HPSA Extension Pack

EP - Developer's Reference

Release V6.1

Legal Notices

Warranty.

Hewlett-Packard makes no warranty of any kind with regard to this manual, 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.

Trademark Notices.

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

Linux is a U.S. registered trademark of Linus Torvalds

Microsoft® is a U.S. registered trademark of Microsoft Corporation.

Red Hat® Enterprise Linux® is a registered trademark of Red Hat, Inc.

EnterpriseDB® is a registered trademark of EnterpriseDB.

Postgres Plus® Advanced Server is a registered trademark of EnterpriseDB.

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

UNIX® is a registered trademark of the Open Group.

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

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

Document id: EP-pd001302

Table of Contents

Support

Support for the HP Service Activator Extended Pack product is available on the following mailing list:

hpsa-support@hp.com

In This Guide

This guide explains how to use the Solution Container for developers.

Audience

The audience for this guide is the Solutions Integrator (SI). The SI has a combination of some or all of the following capabilities:

Understands and has a solid working knowledge of:

- UNIX® commands
- Windows® system administration

Understands networking concepts and language

Is able to program in Java™ and XML

Understands security issues

Understands the customer's problem domain

References

Conventions

The following typographical conventions are used in this guide.

Font	What the Font Represents	Example
Italic	Book or manual titles, and man page names	Refer to the *HP Service Activator — Workflows and the Workflow Manager* and the *Javadocs* man page for more information.
	Provides emphasis	You *must* follow these steps.
	Specifies a variable that you must supply when entering a command	Run the command: `InventoryBuilder` *<sourceFiles>*
	Parameters to a method	The *assigned_criteria* parameter returns an ACSE response.
Bold	New terms	The **distinguishing attribute** of this class...
`Computer`	Text and items on the computer screen	The system replies: `Press Enter`
	Command names	Use the `InventoryBuilder` command ...
	Method names	The `get_all_replies()` method does the following...
	File and directory names	Edit the file `$ACTIVATOR_ETC/config/mwfm.xml`
	Process names	Check to see if `mwfm` is running.
	Window/dialog box names	In the `Test and Track` dialog...
	XML tag references	Use the `<DBTable>` tag to...
`Computer Bold`	Text that you must type	At the prompt, type: **`ls -l`**
Keycap	Keyboard keys	Press **Return**.
[Button]	Buttons on the user interface	Click `[Delete]`. Click the `[Apply]` button.
Menu Items	A menu name followed by a colon (:) means that you select the menu, then the item. When the item is followed by an arrow (->), a cascading menu follows	Select Locate:Objects->by Comment.

Install Location Descriptors

The following names are used throughout this guide to define install locations.

Descriptor	What the Descriptor Represents
$ACTIVATOR_OPT	The install base location of Service Activator. The UNIX location is /opt/OV/ServiceActivator The Windows location is <drive>:\HP\OpenView\ServiceActivator\
$ACTIVATOR_ETC	The install location of specific Service Activator configuration files. The UNIX location is /etc/opt/OV/ServiceActivator The Windows location is <drive>:\HP\OpenView\ServiceActivator\etc\
$ACTIVATOR_VAR	The install location of specific Service Activator logging files. The UNIX location is /var/opt/OV/ServiceActivator The Windows location is <drive>:\HP\OpenView\ServiceActivator\var\
$ACTIVATOR_BIN	The install location of specific Service Activator binary files. The UNIX location is /opt/OV/ServiceActivator/bin The Windows location is <drive>:\HP\OpenView\ServiceActivator\bin\
$ACTIVATOR_THIRD_PARTY	The location for new Java components such as workflow nodes and modules. Third-party libraries can also be placed in this directory. The UNIX location is /opt/OV/ServiceActivator/3rd-party The Windows location is <drive>:\HP\OpenView\ServiceActivator\3rd-party\ Customized inventory files are stored in the following locations: UNIX: $ACTIVATOR_THIRD_PARTY/inventory Windows: $ACTIVATOR_THIRD_PARTY\inventory
$JBOSS_HOME	HOME The install location for JBoss. The UNIX location is /opt/HP/jboss The Windows location is <drive>:\HP\jboss
$JBOSS_DEPLOY	The install location of the Service Activator J2EE components. The UNIX location is

	/opt/HP/jboss/server/default/deploy The Windows location is <drive>:\HP\jboss\server\default\deploy
$ACTIVATOR_DB_USER	The database user name you define. Suggestion: ovactivator
$ACTIVATOR_SSH_USER	The Secure Shell user name you define. Suggestion: ovactusr
$SOSA_HOME	The install base location of SOSA. The default UNIX location is /opt/OV/Sosa The default Windows location is <drive>:\HP\OpenView\Sosa\
$SOSA_BIN	The install location of specific SOSA binary files. The default UNIX location is /opt/OV/Sosa/bin The default Windows location is <drive>:\HP\OpenView\Sosa\bin\
$SOSA_ETC	The install location of specific SOSA configuration files. The default UNIX location is /opt/OV/Sosa/config The default Windows location is <drive>:\HP\OpenView\Sosa\config\
$ECP_HOME	The install base location of Equipment Connections Pool. The default UNIX location is /opt/OV/ECP The default Windows location is <drive>:\HP\OpenView\ECP\
$ECP_BIN	The install location of specific Equipment Connections Pool binary files. The default UNIX location is /opt/OV/ECP/bin The default Windows location is <drive>:\HP\OpenView\ECP\bin\
$ECP_ETC	The install location of specific Equipment Connections Pool configuration files. The default UNIX location is /opt/OV/ECP/conf The default Windows location is <drive>:\HP\OpenView\ECP\conf\

1 Introduction

1.1 Purpose

This document is a manual for developers of applications and solutions designed for the Solution Container. Its purpose is to provide a wide explanation of the different features and characteristics involved in the development.

1.2 Document Scope

This document is focused on the different tools provided by the Solution Container and the designing criteria for new applications.

1.3 Definitions

1.3.1 Acronims

MWFM: Micro Work Flow Manager

HPSA: HP Service Activator

EP: Extension Pack

SC: Solution Container

WFLT: Work Flow Launcher and Tracker

CCWF: Concurrent Workflows Module

ECP: Equipment Connection Pool

SOSA: Service Order Smart Adapter

LM: Lock Manager

2 General Description

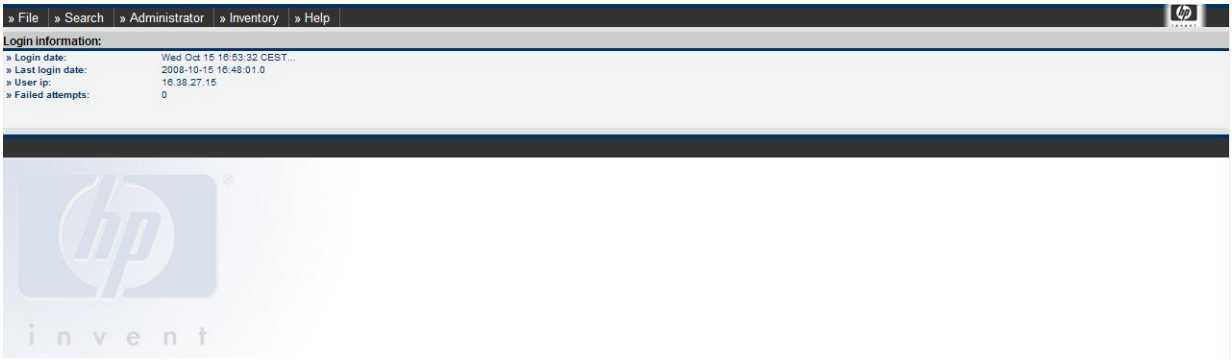
Solution Container, from now on SC, is an application framework for the development and deployment of user applications. Integrated into the HPSA, it provides mechanisms to integrate the applications into the activation system.

SC objectives are:

- To establish a common interface for all the user applications.
- To establish a main design which provides a clear separation between logical and presentation layers.
- To provide APIs and designing regulations for a lively and effective development of the user applications.
- To personalize the applications for each user accessing the tool.

2.1 Common Interface

This tool provides a common visual interface in which developers can deploy new user applications.



SC provides several APIs for developing and a designing guide which makes it easy developing and deploying new user applications. Integrated in the SC, all the applications possess the same look and feel based on configured menus.

2.2 Application design

Apart from a common interface, the user applications developed for the SC share the same designing criteria.

These applications are object-oriented. They manage every piece of data as an object and call it a "*component*". As objects, each of these components has properties and methods than are consulted and invoked in the application logic.

All the applications are based on the Struts framework, what allows setting a clear separation between logical and presentation layers. The following points provide a brief description of each of these layers in which a user application is divided.

2.2.1 The application view

The application view is the way a client can interact with the application's data. It defines what component, what properties of these components and what operations associated to these components will be accessible by every client. To simplify the process, the SC provides mechanisms for generating views from a single component information and defining operations associated to it.

As every application is integrated in the SC they share the same look and feel. The presentation layer is developed from this starting criterion using the different Tools provided by the SC.

2.2.2 The data model

The data model of an application consists in the definition of which components will be managed by the application. These components are mapped as Java Beans and usually are stored in a database.

