type of Java bean that the tag returns.
- There is a little “loop” symbol to the right of the `flowInstanceMetricValueList` tag to indicate that this tag typically loops through the return values.

flowInstanceMetricValueList

This tag accesses the `metric_fact_values` table within the OVBPI database.

This tag returns all the metric instance values that have been recorded for the given `metricId`. The tag returns an `InstanceMetricValueBean` for each metric instance value.

For each metric instance value returned, you are able to get the `flowInstanceId` of the actual flow instance that generated the metric instance.

flowInstanceMetricValue

This tag accesses the `metric_fact_values` table within the OVBPI database.

This tag is used when you know the `metricId` and the `flowInstanceId` that you are interested in. This allows you to retrieve the metric value produced when this flow instance ran.

Because a flow instance might generate a number of metric instances, you have the ability to specify an index (starting from zero).

Code Example

Here is an example JSP that uses the <metrics> tags to display the metric instance data.

Listing Metric Instance Values

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>

<html>
  <head><title>Metric Instance Values</title></head>
  <body>
  <%
    String flowId = "";
    String flowName = "Call System";
  %>

  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <h1>Metric Values:</h1>
```

```
  <!-- 1 -->
  <metrics:metricDefinitionList var="metricDefBean" flowId="<%= flowId %>" >

    <h2><%= metricDefBean.getMetricName() %></h2>

    <table border="1">
    <tr>
      <th>FlowInstId</th>
      <th>Start</th>
      <th>End</th>
      <th>Value</th>
      <th>Status</th>
    </tr>
    <!-- 2 -->
    <metrics:flowInstanceMetricValueList var="InstanceMetricValueBean"
                             metricId="<%= metricDefBean.getMetricId() %>"
                             maxInstances="20">
      <tr>
        <td><%= InstanceMetricValueBean.getFlowInstanceId() %></td>
        <td><%= InstanceMetricValueBean.getStartTime() %></td>
        <td><%= InstanceMetricValueBean.getEndTime() %></td>
        <td><%= InstanceMetricValueBean.getValue() %></td>
        <td><%= InstanceMetricValueBean.getStatus() %></td>
      </tr>

    </metrics:flowInstanceMetricValueList>

    <table>

  </metrics:metricDefinitionList>

  </body>
</html>
```

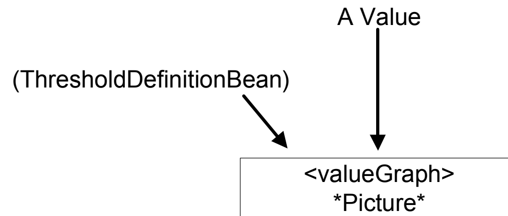
where:

- Step **1** uses the `flowId` to loop through all the metric definitions.
- Step **2** loops through all the metric instance values recorded for the given `metricId`. For each metric instance value, the JSP displays some of the details, including the flow instance that produced each metric instance value.

Dials

The following diagram shows the <metrics> tag used for producing dials:

Figure 6 <metrics> Tags - Dials



The `valueGraph` tag is a general purpose tag that allows you to draw a dial representing a value. If you pass in a threshold definition bean then the dial is drawn showing the threshold and the value is represented against that threshold.

The `valueGraph` tag is able to draw two types of dials:

1. A regular dial (`graphType="dial"`)

   A regular dial is to be used when you are graphing a value against a threshold that is one of the types:

   — Absolute Duration/Weight/Value

   — Backlog

   — Throughput

   In the `<metrics>` Java docs, these thresholds are collectively referred to as `overValue` and `underValue` threshold types. This is because they are thresholds that measure whether a metric value is over or under specific values.

   If you call the `valueGraph` tag, asking to draw a `dial`, and you pass in a `ThresholdDefinitionBean` that is not of type Absolute, Backlog or Throughput, the threshold is not shown on the graph.

2. A swing dial (graphType="swingdial")

   A swing dial, or "upside down dial", is used when you are graphing a value against a threshold that is one of the types:

   — Relative

   — Deadline

   In the <metrics> Java docs, the Relative threshold type is referred to by the terms overUsual, underUsual and unusual. This is because the Relative threshold type is measuring where a metric value is relative to the "usual" behavior.

   If you call the valueGraph tag, asking to draw a swingdial, and you pass in a ThresholdDefinitionBean that is not of type Relative or Deadline, the threshold is not shown on the graph.

Code Examples

Here are some example JSPs that use the <metrics> tags to display dials.

Dial - Most Recent Average

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - Show a dial</title></head>
  <body>
  <%
     String flowId = "";
     String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Calls Resolved ON Contract" />

  <!-- 2 -->
  <metrics:thresholdDefinition var="thresholdDefBean"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            thresholdName = "ON Contract Calls Resolved" />
```

```
<%
  if (thresholdDefBean == null)
      return;
%>
 <h1>Most recent average value:</h1>

  <!-- 3 -->
  <metrics:metricStatistics var="metricStatisticsBean"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            infoType="Completed"
                            age="LATEST" />
  <%
    Float value = null;
    String formattedValue = "";

    <!-- 4 -->
    if (metricStatisticsBean != null)
    {
      value = metricStatisticsBean.getAverage();
      if (value == null)
      {
            formattedValue = "No value available!";
      }
      else
      {
            formattedValue = value + " Percent";
      }
  %>
      <table border="0">
        <tr>
           <td align="center"><%= thresholdDefBean.getThresholdName() %></td>
        </tr>
        <tr>
            <td>
                <!-- 5 -->
                <metrics:valueGraph imageBackgroundColor="white"
                           width="150"
                           height="100"
                           graphType="dial"
                           graphBackgroundColor="#EBEBEB"
                           legend="false"
                           value="<%= value %>"
                           thresholdDefinitionBean="<%= thresholdDefBean %>"
                           graphValueLabel="<%= formattedValue %>" />
            </td>
        </tr>
      </table>
  <%
    }
  %>
```

```
  </body>
</html>
```

where:

- Step **1** uses the `flowId` to look up the metric definition for the `Calls Resolved ON Contract` metric.
- Step **2** gets a `ThresholdDefinitionBean` for the threshold `ON Contract Calls Resolved`. The code then checks that the threshold actually exists. The code gets the threshold so that the threshold can be passed into the `valueGraph` tag when drawing the dial.
- Step **3** is where you get the actual value to be graphed.