2.2.3 Application logic

The application logic is implemented through Struts actions, which are invoked inside an application by selecting the different menu options available. SC set no criteria on the development of the application functionality, what allows opened application developments of very different natures.

2.3 Development tools

As it has been said before, SC provides several tools for developing and deploying user applications on an easy way.

These tools contain:

- A design guide for the application development.
- A maven library for the application structure definition.
- Controlling JSP files for the automatic views and status loading.
- JavaScript APIs for the automatic view generation.
- Generic Struts actions and forms for the searching performance and results presentation.
- Generic components for the integration with HPSA which makes easier the authentication process and the activation workflows launching and tracking.
- A visual tool for the user management.

2.4 User management

The SC implements its own user management, which allows setting permissions for accessing the different available applications.

There is a single entrance point to the SC from which, based on the user's account, the accessible applications and menus are loaded.

The users and menus structure will be explained in detail later.

2.5 Integration with HPSA

SC provides several mechanisms for the integration with HPSA, establishing an easy manner to access the HPSA and invoking activation tasks over the system.

There are also predefined applications integrated with HPSA that can be deployed into the SC, offering this way a high amount of Solutions based on this tool.

Further information about specific functionality related to HPSA can be found in further sections dedicated to the integration with HPSA.

2.6 Applications included in the SC

SC provides the next included user applications:

- The user management tool.
- The SOSA management tool.
- The SNMP management tool.
- The ECP management tool.
- A tool for Equipment configuration management.
- Access to the HPSA's Inventory window, which provides configurable database tree representations.

3 Solution Container

As it was explained in the previous point, every user application is deployed into the SC.

In this section there will be reviewed the main concepts involved in the SC development.

3.1 Single entrance point

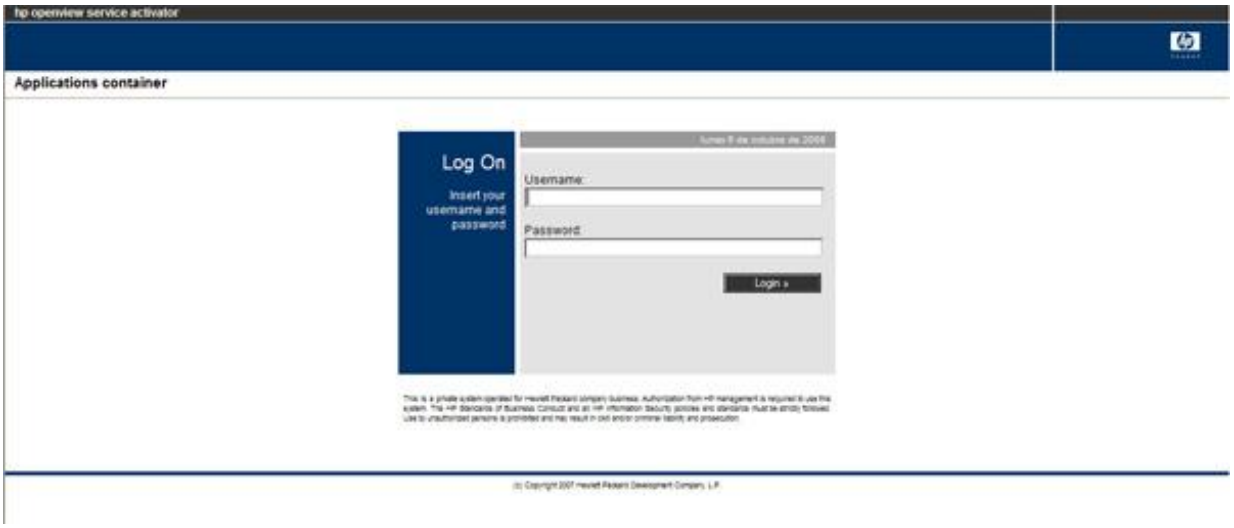
There is only a single entrance point to access the SC where the user must enter a valid username and password and, in base of his account, there will be loaded the menus which of his available applications.

The URL to access the SC is:

http://localhost:8080/ep/jsp/future-gui/hpac.jsp

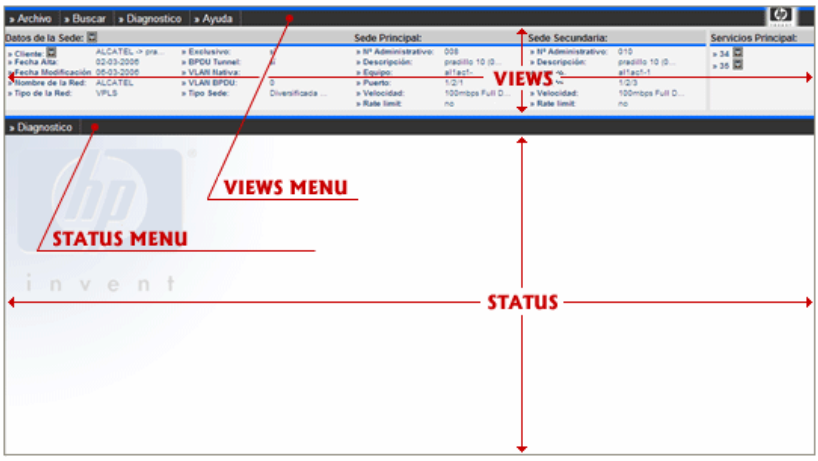
where *localhost* may be substituted by the server's IP.

The figure below shows the web page which is always shown to log on the SC.



3.2 SC structure

The SC has a well defined visual structure which divides the screen in several modules, each of them with a specific functionality.



The following sections describe the main characteristics of these modules.

3.3 The views menu

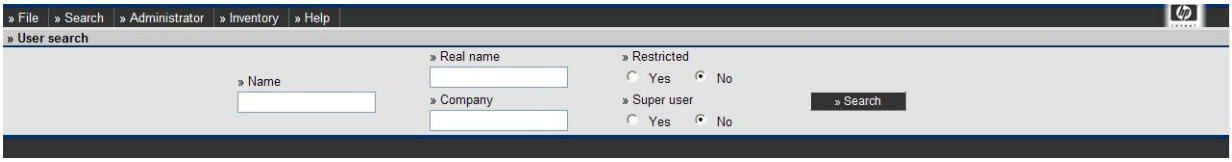
It provides accesses to the different user applications. It is loaded the first time the user enters the SC (just before the log on) and remains without changes while the user session lasts. Each user application includes one or more menus in the views menu bar.

As it can be seen in the example above, the user may access to the administration GUI (menus *Search* and *Administrator*) and the Inventory application (menu *Inventory*). The other menus (*File* and *Help*) belong to the SC.

3.4 The views

Once an application has been selected in the view menu this view frame can contain:

- Component information: it is the one shown in the previous figure. It presents the available data of the selected component. SC provides APIs and design guides for the quick development of this kind of views.



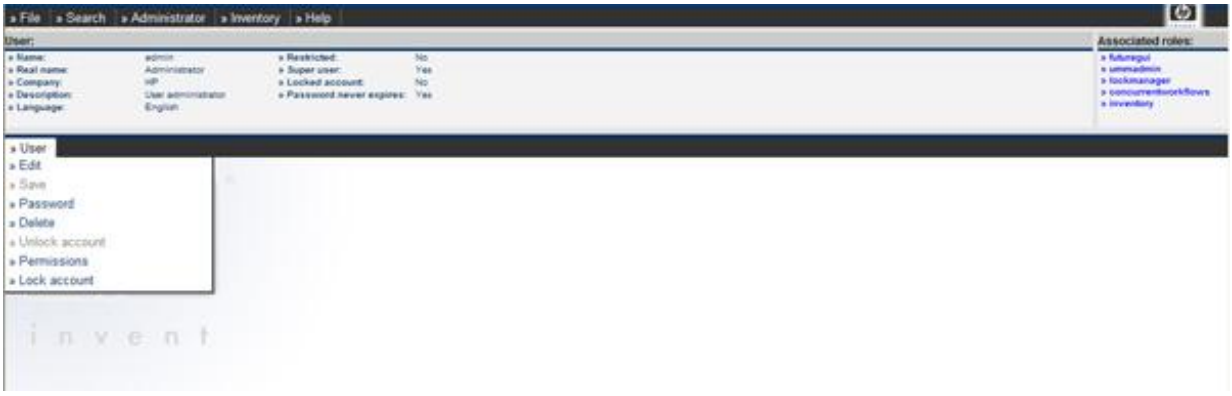
- Component search: it presents a form through which a component can be located. The form submitted becomes a query to get a list of components. Once the component has been selected the SC loads its information view. The SC provides the needed functionality to automate this task.

The data load in this frame will be referred from now on this document as "*actual view*".

3.5 The status menu

The status menu is associated to the actual view and contains the operations that can be executed over the selected component which data is being showed.