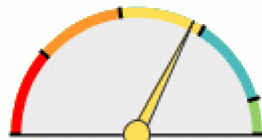
  In this example, the value to be graphed is the most recent average, and to get this data you need to ask for the latest information on all completed metrics - hence the parameters `infoType="Completed"` and `age="LATEST"`.
- Step **4** checks that there is a valid `MetricStatisticsBean`.

  If there is a valid `MetricStatisticsBean`, the code then tries to retrieve the average. The code then builds a formatted display string that is to be used when displaying the dial.
- Step **5** calls the `valueGraph` tag to produce the dial picture showing the average value against the threshold.

  The code just passes in the raw value to the tag. If you wanted the value rounded to (for example) two decimal places then your code needs to do this before passing the value into the `valueGraph` tag.

When you run the JSP it might produce a dial as shown in Figure 7:

Figure 7 Dial - Most Recent Average



Dial - Over Time

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>
<html>
  <head><title>Metrics - Show a dial over time</title></head>
  <body>
  <%
     String flowId = "";
     String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Calls Resolved ON Contract" />
  <!-- 2 -->
  <metrics:thresholdDefinition var="thresholdDefBean"
          metricId="<%= metricDefBean.getMetricId() %>"
          thresholdName = "ON Contract Calls Resolved" />
<%
  if (thresholdDefBean == null)
    return;
%>

  <h1>Average value over time:</h1>

  <!-- 3 -->
  <metrics:metricStatistics var="metricStatisticsBean"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            infoType="Completed"
                            age="-1 H"
                            untilLatest="true" />
  <%
    Float value = null;
    String formattedValue = "";

    if (metricStatisticsBean != null)
    {
      value = metricStatisticsBean.getAverage();
      if (value == null)
      {
        formattedValue = "No value available!";
      }
      else
      {
        formattedValue = value + " Percent";
      }
  %>
```

```
    <table border="0">
      <tr>
        <td align="center"><%= thresholdDefBean.getThresholdName() %></td>
      </tr>
      <tr>
        <td>
            <!-- 4 -->
            <metrics:valueGraph imageBackgroundColor="white"
                                width="150"
                                height="100"
                                graphType="dial"
                                graphBackgroundColor="#EBEBEB"
                                legend="false"
                                value="<%= value %>"
                          thresholdDefinitionBean="<%= thresholdDefBean %>"
                                graphValueLabel="<%= formattedValue %>"
                                  />
         </td>
      </tr>
    </table>
  <%
    }
  %>

  </body>
</html>
```

where:

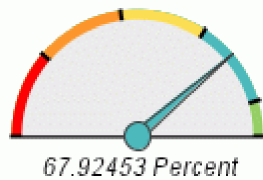
- Step **1** uses the `flowId` to look up the metric definition for the `Calls Resolved ON Contract` metric.
- Step **2** gets a `ThresholdDefinitionBean` for the threshold `ON Contract Calls Resolved`. The code then checks that the threshold actually exists. The code gets the threshold so that the threshold can be passed into the `valueGraph` tag when drawing the dial.
- Step **3** is where you get the actual value to be graphed.

  The `age="-1 H"` parameter you are asking for the metric statistic from the previous one hour. But by also passing the parameter `untilLatest="true"` you are asking the `metricStatistics` tag to calculate and return the overall statistics for the entire one hour period.
- Step **4** actually draws the dial.

  The code just passes in the raw value to the tag. If you wanted the value rounded to (for example) two decimal places then your code needs to do this before passing the value into the `valueGraph` tag.

When you run the JSP it might produce a dial as follows:

Figure 8 Dial - Average Over Time



Swing Dial

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - Show a swing dial</title></head>
  <body>
  <%
     String flowId = "";
     String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Call Processing Time" />
  <!-- 2 -->
  <metrics:thresholdDefinition var="thresholdDefBean"
          metricId="<%= metricDefBean.getMetricId() %>"
          thresholdName = "Call Processing Speed" />

  <h1>Swing Dial</h1>

  <!-- 3 -->
  <metrics:metricStatistics var="metricStatisticsBeanTotal"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            infoType="Total"
                            age="LATEST" />
  <%
    Float overallAvg = null;
    Float overallStdDev = null;

    if (metricStatisticsBeanTotal != null)
    {
      overallAvg    = metricStatisticsBeanTotal.getAverage();
      overallStdDev = metricStatisticsBeanTotal.getStandardDeviation();
    }
    else
    {
      overallAvg    = new Float(0);
      overallStdDev = new Float(0);
    }
  %>
```

```
  <!-- 4 -->
  <metrics:metricStatistics var="metricStatisticsBean"
                            metricId="<%= metricDefBean.getMetricId() %>"
                            infoType="Completed"
                            age="LATEST" />
  <%
    if (metricStatisticsBean != null)
    {
      Float value = metricStatisticsBean.getAverage();
      String formattedValue;
      if (value == null)
      {
        formattedValue = "No value available!";
      }
      else
      {
        formattedValue = value.toString() + " Seconds";
      }
  %>

      <table border="0">
         <tr>
            <td align="center">
                     <%= thresholdDefBean.getThresholdName() %>
            </td>
         </tr>
         <tr>
            <td>
               <!-- 5 -->
               <metrics:valueGraph imageBackgroundColor="white"
                                   width="150"
                                   height="100"
                                   graphType="swingdial"
                                   graphBackgroundColor="#EBEBEB"
                                   legend="false"
                                   value="<%= value %>"
                            thresholdDefinitionBean="<%= thresholdDefBean %>"
                                   graphValueLabel="<%= formattedValue %>"
                                   standardDeviation="<%= overallStdDev %>"
                                   average="<%= overallAvg %>" />
            </td>
         </tr>
      </table>
  <%
    }
  %>

  </body>
</html>
```

where:

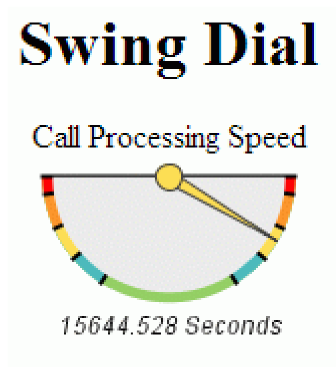
- Step **1** uses the `flowId` to look up the metric definition for the `Call Processing Time` metric.
- Step **2** gets a `ThresholdDefinitionBean` for the threshold `Call Processing Speed`. The code then checks that the threshold actually exists. The code gets the threshold so that the threshold can be passed into the `valueGraph` tag when drawing the dial.
- Step **3** gets the latest total for the statistics. This is so you can get the overall average and standard deviation. You require the overall average and standard deviation so you can pass them to the `valueGraph` tag.
- Step **4** is where you get the actual value to be graphed.

  In this example, the value to be graphed is the most recent average, and to get this data you need to ask for the latest information on all completed metrics - hence the parameters `infoType="Completed"` and `age="LATEST"`.
- Step **5** calls the `valueGraph` tag to produce the swingdial picture showing the average value relative to the threshold. Because you are drawing a swingdial, you are required to pass in the overall average and standard deviation.

  The code just passes in the raw value to the tag. If you wanted the value rounded to (for example) two decimal places then your code needs to do this before passing the value into the `valueGraph` tag.

When you run the JSP it might produce a swing dial as follows:

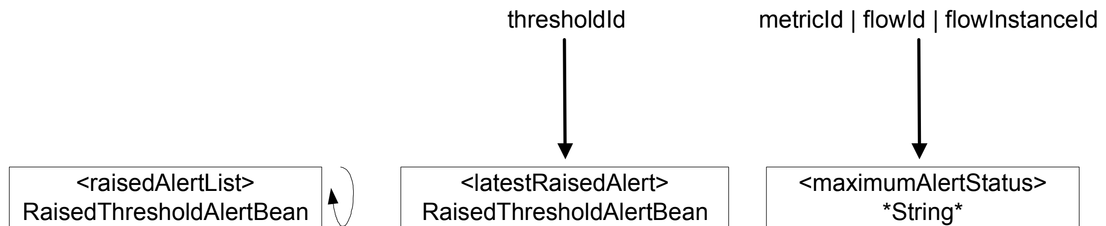
Figure 9 Swing Dial



Alerts

The following diagram lists the <metrics> tags used for accessing metric alerts:

Figure 10 <metrics> Tags - Alerts



where:

- Each box lists the tag name, and underneath that is the type of Java bean that the tag returns. For the `maximumAlertStatus` tag, that does not return a Java bean, it returns the string value of the maximum alert.
- There is a little “loop” symbol to the right of the `raisedAlertList` tag to indicate that this tag typically loops through the return values.

raisedAlertList

This tag allows you to get the alerts that have been raised in a specified time period. There are parameters that you can pass in to the tag to specify the particular metric or flow that you are interested in.

You can use this tag to loop through the alerts that have been raised.

latestRaisedAlert

For a particular threshold, you are able to get the latest alert that has been raised.

maximumAlertStatus

This tag allows you to find out the overall highest alert that has occurred against a given metric, or flow, or flow instance. This tag is used on the main page of the OVBPI Dashboard to show the overall maximum alert against each flow.

Code Example

Here is an example JSP that uses the <metrics> tags to display alerts.

Displaying Alerts

```
<%@ taglib uri="com.hp.ov.bia.views.taglibs.metrics" prefix="metrics" %>
<%@ taglib uri="com.hp.ov.bia.views.taglibs.flow" prefix="flow" %>

<html>
  <head><title>Metrics - List the alerts</title></head>
  <body>
  <%
     String flowId = "";
     String flowName = "Call System";
  %>
  ...get the flow Id (see Accessing Definitions By Name on page 122)...

  <!-- 1 -->
  <metrics:metricDefinition var="metricDefBean"
                            flowId="<%= flowId %>"
                            metricName = "Calls Resolved ON Contract" />

  <h1>Alerts - over last 12 hours</h1>

  <table border=1>
    <tr>
      <th>Metric Name</th>
      <th>Threshold Name</th>
      <th>Status</th>
      <th>Time Raised</th>
    </tr>
```

```
  <!-- 2 -->
  <metrics:raisedAlertList var="raisedAlertBean"
                           metricId="<%= metricDefBean.getMetricId() %>"
                           noOfEntries="1000"
                           timeFrom="-12 H" >
     <tr>
       <td><%= raisedAlertBean.getMetricName() %></td>
       <td><%= raisedAlertBean.getThresholdName() %></td>
       <td><%= raisedAlertBean.getAlertStatus() %></td>
       <td><%= raisedAlertBean.getRaisedTime() %></td>
     </tr>

  </metrics:raisedAlertList>
</table>

<h1>Latest Raised Alert</h1>

<table border=1>
  <tr>
    <th>Threshold Name</th>
    <th>Metric Name</th>
    <th>Status</th>
    <th>Time Raised</th>
  </tr>
  <!-- 3 -->
  <metrics:thresholdDefinitionList var="thresholdDefBean"
                        metricId="<%= metricDefBean.getMetricId() %>" >

    <metrics:latestRaisedAlert var="raisedAlertBean1"
                  thresholdId="<%= thresholdDefBean.getThresholdId() %>" />
    <tr>
      <td><%= raisedAlertBean1.getThresholdName() %></td>
      <td><%= raisedAlertBean1.getMetricName() %></td>
      <td><%= raisedAlertBean1.getAlertStatus() %></td>
      <td><%= raisedAlertBean1.getRaisedTime() %></td>
    </tr>

  </metrics:thresholdDefinitionList>
</table>
```

```
  <h1>Highest Overall Alert for flow (<%= flowName %>)</h1>

  <!-- 4 -->
  <metrics:maximumAlertStatus var="overallStatus" flowId="<%= flowId %>" />

  <table border=1>
    <tr>
      <th>Status</th>
    </tr>
    <tr>
      <td><%= overallStatus %></td>
    </tr>
  </table>

  </body>
</html>
```

where:

- Step **1** uses the `flowId` to look up the metric definition for the `Calls Resolved ON Contract` metric.
- Step **2** calls the `raisedAlertList` tag and loops through the alerts that have occurred during the past 12 hours, for the specified metric.

  The `noOfEntries` parameter allows you to limit the number of alerts returned to your code.
- Step **3** loops through all the thresholds defined for the given metric, showing the latest alert that has been raised for each threshold.
- Step **4** uses the `maximumAlertStatus` tag to find out the overall maximum alert against the given `flowId`.

6

Direct OVBPI Database Access

Using the OVBPI JSP tag libraries, you are able to retrieve most of the details that you would typically need for a business dashboard. However, there may be situations where you require extra details. Maybe some custom calculation that you would like to make across your flow instances, or something that the standard tag libraries just don't provide.

OVBPI maintains all its information in an SQL database. The schema for this database is published in the *OVBPI System Administration Guide*.

This chapter looks at accessing the OVBPI database directly from within your custom dashboard.

Connecting

The Dashboard provides a bean that allows you to connect to the OVBPI database. The bean is:

```
com.hp.ov.bia.views.DBConnection
```

This bean uses the standard Dashboard configuration details (from the `DashboardConfig.properties` file) to connect to the OVBPI database.

The `DBConnection` bean provides a pooled database connection. That is, when you close a connection it is actually pooled and made available for another user. (This pooling mechanism makes use of the Database Connection Pooling library from the Apache organization.)

Before the `DBConnection` bean can be used, it needs to be initialized correctly. To make it easy for you, the OVBPI Dashboard provides a method that does this for you.

The method is called: `getDatabaseConnection()` and it is found in the JSP: `gen/common.jsp`

This method requires that you pass in the current JSP page context object (which is in the pre-defined JSP variable `pageContext`).

This “pooled” database connection **must be closed** before the end of the JSP page. If your JSP does not close the connection, you may cause the underlying database system to run out of resources and your dashboard may eventually hang.

To close the database connection, you simply call the method: `closeConection()` (yes...it is spelt incorrectly :-)

So to access the OVBPI database directly within your JSP, you need to:

- Include `common.jsp`
- Call `getDatabaseConnection()`
- Do whatever SQL you need...
- Close this database connection

The overall outline of your JSP code is as follows:

```
<%@ include file="../gen/common.jsp" %>

<%
    // Get a pooled connection
    DBConnection dbConn = getDatabaseConnection(pageContext);
%>

    ...your database access in here...

<%
    // Close the database connection for this page
    dbConn.closeConection();
%>
```

Issuing SQL Statements

Once you have a database connection you can issue SQL statements against the OVBPI database.

The `DBConnection` bean provides some methods to assist with issuing SQL statements.

There are two main types of SQL queries you can make:

- Fixed SQL statements
- Prepared SQL statements

Fixed SQL Statements

A fixed SQL statement is where you have a complete SQL statement that you wish to execute.

For example:

```
select flow_id, ActiveFlows, flowname, status
  from flows
  where status != 'Deleted'
```

This example returns the specified column values from the `flows` data table, for all flows that were not marked for deletion. (That is, all non-superseded flows.)

Within your Java code can build the required SQL statement and include values from your code. For example:

```
sql = "select flow_id, ActiveFlows, flowname, status from " + flowsTable +
          " where status != '" + flowStatus + "'";
```

This example substitutes the current values for the variables `flowsTable` and `flowStatus` to produce a complete SQL statement. The variable `sql` contains this complete SQL statement.

To execute complete/fixed SQL statements like these, you use the `DBConnection` method: `executeQuery()`

For example:

```
<%@ page import="java.sql.ResultSet"%>
<%
    ResultSet results;
    String sql;

    String flowName = "";
    String flowId = "";
    long   activeCount = 0;
    String flowStatus = "";

    DBConnection dbConn = getDatabaseConnection(pageContext);

    sql = "select flow_id, ActiveFlows, flowname, status from flows " +
          " where status != 'Deleted'";

    results = dbConn.executeQuery(sql);
%>
    <h1>Flow Definition List</h1>
    <table cellspacing="0" border="1">
    <tr>
        <th>Flow Name</th>
        <th>Flow ID</th>
        <th>Active Flow Count</th>
        <th>Flow Status</th>
    </tr>
<%
    while (results.next())
    {
        flowId = results.getString("flow_id");
        flowName = results.getString("flowname");
        activeCount = results.getLong("ActiveFlows");
        flowStatus = results.getString("status");
%>
        <tr>
           <td><%= flowName %></td>
           <td><%= flowId %></td>
           <td><%= activeCount %></td>
           <td><%= flowStatus %></td>
        </tr>
<%
    }
%>
    </table>
<%
    // Close the database connection for this page
    dbConn.closeConection();
%>
```

where:

- You get the database connection:

```
DBConnection dbConn = getDatabaseConnection(pageContext);
```

- You use this database connection to issue the SQL statement:

```
results = dbConn.executeQuery(sql);
```

  This returns the results in a JDBC result set (`java.sql.ResultSet`).