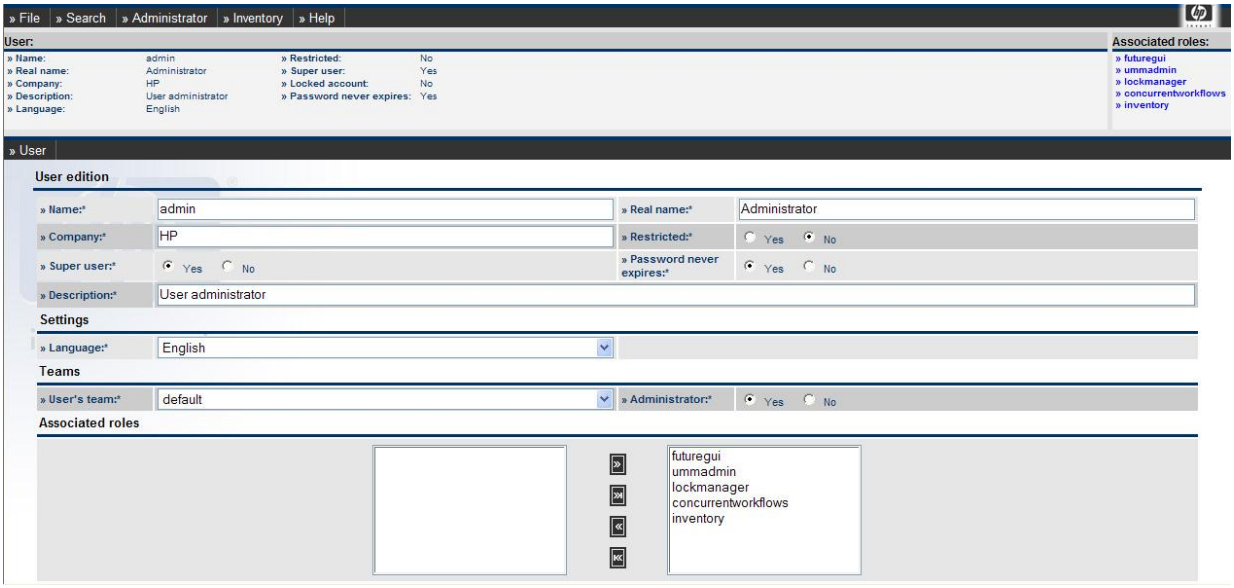
They are showed just below the view space and remains there while the user is working on the same component.



3.6 The status space

This space is placed below the status menu. Each time a task performed over a selected component is started up the result will be shown here.

In this space there will be launched the operations over a component and the user interaction can be carried on.



Each time the user interacts with a component the status is being modified. Checking this status the SC enables or disables the available component menus.

Imagine for instance an application performed for text file edition. The *Save file* option will only be enabled when the text file has been modified. Modifying the file means a change in the status, which

results in the enabling of the menu *Save file*. This principle is the same followed by the SC, which allows to define status that enable or disable certain menu options. SC provides the needed functionality for defining and managing the status changes.

4 Application development

The sections below will explain how to create a user application and deploy it into the SC.

The example that will be used employs the *MenuData* tool provided with EP. This tool allows the management of every database components of a given application, such as menus, roles, views, status and permissions through an XML-formatted file. The schema for the document is found in `$ACTIVATOR_ETC/config/menudata.xsd`. With this document it is possible to use the *MenuData* script, located in `$ACTIVATOR_BIN`, to record the solution from the XML file into Service Activator's static repository. Later in this document a more detailed explanation of this plugin can be found.

Along the example it will be created an easy application (Hello World!!!) which will guide the developer through the application implementation and will show the main performance of an application. The implementation process consists in defining a component, called HelloWorldComponent, with an associated operation that will present a welcome message. The next sections will explain the needed steps for a user application definition using this example.

NOTE: Along this example different APIs provided by the SC will be used. The objective of this chapter is not to describe in detail those APIs but to introduce the developers in their use. In latest sections they will be explained in detail.

4.1 Application model definition

For the *Hello World!!!* application the model will consist in an easy component, called *HelloWorldComponent*, which must be in charge of showing a welcome message on the screen. As it was explained on a previous section, the model definition is based in Java Beans.

```
public class HelloWorldComponent
{
  private String helloMessage;
   private String author;
   private String date;

  public HelloWorldComponent(String helloMessage) {
       setHelloMessage(helloMessage);
  }

  public String getAuthor() {
     return author;
   }
   public void setAuthor(String author) {
     this.author = author;
   }
   public String getDate() {
     return date;
   }
   public void setDate(String date) {
     this.date = date;
   }
   public String getHelloMessage() {
     return helloMessage;
   }
   public void setHelloMessage(String helloMessage) {
     this.helloMessage = helloMessage;
   }
```

```
}
```

4.2 Application definition

The developer may define a new application through the MenuData tool, provided with the installation. Thus:

```
<Solution>

  <Applications>
    <Application>
      <Name>HelloWorldApplication</Name>
      <Description>Hello World Application</Description>
      <Enabled>true</Enabled>
      <To-Roles>
        <To-Role>futuregui</To-Role>
      </To-Roles>
    </Application>
  </Applications>

</Solution>
```

4.3 Main menus definition

Once the application has been defined, the first step consists on defining the main menus that will be present in the view menu bar. As it was said before, each application should include one or more menus in this view menu bar.

In the SC every menu has to be associated to a view. All the menus which must appear in the view menu bar have to be associated to the *root* view, which is only used to set the menus of this bar.

```
<Solution>

  << Application HelloWorldApplication definition >>

  <Menus>
    <Menu>
      <Name>HelloWorld</Name>
      <Description>Hello World</Description>
      <To-Application>HelloWorldApplication</To-Application>
      <Key>menu.principal</Key>
      <Bundle>com/hp/spain/futuregui/HelloWorldApplicationResources</Bundle>
      <To-Views>
        <To-View position="100">root</To-View>
      </To-Views>
      <To-Roles>
        <To-Role>futuregui</To-Role>
      </To-Roles>
    </Menu>
  </Menus>

</Solution>
```

This code includes a menu for our application in the views bar. As all the messages in the application are localized so when defining the menu a properties file and a key to name the menu are required.

The position of the menu in the views bar is set with the 'position' attribute. The order is calculated from left to right.

Next an example of another menu depending of the “HelloWorld” created in the last example:

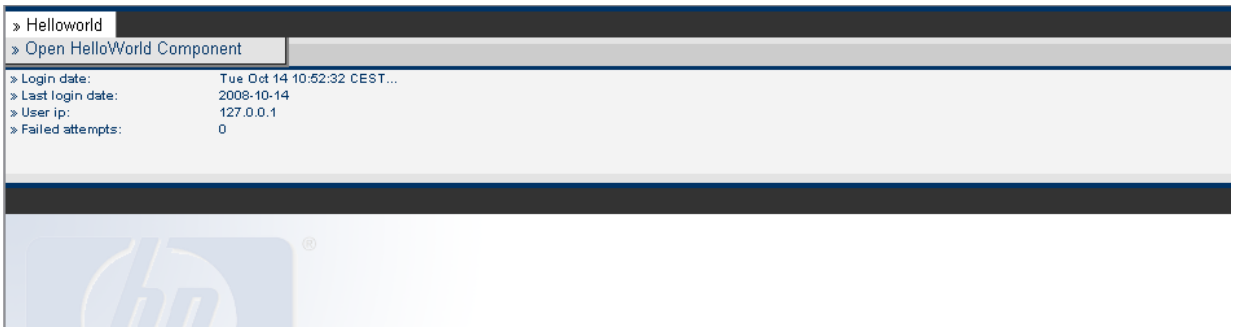
```
<Solution>

  << Application HelloWorldApplication definition >>

  <Menus>
    << Menu HelloWorld definition >>
    <Menu>
      <Name>OpenHelloComponent</Name>
      <Description>Open Hello Component</Description>
      <To-Application>HelloWorldApplication</To-Application>
      <Key>menu.open</Key>
      <Bundle>com/hp/spain/futuregui/HelloWorldApplicationResources</Bundle>
      <Parent>HelloWorld</Parent>
      <Action>OpenHelloComponentAction.do</Action>
      <Location>default</Location>
      <To-Views>
        <To-View position="100">root</To-View>
      </To-Views>
      <To-Roles>
        <To-Role>futuregui</To-Role>
      </To-Roles>
    </Menu>
  </Menus>

</Solution>
```

Don't forget to add the key in the properties file ‘HelloWorldApplicationResources’:

```
menu.principal = Helloworld
menu.open = Open HelloWorld Component
```

As can be seen, the definition is the same has the one seen before, but with a new tag called “parent” that sets the father menu.

At this point, the application includes a new menu in the main bar:



In this example, when clicking in the menu an action is desired to be executed. As has been seen, the functionality of the *views menu* is to load views. This view has to be loaded by a Struts action that is defined with the “action” attribute.

In the example, *OpenHelloComponentAction.do*, will be the action that will return our “HeloWorldComponent” and show it in the views frame.

4.4 Actions of the main menus

This section will focus on the component location and presentation actions. The actions associated to the *root* view must satisfy these two requirements:

- Provide the necessary functionality for creating or locating a component.
- Open a component view.

For the *Hello World!!!* example, the code for the *OpenHelloComponentAction* must be:

```
public class OpenHelloWorldComponentAction
extends Action
implements HelloWorldConstants
{

  public ActionForward execute(     ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
  throws IOException, ServletException {
     String target;
     HelloWorldComponent helloWorldComponent;

       try {
       helloWorldComponent = new HelloWorldComponent("Hello World!!!");
       helloWorldComponent.setAuthor("Javier");
       helloWorldComponent.setDate(new Date().toString());
       if (Context.getInstance().containsKey(MYCOMPONENT)) {
         Context.getInstance().remove(MYCOMPONENT);
       }
       Context.getInstance().add(MYCOMPONENT, helloWorldComponent);
       FGLogger.logInfo("HelloWorldComponent loaded");
       target = SUCCESS;
       }
       catch(Exception e) {
             e.printStackTrace();
       target = FAILURE;
       }
     return mapping.findForward(target);
  }
}
```

This is the code used to create the component. It creates an instance of it and stores it into the application context.