- You then loop through the results, pulling out the values that you want in your JSP:

```
    flowId = results.getString("flow_id");
    flowName = results.getString("flowname");
    activeCount = results.getLong("ActiveFlows");
    flowStatus = results.getString("status");
```

- You display these values on the Web page
- You close the database connection when you are finished

Prepared SQL Statements

The underlying databases supported by OVBPI offer the ability to “prepare” SQL statements. This is where you submit the statement to the database management system (DBMS) and it pre-compiles the statement. This pre-compiled version is then stored internally within the DBMS. You are then able to tell the DBMS to run this pre-compiled (prepared) version of the statement whenever you need it. What’s more, you can specify variables in this prepared statement and provide values for them at run time.

Prepared statements are used when you have a statement that you are likely to want to run over and over again. By preparing it once, all subsequent executions are quicker.

Working with a prepared statement is a two step process. First you need to prepare the statement, then you can execute it.

Preparing the Statement

A prepares SQL statement is stored against your database connection. So the method to prepare a statement is provided by the `DBConnection` bean.

The method is:

```
public PreparedStatement prepareStatement(String sql)
```

You provide the SQL statement (as a `String`) and you are returned a `PreparedStatement` object.

Executing a Prepared Statement

Once you have prepared the statement you are able to execute it whenever you need by passing the `PreparedStatement` object to the method:

```
public ResultSet executePreparedQuery(PreparedStatement prepStatement)
```

This runs your prepared statement and returns the results in a Java result set (`java.sql.ResultSet`) object.

Substituting Values

Prepared statements are able to have values passed in to them at execution time. You mark where these substitutions are to occur by using the question mark (?) character within your SQL. You can not substitute SQL keywords or table names, just data values.

For example:

```
<%@ page import="java.sql.PreparedStatement,
                 java.sql.ResultSet"%>

<%
    flowName = "Order Flow";
    myStatus = "Deleted";

    PreparedStatement prepStatement;

    sql = "select flow_id, ActiveFlows, AvrgTime, status from flows " +
                 " where flowname = ? " +
                 " and status != ? ";

    // Prepare the statement
    prepStatement = dbConn.prepareStatement(sql);

    // Set the values to be used with this execution of the statement
    prepStatement.setString(1, flowName);
    prepStatement.setString(2, myStatus);

    // Execute the prepared statement
    results = dbConn.executePreparedQuery(prepStatement);
%>
```

where:

- You construct the SQL statement as normal, however, you specify a question mark (?) wherever you wish to substitute a run-time value
- You use the `DBConnection` method `prepareStatement()` to prepare this statement, and it returns a `PreparedStatement` object

- When you come to execute this prepared statement, you need to set values for all of your substitutions

  Here you set the two string values for the flow name that you want to find and the flow status that you require.

  If you were substituting a non-string value you have a range of different methods available. For example:

  ```
  setInt()
  setDouble()
  setTimestamp()
  etc.
  ```

  Refer to the Java JDK documentation of `java.sql.PreparedStatement` for all available methods.

- You call the `executePreparedQuery()` method (on the database connection) to execute the prepared statement, and get the results

Which One to Use?

Most of your queries are probably going to be fixed SQL statements.

There are times when it is best to use a prepared statement. Here are some tips:

- If you need to substitute a date/time value into an SQL statement then you may find it easier to use a prepared statement

  With a fixed SQL statement it can be difficult to get the date format correct.

- If you need to issue an SQL statement while you are looping through a previous SQL statement's result set, you need to use prepared statements

  If you try to do nested SQL using fixed SQL statements you get a "Result set closed" error when you try to issue the second/inner SQL statement.

Additional Helper Beans

There are two additional Java beans that are worth knowing about:

- `com.hp.ov.bia.views.util.Constants`
- `com.hp.ov.bia.views.DBSql`

Constants Bean

This bean contains a set of defined constants that might be helpful when writing your SQL.

The following SQL statement was shown in an earlier code example:

```
sql = "select flow_id, ActiveFlows, flowname, status from flows " +
         " where status != 'Deleted'";
```

The `status != 'Deleted'` relies on the fact that the string written into the OVBPI flows data table for a superseded flow is `'Deleted'`. There is nothing wrong with using this, however you might wish to write your SQL where any OVBPI text strings are listed externally in one place. The `Constants` bean provides a list of known OVBPI strings used within the database.

You can refer to the javadocs for the `Constants` bean for all the available constants. (see *OVBPI-CD*`\docs\html\OVBPIJavadoc\index.html`)

Some example constants:

```
FLOW_STATE_COMPLETED  = 'Completed'
FLOW_STATE_DELETED    = 'Deleted'
FLOW_STATE_IMPACTED   = 'Blocked'
FLOW_STATE_IMPEDED    = 'Impeded'
```

As well as the javadocs, you can also access the Java code for the `Constants` bean. The code is located in:

```
webapps\ovbpidashboard\WEB-INF\classes\
                                com\hp\ov\bia\views\util\Constants.java
```

This means that, instead of writing your SQL as:

```
sql = "select flow_id, ActiveFlows, flowname, status from flows " +
         " where status != 'Deleted'";
```

you could write it as:

```
sql = "select flow_id, ActiveFlows, flowname, status from flows " +
         " where status != '" + Constants.FLOW_STATE_DELETED + "'";
```

The only difference is that, should OVBPI ever change the text values used within the database, then the `Constants` file would also be updated to reflect the new values...and the second code example (that uses the constants from the `Constants` bean) continues to work.

DBSql Bean

This bean contains a set of methods that issue SQL calls against the OVBPI database. These methods were written for the first generation of the OVBPI Dashboard - called the “Web console”. These methods are all still valid.

Most of the methods within the `DBSql` bean provide information that you can already request using the `flow` tag library, so you probably don’t need to use them.

Associated Data Table

One of the reasons you would typically need to write custom SQL against the OVBPI database is if you need to construct a query across both your flow instances and their associated/related data instances. That is, your flow might be tracking orders and, for each order, you are holding information such as:

```
OrderID
ShipAddress
Value
CustID
CustAddress
CustType
```

If you want to construct an SQL query that filters all flow instances for certain customers and orders, you need to be able to join the `flow instance` data table with this flow's associated data table. (see *OVBPI System Administration Guide* for details of the database schema.)...but how can you get the name of this associated data table?

Getting the Name of the Data Table

Each flow definition, within the `Flows` table, contains the ID of its associated/related data table. This data table ID is held in the `Primary_Entity` column. The `Primary_Entity` value corresponds to an entry in the `Data_Objects` table. When you look up the entry in the `Data_Objects` table, the column `InstanceTable` contains the name of the associated/related data table.

Let's look at some example code for returning the data table name, given a connection to the OVBPI database and the flowID. (This example code is written within a JSP so that it can be included easily in calling JSPs. You can obviously choose to write this as a standalone Java bean if you wish...)

Here is a JSP code example that returns the name of the associated data table, given a connection to the OVBPI database and a flow ID:

```
<%!
public String getDataTableForFlowID(DBConnection dbConn, String flowId)
      throws Exception
{
    String sql;
    java.sql.ResultSet results = null;

    // Find the ID of the table name
    // ----------------------------
    sql = "select primary_entity from flows where flow_id = '" + flowId + "'";

    results = dbConn.executeQuery(sql);

    String dataTableId = "";
    if (results.next())
    {
        dataTableId = results.getString("primary_entity");
    }

    // Now look up the actual table name in the Data_Objects table
    // -----------------------------------------------------------
    sql = "select InstanceTable from data_objects where model_id = '" +
                                                      dataTableId + "'";
    results = dbConn.executeQuery(sql);

    String dataTable = "";
    if (results.next())
    {
        dataTable = results.getString("InstanceTable");
    }

    return dataTable;
}
%>
```

where:

- This example is written within a JSP structure
- The code uses the `Flows` table to locate the ID of the associated data table within the `Data_Objects` table.
- Once you have the data table ID, the code looks this up in the `Data_Objects` table to get the actual name of the data table.

Displaying the Associated Data Table Name

Here's in a example code segment that calls the `getDataTableForFlowID()` method to display the associated data table name for a given flow:

```
<%
    // Get the flow ID
    // ---------------

    String flowId = "";
    String flowName = "Order Flow";
%>

<h1>Flow Name is: <%= flowName %></h1>
<flow:flowOutlineList var="flowOutlineBean" nameFilter="<%= flowName %>" />

<%
    if (flowOutlineBean != null)
    {
        flowId = flowOutlineBean.getFlowId();
    }

    // Get the associated data table name
    // ----------------------------------
    DBConnection dbConn = getDatabaseConnection(pageContext);
%>

<%@ include file="ex_db_methods.jsp" %>
<%
    String dataTable = getDataTableForFlowID(dbConn, flowId);
%>

<h3>Associated Datatable name is: <%= dataTable %></h3>
<%
    // Close the database connection for this page
    dbConn.closeConection();
%>
```

where:

- The code to return the name of the associated data table is held in the file `ex_db_methods.jsp`.
- The `getDataTableForFlowID()` method then returns the name of the associated data table for this flow

  You pass in both the current database connection and the flow ID.

SQL Issues

Deadlocks When Using a Microsoft SQL Server Database

It seems that, when using the MSSQL database system, you can create deadlocks quite easily.

The Deadlock Issue

If you issue SQL select statements that join tables together, it is possible to create a deadlock situation with other applications joining the same table...if they join the tables in a different order to you.

So, if you intend to use SQL select statements that join multiple tables together, you are advised to set the priority for database access to be low. You do this using the following MS SQL statement:

```
set deadlock_priority low;
```

You then need to write your custom dashboard to retry the SQL statement if it fails to access the data.

This prevents deadlock situations occurring in cases where your custom dashboard and the OVBPI Engine are accessing database tables at the same time. In this case, your custom dashboard might initially fail to access the data; however, it would subsequently retry.

By setting the access priority to low for applications, the OVBPI Engine has priority and its performance is not impacted by an application that requires only read-access to the data; see The DBConnection Bean on page 169.

This deadlock issue **does not apply** to a OVBPI implementation using an Oracle Server.

The DBConnection Bean

The `DBConnection` bean sets the deadlock priority to low and all SQL statements are performed within a retry loop. So, if you use the `DBConnection` bean to issue all your SQL statements against the OVBPI database, all deadlock and retry issues are handled for you.

Select Columns by Name

When developing your own SQL statements, you are advised against using `select *` statements; you are advised to specify all the required columns by name. The reason for this is one of future maintenance. If you select the columns by name, this continues to work if and when any additional columns are added to that table in a future product release. However, if you use the `select *` command, it may cause your customized dashboard to fail when used with a later version of OVBPI.

Example Customizations

Listing Specific Flow Instances

Let's consider an example where you put together a JSP page that lists all the flow instances that have taken longer than two hours to get to a given node

Assume you are given three things:

- The name of the flow
- The name of the node

  This is the node that flow instances must have reached within the specified duration limit.
- The duration limit

The basic steps you need to follow are:

- Given the flow name, use the `flowOutlineList` tag to get the flow ID
- Convert the given node name into a node ID

  Use the `flow` tag, followed by the `flowNodeList` tag (with a name filter).
- Get a connection to the OVBPI database
- Issue the SQL

  This needs to find all flow instances that are currently active and have not yet reached the given node ID. This means that the node ID is in a state of "Initial".
- Loop through the returned list and display those that have been running for longer than the specified time (two hours)
- Close the database connection

Let's consider this in two parts, looking firstly at the required SQL query, and then how to call this SQL from within your JSP.