Then, it is necessary to call a view where the component information can be displayed. This is done in the *struts-config.xml* file where the Struts actions are mapped and their possible exits are set:

```
<action
      path="/OpenHelloComponentAction"
      type="com.hp.spain.OpenHelloComponentAction" cope="request">
   <forward
      name="success"
      path="/jsp/future-gui/index.jsp?viewName=HelloComponentView&
            fjsp=/jsp/helloworld/initial.jsp "/>
   <forward
      name="failure"
      path="/jsp/future-gui/index.jsp?fjsp=/jsp/error.jsp"/>
</action>
```

One of the utilities provided by the SC consists in several controlling JSP files used to manage the loaded views and their different menus.

The *index.jsp* file is the one employed to load the views and get their associated menus. As it can be seen in the code above, it is the JSP file called as the result for the action.

The *index.jsp* file receives two possible parameters:

- *viewName*: contains the view name which is going to be loaded in the view frame, if any. See the next section to learn about the view creation.
- *fjsp*: contains the URL of the JSP file which is going to be initially loaded in the status space just before the view has been loaded.

Either the view or the JSP status file are issues that will be explained in retail in further sections.

4.5 View Definition

The code below shows how to include a view into the SC.

```
<Solution>

  << Application HelloWorldApplication definition >>

  <Views>
    <View>
      <Name>HelloComponentView</Name>
      <Description>Hello Component View</Description>
      <Url>/jsp/future-gui/hello-world/HelloComponentView.jsp</Url>
    </View>
  </Views>

  << Menus definition >>

</Solution>
```

The main attribute called "jsp" is the view JSP file target . As it was explained in a previous chapter, SC provides specific APIs for view development which consist in JavaScript objects that compose on an easy way any presentation view JSP file. Anyway, SC allows to include any kind of JSP file to show information. The only requirement is that the JSP file must be loaded into the view space and, thus, there is a limitation on the available space.

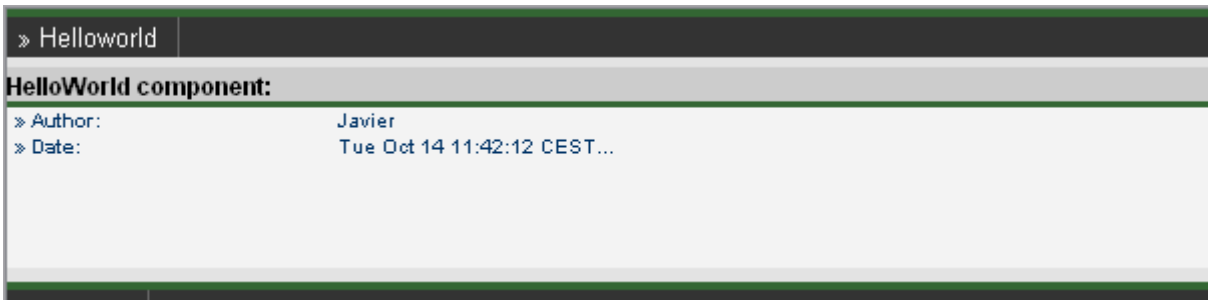
```
<%@ taglib uri = "/tags/struts-bean" prefix="bean" %>

<%@ page import =
   "com.hp.spain.example.helloworld.HelloWorldComponent" %>
<%@ page import =
   "com.hp.spain.example.helloworld.struts.HelloWorldConstants" %>
<%@ page import = "com.hp.spain.hputils.framework.Context" %>

<%
HelloWorldComponent myComponent =
  (HelloWorldComponent)Context.getInstance().get(HelloWorldConstants.MYCOMPON
ENT);
%>

<script>

var mmi = new MainMenuInfo();
```

```
mmi.addTitle(
   "<bean:message bundle="HelloWorldAR" key="helloworld.title" />",
   null);
mmi.addAttribute(
   "<bean:message bundle="HelloWorldAR" key="helloworld.author" />",
   <%= myComponent.getAuthor() %>,
   0,
   0,
   null);
mmi.addAttribute(
   "<bean:message bundle="HelloWorldAR" key="helloworld.date" />",
   <%= myComponent.getDate() %>,
   0,
   1,
   null);

new MenuInfoWriter(mmi, null, null).write();

</script>
```

Don't forget to add the key in the properties file:

```
menu.principal = Helloworld
menu.open = Open HelloWorld Component
helloworld.title = HelloWorld component
helloworld.author = Author
helloworld.date = Date
```

Once this step has been completed, the application will be able to display the view, when selecting the 'Open Hello Component' menu:



4.6 Status menu definition

The status menus are associated to the actual view and are displayed just below the view JSP file. They represent the possible operations that can be performed over the selected component which information is being showed in the view. The next code explains how must they be defined:

```
<Solution>

  << Application HelloWorldApplication definition >>

  << View HelloComponentView definition >>

  <Menus>
    << Menu HelloWorld definition >>
    << Menu OpenHelloComponent definition >>
    <Menu>
      <Name>HelloComponentActions</Name>
      <Description>Hello Component Actions</Description>
```

```
      <To-Application>HelloWorldApplication</To-Application>
      <Key>menu.actions</Key>
      <Bundle>com/hp/spain/futuregui/HelloWorldApplicationResources</Bundle>
      <To-Views>
        <To-View position="100">HelloComponentView</To-View>
      </To-Views>
      <To-States>
        <To-State>default</To-State>
      </To-States>
      <To-Roles>
        <To-Role>futuregui</To-Role>
      </To-Roles>
    </Menu>
    <Menu>
      <Name>SayHello</Name>
      <Description>Say Hello</Description>
      <To-Application>HelloWorldApplication</To-Application>
      <Key>menu.sayHello</Key>
      <Bundle>com/hp/spain/futuregui/HelloWorldApplicationResources</Bundle>
      <Parent>HelloComponentActions</Parent>
      <Action>SayHelloAction.do</Action>
      <Location>default</Location>
      <To-Views>
        <To-View position="100">HelloComponentView</To-View>
      </To-Views>
      <To-States>
        <To-State>default</To-State>
      </To-States>
      <To-Roles>
        <To-Role>futuregui</To-Role>
      </To-Roles>
    </Menu>
  </Menus>

</Solution>
```

Every internationalized text must be placed in a properties file which has to be contained into the application JAR file, assuring this way that it will be accessible in the classpath when the SC is running.

At this point the content of this properties file should be something like this:

```
menu.principal = Helloworld
menu.open = Open HelloWorld Component
menu.actions = Actions
menu.sayHello = Say Hello

helloworld.title = HelloWorld component
helloworld.author = Author
helloworld.date = Date
```

As it can be seen, the status menu definition is very similar to the view menu one, but instead of associating them to the *root* view it has to be done to the actual view.

4.7 Application status definition

Each application can remain in a determined status in base of the operation which is being currently performed. This status sets at each moment which status menus have to be enabled.

In the *Hello World!!!* example there must be defined a single status, called *default*, that will enable every application status menu. The code needed for this can be seen below.

```
<Solution>

  << Application HelloWorldApplication definition >>

  << View HelloComponentView definition >>

  <States>
    <State>
      <Name>default</Name>
      <Description>Default state</Description>
      <To-Application>HelloWorldApplication</To-Application>
    </State>
  </States>

  << Menus definition >>
  << Note that the last two menus were associated to the state default >>

</Solution>
```

Since the two status menus have been associated to the application status, changing to this status will cause the enabling of both of them.

In the next section the necessary steps to load a given status will be explained.

4.8 Application status management

Defining a status does not mean it will be loaded just when the view is also loaded. It is the status JSP files responsibility to set the proper status and manage them. For that, the developers may employ a JavaScript API which allows to dynamically set the status inside a JSP file in an easy manner.

For instance, take the code of the initial status JSP file associated to the example view. The invocation of such JSP file was:

/jsp/future-gui/index.jsp?view=HelloComponentView&fjsp=/jsp/initial.jsp

As it can be seen, when the view is opened, far away than the view name, it was necessary to define the initial JSP file that had to be loaded into the status space.

JSP files loaded into the status space must follow the next criteria:

- Set the application status, if any. If this is not done, the application will remain in the last status defined before. Otherwise, if there was no status defined before the status menus displayed inside the status menu bar will appear disabled.
- They are in charge of displaying the information to the user, even presenting an action result or requesting data in a given form.
- They must pay attention to the events generated when a status menu is clicked.

Lets explain each of these points through the initial JSP file of the example application, *initial.jsp*.

```
<%
String useRandomColor =
   (String) session.getAttribute(
      com.hp.ov.activator.mwfm.futuregui.servlet.Constants.USE_RANDOM_COLOR
   );
String mainColor =
```

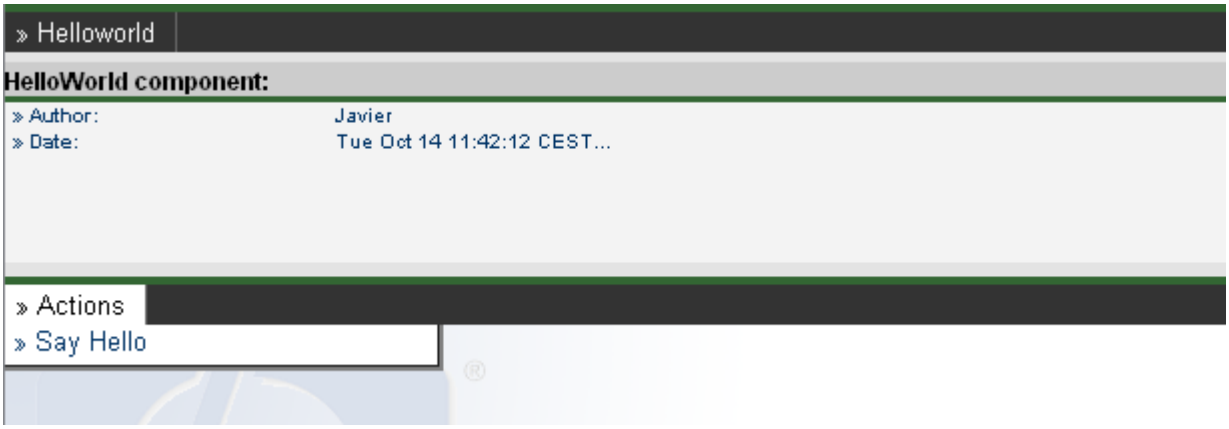
```
    (String) session.getAttribute(
       com.hp.ov.activator.mwfm.futuregui.servlet.Constants.APP_MAIN_COLOR
    );
%>

<script>
function changeStatus(clickedMenuName, clickedMenuAction) {
  window.location.href = clickedMenuAction;
}
</script>

<html>

<head>
  <link
       rel="stylesheet"
       href="/activator/css/future-gui/estilos<%=
(useRandomColor.equals(com.hp.spain.futuregui.Constants.TRUE)) ? mainColor :
""%>.css">
  <link
       rel="stylesheet"
       href="/activator/css/future-gui/subestilos<%=
(useRandomColor.equals(com.hp.spain.futuregui.Constants.TRUE)) ? mainColor :
""%>.css">
</head>

<body
<%
if (useRandomColor.equals(com.hp.spain.futuregui.Constants.TRUE)) {
%>
  background="/activator/images/future-gui/fondo.gif"
<%
}
%>
  onload="window.parent.loadParticularMenu();window.parent.loadStatusParticul
arMenu('HelloWorldDefaultStatus');">

</body>

</html>
```

The selected code shows how to invoke the JavaScript API for the status Management. The most important items are:

- [`window.parent.loadParticularMenu()`] This JavaScript function displays the status menus. When a view is showed, the SC loads its associated status menus, but they will not be displayed if it is not requested using this function.
- [`window.parent.loadStatusParticularMenu('HelloWorldDefaultStatus')`] This JavaScript function sets the application status, and so, enables the menus associated to that status. As it was seen before, the default status will enable all the view menus. Even this function or the previous one must be invoked when the status JSP file has been loaded (*onLoad* event).
- The own developer's code. In this case, an empty page is shown with a background image (*fondo.gif*).
- [`function changeStatus(clickedMenuName, clickedMenuAction)`] This JavaScript function is called every time a status menu (with an associated action) is clicked, and must be implemented in each status JSP file. It receives the name of the clicked status menu and the URL of

the associated action. By default, the action associated to the status menu is not invoked automatically when the menu is clicked; it is the developers responsibility to perform that invocation along the JavaScript code of the *changeStatus* function. This allows, for example, to insert different tasks before the action is executed, such as errors management or appending arguments to the action. In the example the status menu action is invoked without any previous task.

At this point, the application should be like the one in the image below:



4.9 Status menu actions

This section will focus on developing actions associated to a component, which are invoked as it has been explained previously. These actions must satisfy the next requirements:

- Contain the needed functionality for executing the component task.

The next code belongs to the *SayHelloAction*, defined in a section before.

```
public class SayHelloAction
extends Action
implements HelloWorldConstants
{

  public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
  throws IOException, ServletException {
    String target;
    HelloWorldComponent helloWorldComponent;

    try {
      helloWorldComponent =
        (HelloWorldComponent) Context.getInstance().get(MYCOMPONENT);
      request.setAttribute(
        MYCOMPONENTMESSAGE,
        helloWorldComponent.getHelloMessage());
      target = SUCCESS;
    }
    catch(Exception e) {
      e.printStackTrace();
      target = FAILURE;
    }
```

```
    return mapping.findForward(target);
  }
}
```

This action gets the component from the Application Context and calls the *getMessage()* method to obtain the *Hello World!!!* text. Then, this String is stored as an attribute into the request. The *struts-config.xml* file contains the mapping to the JSP file where the results must be displayed.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>

  <action-mappings>
      <action path="/OpenHelloWorldComponentAction"
      type="com.hp.spain.example.helloworld.struts.action.
                  OpenHelloWorldComponentAction"
      scope="request">
            <forward
               name="success"
               path="/jsp/future-gui/index.jsp?viewName=HelloComponentView
                        &amp;fjsp=/jsp/helloworld/initial.jsp"/>


            <forward
                  name="failure"
                  path="/jsp/future-gui/index.jsp?
                        fjsp=/jsp/helloworld/componentError.jsp"/>
      </action>

      <action path="/SayHelloAction"
                  type="com.hp.spain.example.helloworld.struts.action.
                        SayHelloAction"
                  scope="request">
            <forward
                  name="success"
                  path="/jsp/helloworld/showHelloMessage.jsp"/>
            <forward
                  name="failure"
                  path="/jsp/helloworld/execError.jsp"/>
      </action>

  </action-mappings>

  <message-resources
      parameter="com.hp.spain.example.helloworld.struts.
                              HelloWorldApplicationResources"
                  key="HelloWorldAR"/>

</struts-config>>
```

4.10 Action result

The result of any action is displayed in a JSP file loaded into te status space. These JSP files, far away than displaying the result, must implement the status Management defined in a previous chapter.

In the example, the *ShowHelloMessage.jsp* will display the *Hello World!!!* text. This JSP's code should be:

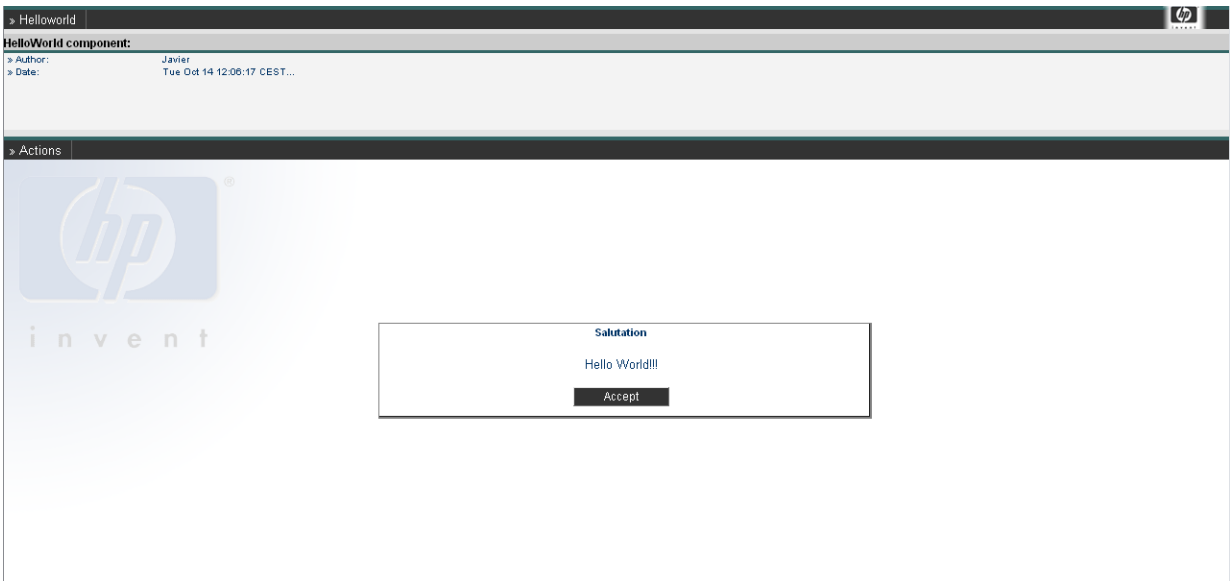
```
<%@ taglib uri = "/tags/struts-bean" prefix="bean" %>

<%@ page import =
"com.hp.spain.example.helloworld.struts.HelloWorldConstants" %>

<%
String useRandomColor =
   (String) session.getAttribute(
      com.hp.ov.activator.mwfm.futuregui.servlet.Constants.USE_RANDOM_COLOR
   );
String mainColor =
   (String) session.getAttribute(
      com.hp.ov.activator.mwfm.futuregui.servlet.Constants.APP_MAIN_COLOR
   );
%>

<script>
function changeStatus(clickedMenuName, clickedMenuAction) {
  window.location.href = clickedMenuAction;
}
</script>