The SQL Query

The actual SQL required for this customization can be written as a reusable method. You could then place this method within the calling JSP, in its own JSP or within something like `gen/common.jsp`.

The method you require is as follows:

```
<%!
  public java.sql.ResultSet notYetHere(DBConnection dbConn, String nodeId)
  throws Exception
  {
      String sql;
      java.sql.PreparedStatement prepStatement;
      java.sql.ResultSet results;

  sql = "select flow_instance.flowinstance_id, flow_instance.identifier," +
              " flow_instance.starttimelongmillis " +
          " from node_instance, flow_instance " +
 " where node_instance.flowinstance_id = flow_instance.flowinstance_id " +
 " and   node_instance.status = '" + Constants.NODE_STATE_NOT_STARTED + "'" +
 " and   node_instance.node_id = ?" +
 " and flow_instance.status != '" + Constants.FLOW_STATE_COMPLETED + "'";

      prepStatement = dbConn.prepareStatement(sql);
      prepStatement.setString(1, nodeId);

      results = dbConn.executePreparedQuery(prepStatement);

      return results;
  }
%>
```

Let's look at the SQL statement in more detail:

- The selected columns

```
select flow_instance.flowinstance_id, flow_instance.identifier,
       flow_instance.starttimelongmillis
```

This selects three columns from the database. All three columns come from the `flow_instance` data table. The reason each column name is prefixed with the actual name of the `flow_instance` table is because some of the selected column names also appear in the `node_instance` table. The prefix makes sure that you fully specify each column to say exactly which data table you are referring to.

- The FROM clause

```
from node_instance, flow_instance
```

The columns are selected from a join of two data tables. This `select` statement joins the `node_instance` table with the `flow_instance` table.

- The WHERE clause

```
where node_instance.flowinstance_id = flow_instance.flowinstance_id
```

Because the `select` is joining two tables together, it must specify the column on which the tables is to be joined. This is specified in the `WHERE` clause of the select statement.

This part of the `where` clause is saying that the select must join entries where the value in the `flowinstance_id` column of the `node_instance` table must match the value in the `flowinstance_id` column of the `flow_instance` table. In other words, match up the entries for the same flow instance ID.

```
and   node_instance.status = '" + Constants.NODE_STATE_NOT_STARTED + "'"
```

This part of the `where` clause says that the select statement should match entries only where the `status` of the node instance is equal to the string `Initial`. (`Constants.NODE_SATE_NOT_STARTED` = “Initial”)

```
and   node_instance.node_id = ?
```

This part of the `where` clause specifies that the node ID will be passed in to this statement at run time.

```
and flow_instance.status != '" + Constants.FLOW_STATE_COMPLETED + "'"
```

This part of the `where` clause says that the select statement should select entries only where the `status` column in the `flow_instance` table is not equal to the string `Completed`. (`Constants.FLOW_STATE_COMPLETED` = “Completed”)

In other words, select only flow instances that are still actively running.

The above SQL `select` statement joins the `node_instance` table with the `flow_instance` table, to return all flow instances that have not yet reached the given node ID. It returns a Java result set containing an entry for each matching flow instance. Each entry within this result set contains the following three columns:

```
flowinstance_id
identifier
starttimelongmillis
```

The caller of this method is then able to loop through and process this result set.

The Calling JSP

Let's look at a code extract from a JSP that uses your `notYetHere()` method to display flow instances (for the flow: `Order Flow`) that have taken longer than two hours to reach the node `Ship Order`.

When displaying the list of slow flow instances, the JSP provides a link to the `gen/flowInstance.jsp` page to enable the user to see further details for any of the instances:

```
<%
    String flowId = "";
    String flowName = "Order Flow";

    String nodeId = "";
    String nodeName = "Ship Order";

    final  long oneHour = 3600000;     // 1 hour = 1*60mins*60secs*1000mSecs
    long   durationLimit = 2 * oneHour;
%>

<!-- Get the Flow ID for the flow -->

<flow:flowOutlineList var="flowOutlineBean" nameFilter="<%= flowName %>" />
<%
    if (flowOutlineBean != null)
    {
        flowId = flowOutlineBean.getFlowId();
    }
%>
```

```
<!-- Get the node ID for the node -->

<flow:flow flowId="<%= flowId %>" var="flowBean" />
<flow:flowNodeList flowBean="<%= flowBean %>" var="flowNodeBean"
                                            nameFilter="<%= nodeName %>" />
<%
    if (flowNodeBean != null)
    {
        nodeId = flowNodeBean.getNodeId();
    }
%>

    <!-- Get a database connection -->
<%
    DBConnection dbConn = getDatabaseConnection(pageContext);

    // Call the notYetHere() method

    ResultSet results = notYetHere(dbConn, nodeId);

    // Set up some variable

    long   flowInstanceStartTime = 0;
    long   nowTime = System.currentTimeMillis(); // Sets the time to now!
    double duration;

    String flowInstanceId = "";
    String identifier = "";

    // Some helpers to display the duration in hours
    NumberFormat df = DecimalFormat.getNumberInstance();
    df.setMaximumFractionDigits(2);       // 2-digit precision

    %>

    <!-- Loop through displaying slow flow instances -->

    <table cellspacing="0" border="1">
        <tr>
            <th>Identifier</th>
            <th>Duration</th>
        </tr>
    <%
```

```
    while(results.next())
    {
        flowInstanceStartTime = results.getLong("starttimelongmillis");

        // Check how long this flow instance has been running for...
        duration = nowTime - flowInstanceStartTime;
        if (duration > durationLimit)
        {
            // Display this one to the screen

            flowInstanceId        = results.getString("flowinstance_id");
            identifier            = results.getString("identifier");
            %>
            <tr>
  <td>
     <a href=../gen/flowInstance.jsp?flowinstanceid=<%= flowInstanceId %> >
            <%= identifier %></a>
  </td>
  <td>
     <%= df.format(duration/oneHour) %> Hours
  </td>
            </tr>
            <%
        }
    }

%>
    </table>
<%
    // Close the database connection for this page
    dbConn.closeConection();
%>
```

Joining the Associated Data

When deriving statistics from the OVBPI database you often do not need to use anything more that the standard tables such as:

```
flows
flow_instance
nodes
node_instance
```

The schemas (layouts) for these tables are all listed in the *OVBPI System Administration Guide*.