<html>

<head>
  <link
       rel="stylesheet"
       href="/activator/css/future-gui/estilos<%=
(useRandomColor.equals(com.hp.spain.futuregui.Constants.TRUE)) ? mainColor :
""%>.css">
  <link
       rel="stylesheet"
       href="/activator/css/future-gui/subestilos<%=
(useRandomColor.equals(com.hp.spain.futuregui.Constants.TRUE)) ? mainColor :
""%>.css">
</head>

<body
<%
if (useRandomColor.equals(com.hp.spain.futuregui.Constants.TRUE)) {
%>
  background="/activator/images/future-gui/fondo.gif"
<%
}
%>
  onload="window.parent.loadParticularMenu();window.parent.loadStatusParticul
arMenu('HelloWorldDefaultStatus');">

<script>
var fa = new FutureAlert(
  "<bean:message bundle="HelloWorldAR" key="helloworld.salutation.title" />",
  "<%= helloMessage %>");
  fa.setBounds(500, 100);
  fa.setButtonText("<bean:message bundle="HelloWorldAR" key="button.accept"
                   />");
  fa.show();
</script>
```

```
</body>

</html>
```

At this point the content of this properties file should be something like this:

```
menu.principal = Helloworld
menu.open = Open HelloWorld Component
menu.actions = Actions
menu.sayHello = Say Hello

helloworld.title = HelloWorld component
helloworld.author = Author
helloworld.date = Date

helloworld.salutation.title = Salutation
button.accept = Accept
```

Once all these steps have been completed, the user application would show the hello message, after selecting the 'Say Hello' menu:

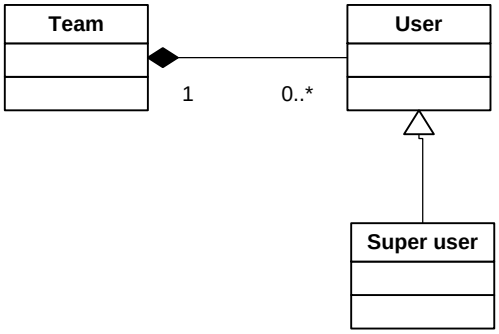


5 User structure

The SC provides its own user Management. The sections below describe each user structure component.

Further information about user management can be found in the document *HPSA Extension Pack – Solution Container – User reference*.

User structure:



5.1 Users

They define the ones allowed to access the SC. This kind of users would have restricted access to the different applications inside the SC.

5.2 User teams

A team sets a group of users with the same (or similar, at least) rights. There is an administrator user (and only one) for each team, and he will be the only one (apart from super users) allowed to manage that team. Administrators can create, update or remove users belonging to their group, and manage the permissions over them.

The Teams usage is optional and it is only available if the DatabaseAvancedAuthModule is configured as the authentication module. For further information see the HPSA documentation.

5.3 Super user

Super users have full administration privileges over any user, group, application or any other element of the SC.

5.4 System user

The system user is unique for the whole system. He can't be deleted or updated. Apart from this, the system user is treated as a super user.

6 User creation

The code below shows the contents of the XML file which should be employed by the *UMMData* tool (see HP Service Activator's documentation) to create a user and a team:

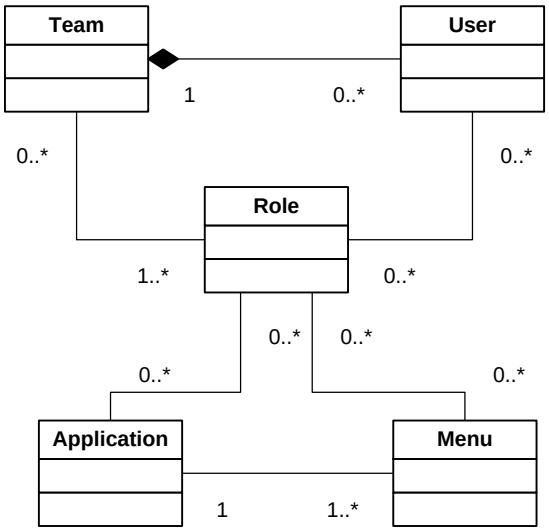
```
<UMM>

  <TeamInfo>
    <Name>TestTeam</Name>
    <Description>Team for testing purposes</Description>
  </TeamInfo>

  <User>
    <Name>TestUser</Name>
    <Password>pass4ut1</Password>
    <RealName>Testing user</RealName>
    <Description>User for testing purposes</Description>
    <CompanyName>HP</CompanyName>
    <Team>TestTeam</Team>
  </User>

</UMM>
```

Once the user is defined, it is necessary to set his permissions to access the different applications. The next sections describe the permissions structure of the SC and some helpful examples.

7 Permissions structure

Permissions are established through roles. The roles define profiles for accessing the SC and set the applications and their menus accessible for the user. Roles are assigned to teams and, for each team, there can be associated one or more users who belong to that role. This means that a user can only be associated to roles which have been assigned to his team before.

Permissions structure:



The relationships between roles and applications determine the applications accessible for those roles. There is also another relationship between roles and menus which determines the menus of an accessible application will be displayed after the user logs on into the SC. That allows not only to assign applications to users but to offer to the user different functionality inside each application.

8 Assigning permissions

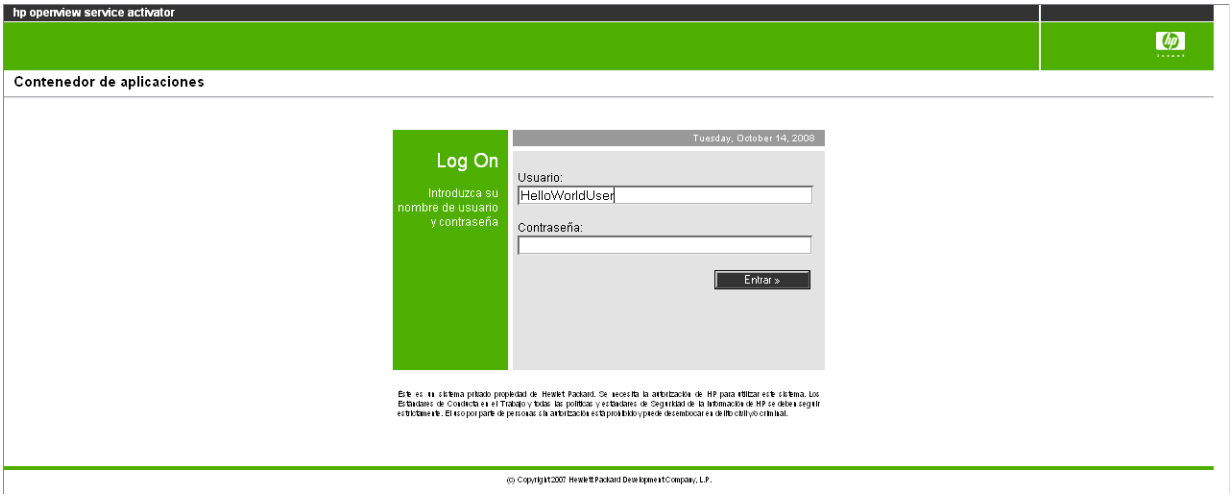
Let's get over the example again to show how to create a role with which the user can access the *Hello World!!!* application.

The code below shows the contents of the XML file which should be employed by the *UMMData* tool (see HP Service Activator's documentation) to create a role:

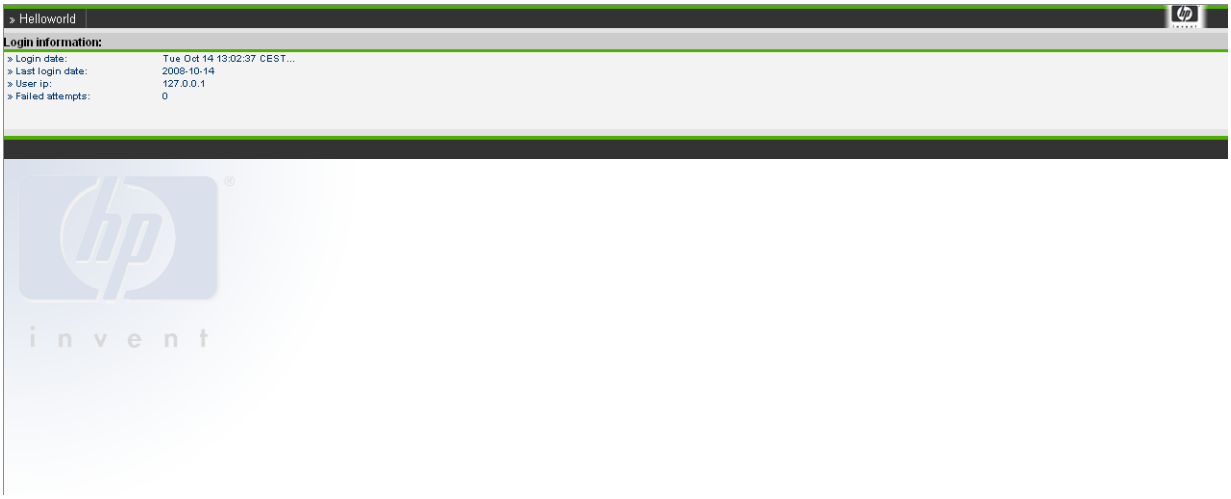
```
<UMM>

  <Role>
    <Name>TestRole</Name>
    <Description>Role for testing purposes</Description>
  </Role>

  <TeamRoleAssignment>
    <TeamName>TestTeam</TeamName>
    <RoleName>TestRole</RoleName>
  </TeamRoleAssignment>

  <UserRoleAssignment>
    <UserName>TestUser</UserName>
    <RoleName>TestRole</RoleName>
  </UserRoleAssignment>

</UMM>
```

Applications and menus are associated to roles in their own definition. See the section related to *Application defelopment* for more information.

At this point, the recently crated user will be able to access the *Hello World!!!* application.



The container loads the applications that user has access to. In this case, the 'Hello World Application':


» Helloworld
Login information:
» Login date:
Tue Oct 14 13:02:37 CEST...
» Last login date:
2008-10-14
» User ip:
127.0.0.1
» Failed attempts:
0

9 Action Audit

SC provides a RMI service with methods for auditing actions. This URL is stored in the SC's Context (see the section related to the Context for further information). The key needed to obtain the URL from the Context is a constant defined in the *com.hp.spain.futuregui.login.LoginConstants* interface.

Audited actions can be managed by super users using the administration GUI provided with the EP.

The next example shows how to audit an action in a java class:

```
import java.rmi.Naming;
import com.hp.spain.futuregui.login.LoginConstants;
import com.hp.spain.futuregui.users.rmi.def.SPIUserManagementRMIDef;
import com.hp.spain.hputils.framework.Context;


SPIUserManagementRMIDef userManager = null;
try {
   userManager =
      (SPIUserManagementRMIDef) Naming.lookup(
         (String) Context.getInstance().
                  get(LoginConstants.SPI_USER_MANAGER_RMI_URL)));
   userManaget.auditAction(…);
} catch (Exception e) {
   Throw new Exception ();
}
```

10 Integration with HPSA

10.1 Workflow Launcher

The EP provides mechanisms to manage the available workflows, which can be started up using a given API and tracked using a given GUI. It is also possible to interact with those workflows that need some extra information while they are running.

This document will explain the way to start up, track and interact with the workflows as it has to be done in the EP.

Workflows can be started up through SOSA, so this document will explain the way to make it. See the SOSA documentation for more information about SOSA.

Tracking of children workflows can be also done. There are two different ways for making this: using the database or using the CCWF. Both are supported by the WFLT and will be explained in further sections.

10.1.1 What is the WFLT?

The WFLT is a tool provided with the EP to make easy and possible the start up of workflows on any specified MWFM and track them.
The workflow's launching and tracking is performed using Struts actions which will execute the different tasks required in the process.

10.1.2 What is SOSA?

Service Order Smart Adapter (SOSA) is a flexible adapter to manage the influx of Service Orders, which are aimed at the transactional activation engine called Service Activator. In this way, SOSA provides additional features for the treatment of these requests compared to a traditional system.

10.1.3 Starting up a workflow

A workflow can be launched either from the Application Environment or from the Inventory. The launching is executed calling Struts action *WFLTAction.do*. This is the action used to start a workflow or to start tracking a workflow which has been already started up. This action has to be invoked with the necessary parameters or attributes which will be discussed later, but it is important to say that every parameter explained in this document can be retrieved either as a request parameter or an attribute, and this makes possible the invocation of the *WFLTAction.do* either from a JSP file or from another Struts action. If the workflow start up with SOSA is intended, it will be expressed in the parameters used with the action.

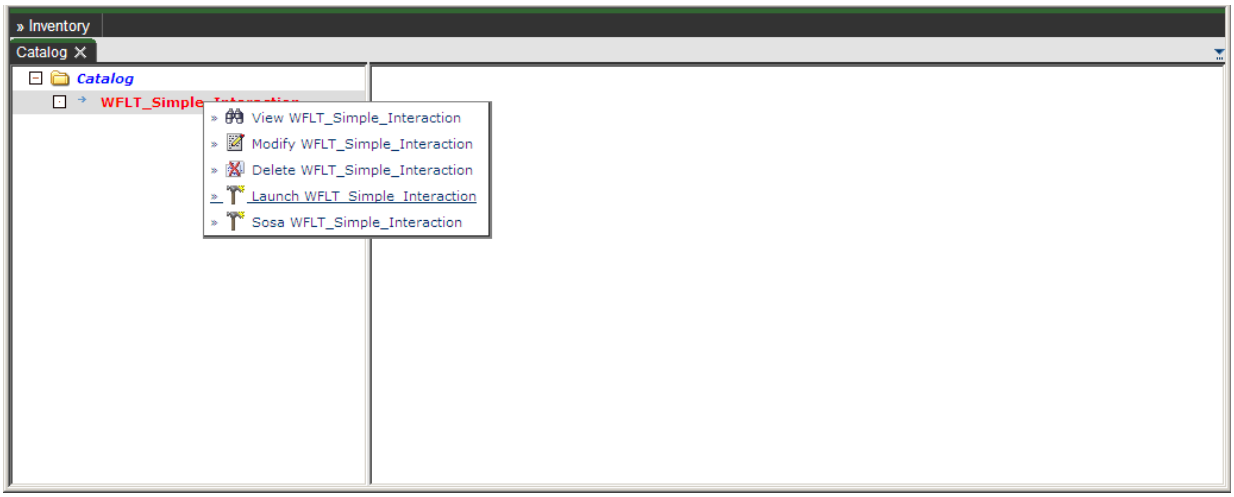
The parameters/attributes accepted by *WFLTAction.do* to start up a workflow are:

- *__wfname*: name of the workflow to be started up. This parameter is mandatory for starting up a workflow, but it is not used when the WFLT is only invoked for tracking an already started up workflow.
- *__wfmwfmname*: name of the MWFM engine where the given workflow must be started up. The names of the different MWFM engines are configured in the *auth.properties* file (see the *Configuration* section for more information). If no MWFM engine name is specified, the default name specified in the *auth.properties* file will be taken.

Ex:

This example shows how to launch a workflow from the inventory defining an operation. The following example shows its appearance.

```
<Operation>
  <Name Bundle="com/hp/spain/wflauncher test/ApplicationResources">
       launch.wf
  </Name>
  <Image>newtool.gif</Image>
  <Object>WfLauncherTest</Object>
  <OperationType>Lanzamiento</OperationType>
  <Action>
       <Page>/activator/WFLTAction.do</Page>
       <Param>
             <Name>__wfname</Name>
             <Value>WfLauncherTest.Name</Value>
       </Param>
       <Param>
             <Name>__wfmwfmname</Name>
             <Value>constant:localmwfm</Value>
       </Param>
       <Param>
             <Name>__wfsosacheck</Name>
             <Value>constant:true</Value>
       </Param>
  </Action>
</Operation>
```



After starting up a workflow, *WFLTAction.do* will be followed by the tracking process.

10.1.3.1 Case packet values specification

Initial values for the workflow's case packet can be specified when invoking the *WFLTAction.do* and a *HashMap* will be automatically composed with them and sent to the workflow to start it up properly. It is possible to specify each case packet entry and value or to specify a *HashMap* already filled with the needed values.

The parameters/attributes accepted by the *WFLTAction.do* for the case packet composition are:

- *__wfCP*: a *HashMap* already filled that will be sent as the initial case packet for the workflow start up. If there are also specified some other parameters/attributes for the case packet, this *HashMap* will be extended with the new retrieved entries. Note that this *__wfCP* attribute can never be received as a request parameter, because it is not possible to receive a Java object that

way. It must be specified as a request attribute, and this means that in this case the invocation of the *WFLTAction.do* has to be made from a previous Struts action, never from a JSP file.

- *wfvar__<<key_name>>*: indicates a new entry for the initial case packet. In the HashMap will be inserted a new key <<key_name>> associated to the specified value of this parameter/attribute. Note that if this is received as a request parameter, it will be inserted into the case packet as a String. If it is received as a request attribute, it will be inserted as it is received and any king of Object can be inserted.

Ex.:
From a JSP file, a workflow called *InsertEquipment* is started up indicating two values for the case packet: one called *name*, another called *model* and a third one called *version*.