You might need to produce statistics that require additional “business” data associated with each flow instances.

This “business” data is held in an associated (related) data table that is uniquely named for each deployed flow, and the layout of each of these data tables is unique for that data definition.

To get the name of the associated data table for a given flow, refer to Associated Data Table on page 166.

Once you have the name of the associated data table, you need to join it to the flow instance table. This is done by joining the following two fields together:

```
flow_instance.primary_entity_inst
<data table name>.id
```

Let’s consider an example:

> You have been asked to produce a customized dashboard that lists the instances of the `Order Flow` showing the individual order data.
>
> That is, produce a list of flow instances for the `Order Flow`, and for each flow instance, list the order details such as: order number, customer ID and order value (showing the actual currency type)

To do this, you need to do the following:

- Get the flow ID for the `Order Flow`
- Get the name of the associated data table for this flow
- Issue an SQL query that returns all the active order details. This query needs to:
  - — Join the flow instance table with the associated data table
  - — select the fields: `ordernumber`, `customerid`, `value` and `value_code`, for all active flow instances

  If it also selects the flow instance ID, then the resultant Web page can set up a link from each order to the `gen/flowInstance.jsp` page.
- Display the list of orders

Let’s consider this in two parts, looking firstly at the required SQL query, and then how to call this SQL from within your JSP.

The SQL Query

The SQL is written as a reusable method. You can place this method within the calling JSP, in its own JSP or within something like `gen/common.jsp`.

The method to return the order details is as follows:

```
<%!
    public java.sql.ResultSet getOrderDetails(DBConnection dbConn,
                                              String dataTable,
                                              String flowId)
    throws Exception
    {
        String sql;
        java.sql.PreparedStatement prepStatement;
        java.sql.ResultSet results;

        // This joins the flow_instance and assoc data table together.

        // It returns the flowinstance_id followed by columns specific
        // to the order data definition.
        // It only selects flows that are still active.

        sql = "select flow_instance.flowinstance_id, " +
                     dataTable + ".ordernumber, " +
                     dataTable + ".customerID, " +
                     dataTable + ".value, " +
                     dataTable + ".value_code " +
              " from flow_instance, " + dataTable +
  " where flow_instance.primary_entity_inst = " + dataTable + ".id" +
  " and   flow_instance.flow_id = ?" +
  " and   flow_instance.status != '" + Constants.FLOW_STATE_COMPLETED + "'";

        prepStatement = dbConn.prepareStatement(sql);
        prepStatement.setString(1, flowId);

        results = dbConn.executePreparedQuery(prepStatement);

        return results;
    }
%>
```

The Calling JSP

Let's look at a code extract from a JSP that uses this `getOrderDetails()` method to display the list of active orders (for the flow: `Order Flow`)

When it displays each order, it provides a link to the `gen/flowInstance.jsp` page to enable the user to see further details for any of the orders:

```
<%
    String flowId = "";
    String flowName = "Order Flow";
%>
<!-- Get the Flow ID for the flow -->

<flow:flowOutlineList var="flowOutlineBean" nameFilter="<%= flowName %>" />
<%
    if (flowOutlineBean != null)
    {
        flowId = flowOutlineBean.getFlowId();
    }
%>
<!-- Get a database connection -->

<%
    DBConnection dbConn = getDatabaseConnection(pageContext);
%>

<!-- Get the name of the associated data table for this flow -->

<jsp:useBean id="sqlBean" class="com.hp.ov.bia.views.DBSql" scope="page"/>
<%
    String dataTable = sqlBean.dbGetDataTableForFlowID(dbConn, flowId);
%>

<%
    // Call the SQL query

    ResultSet results = getOrderDetails(dbConn, dataTable, flowId);

    String flowInstanceId = "";
    String orderNumber = "";
    String custId = "";
    double value = 0;
    String valueCode = "";

    // These are here to help with the display of the value field
    NumberFormat df = DecimalFormat.getNumberInstance();
    df.setMaximumFractionDigits(2);       // 2-digit precision
    %>
```

```
    <h1>Current Orders</h1>
    <table cellspacing="0" border="1" >
        <tr>
            <th>Order Number</th>
            <th>Customer ID</th>
            <th>Value</th>
        </tr>
    <%
    // Now loop through the returned orders - displaying them as you go.
    while(results.next())
    {
        flowInstanceId  = results.getString("flowinstance_id");
        orderNumber     = results.getString("ordernumber");
        custId          = results.getString("customerID");
        value           = results.getDouble("value");
        valueCode       = results.getString("value_code");
        %>
        <tr>
    <td><a href=../gen/flowInstance.jsp?flowinstanceid=<%= flowInstanceId %>>
             <%= orderNumber %></a>
    </td>
            <td><%= custId %></td>
            <td><%= df.format(value) %> <%= valueCode %></td>
        </tr>
        <%
    }
%>
    </table>
<%
    // Close the database connection for this page
    dbConn.closeConection();
%>
```

Lab - Direct SQL

The purpose of this lab is to get you accessing the OVBPI database directly from within your JSPs.

Time to a Node

- Create a new JSP
- This JSP is to list flow instances for the flow `Order Flow`, but only show flow instances that have taken longer than 30 minutes to get to the node: `Ship Order`

Once this is working, change the JSP to report as follows:

- Report against the flow: `Call System`
- Only list flow instances that have taken longer than 10 minutes to get to the node: `Open`

Well done! You have reached the end of the lab.