```
WFLTAction.do?__wfname=InsertEquipment&wfvar__name=EQ0&wfvar__model=HP&wfvar
__version=2.4
```

This will generate a *HashMap* with three entries:

- name: EQ0
- model: HP
- version: 2.4

10.1.3.2 Backwards compatibility

There is a Struts action called *DeprecatedWFLTAction.do* which provides some extra functionality that is actually deprecated and should be never more used, but is still supported here to maintain the backwards compatibility. This action is followed by the *WFLTAction.do*, so the same parameters explained for it are accepted by this one.
This action is used to compose in an automatic way either *HashMaps* or Arrays of Strings which must be inserted into the case packet.
The *HashMap* composition is made getting from the request parameters (and only parameters, never attributes) those starting by *wfvar__hashmapX*, where X is a number beginning from 0. This way, every request parameter name starting by *wfvar__hashmap0* will be inserted in a *HashMap*, those starting by *wfvar__hashmap1* in another one, and so on.
Ex.:
The next invocation generates two *HashMaps*, one with the entries *location* and *country*, and the other with the entries *name*, *model* and *version*:

```
DeprecatedWFLTAction.do?__wfname=InsertEquipment&wfvar__hashmap0name=EQ0&wfv
ar__hashmap0model=HP&wfvar__hashmap0version=2.4&wfvar__hashmap1location=Madr
id&wfvar__ hashmap1country=Spain
```

The Array of Strings composition is very similar, but the parameter names now must begin with *wfvar__arrayiteratorX*, where X is a number beginning from 0.
Ex.
The next invocation generates an Array of Strings composed by "Madrid" and "Spain":

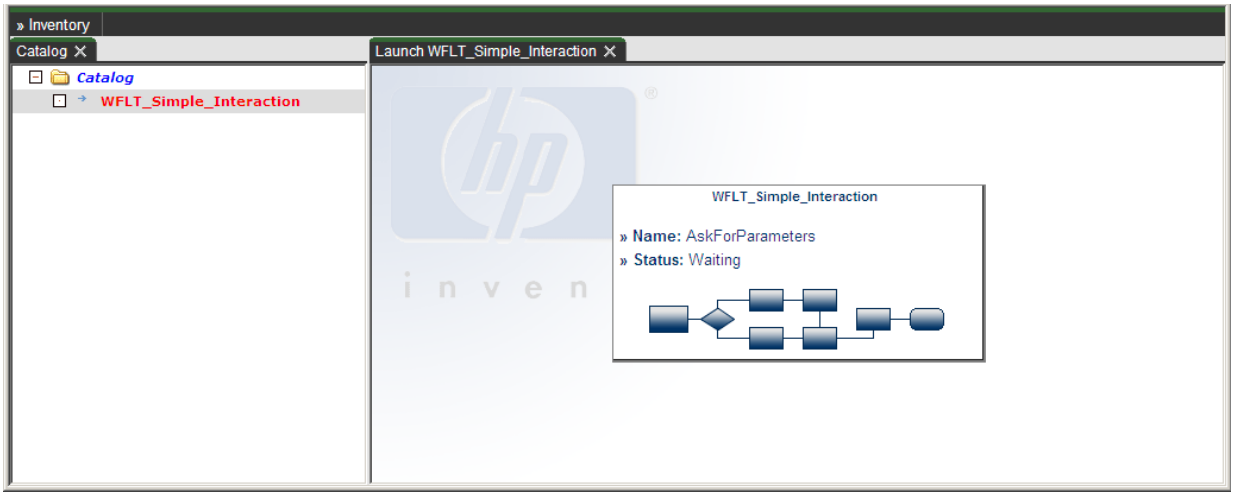
```
DeprecatedWFLTAction.do?__wfname=InsertEquipment&wfvar__arrayiterator0=Madri
d&wfvar__arrayiterator1=Spain
```

Since the previous versions of the WFLT used some initial JSP files from where all this process started, those JSP files have been maintained, but they should never more be used because actually the *WFLTAction.do* is considered the single entry point for the WFLT tool.

There are two JSP files which automatically redirect the user to the *WFLTAction.do*, and they are called *startWorkflow.jsp* and *inventoryStartWorkflow.jsp*.
There is another JSP file, called *hashmapStartWorkflow.jsp*, which automatically redirects the user to the *DeprecatedWFLTAction.do*.

10.1.4 Tracking workflows

Once a workflow has been started up and its job id has been obtained, it can be tracked. The appearance of the workflow tracking can be seen in the figure below.



It is also possible to track children workflows started up by the parent one, and there is no limitation on the depth of children workflows that starts up another grandchildren workflows.

There are two ways to make the tracking of children workflows: using the database or the CCWF.

a. Tracking through database

When using the database, the job id of the workflow that has to be tracked is stored in a given database table. It is the responsibility of the workflows to change the job id stored in that table. That means that when a workflow is started up, its job id is stored in database, and if that workflow starts up a child workflow, this child workflow must replace the job id stored in database with its own job id, and just before it is finished it must restore the original job id of the parent workflow. This way, the workflow tracking will always track the workflow whose job id is stored in the database each time it is requested.

There are three parameters/attributes that must be specified when calling the *WFLTAction.do* to perform this kind of tracking:

- *__wfDatasource*: the name of the data source to be used. This data source must have been defined previously and the user must have access to it.
- *__wfServiceName*: the name of the database table.
- *__wfServicePk*: the primary key of the entry where the job id has to be stored.

b. Tracking through the CCWF

When using the CCWF a flag is specified as a parameter/attribute. This means that the children workflows are started up using the nodes provided by the CCWF. See the section about the CCWF for further information.

Tracking workflows with the CCWF

__wfConcurrentCheck: it is "true" when the CCWF has to be employed to track the children workflows. The default value if not specified is "false".

10.1.4.1 ECP Command tracking

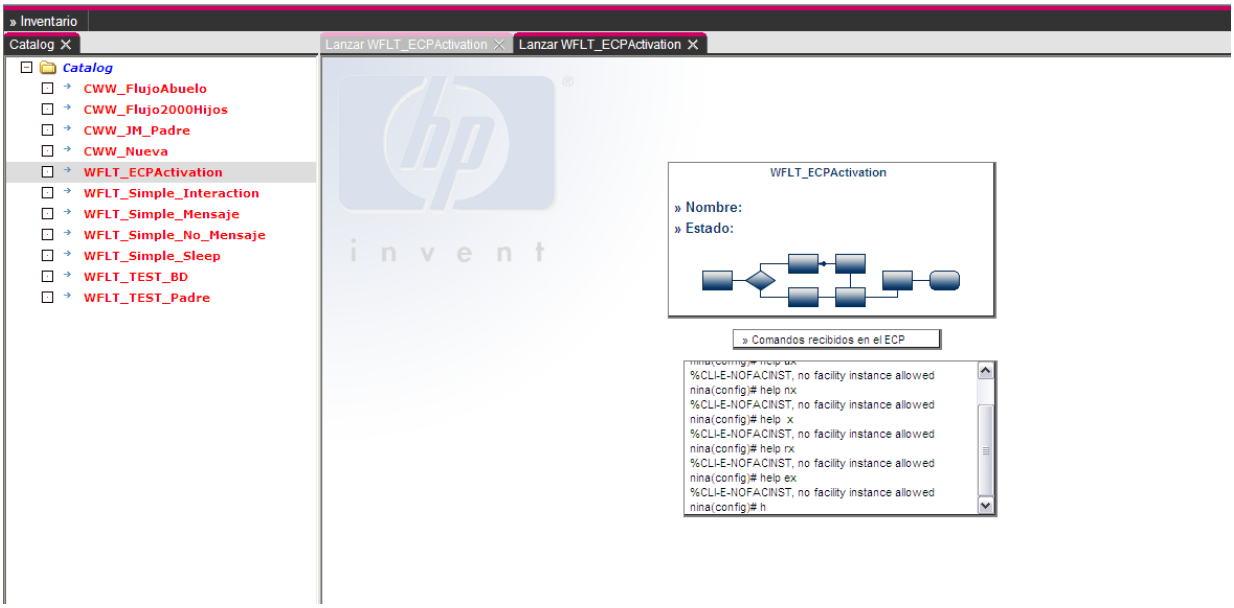
Using the WFLT is possible to track the commands sent and received by the Equipments Connection Pool (ECP). When a workflow launches an activation through the ECP the executed commands can be shown in the screen. By default only the last 20 commands will appear in the screen, but this number is configurable in the *wflt.properties* file.

In order to activate the ECP command tracking is necessary to include an identifier for that specific activation in the request under the key *__wf_command_id*. This id must be unique and will be used to filter the received messages and show only the ones related to a specific activation. At the same time, this identifier must be provided to the ECP under the same key (See the document "ECP Developers reference", section "3.4 Monitoring ECP commands through JMS" for further details). If no id is provided the jobId value will be taken by default.

It is also necessary to include the parameter *__wf_command_audit_active* with value "true" in order to activate the command tracking.

This example shows how to launch a workflow from the inventory with the command tracking feature activated defining an operation. The following example shows its appearance.

```
<Operation>
  <Name
Bundle="com/hp/spain/wflauncher test/ApplicationResources">launch.wf</Name>
      <Image>newtool.gif</Image>
      <Object>WfLauncherTest</Object>
      <OperationType>Lanzamiento</OperationType>
      <Action>
                  <Page>/activator/WFLTAction.do</Page>
                  <Param>
                        <Name>__wfname</Name>
                        <Value>WfLauncherTest.Name</Value>
                  </Param>
                  <Param>
                        <Name>__wfmwfmname</Name>
                        <Value>constant:localmwfm</Value>
                  </Param>
                  <Param>
                        <Name>__wfConcurrentCheck</Name>
                        <Value>constant:true</Value>
                  </Param>
                  <Param>
                        <Name>__wf_command_audit_active</Name>
                        <Value>constant:true</Value>
                  </Param>
                  <Param>
                        <Name>__wf_command_id</Name>
                        <Value>constant:garemo</Value>
                  </Param>
      </Action>
</Operation>
```



10.1.4.2 Interacting with workflows

There are two reasons to assume that a workflow is waiting for user interaction: the workflow must be in the "waiting" status (see the HPSA documentation for further information about the workflow status) and the step name. There are many reasons why a workflow can be set in a "waiting" status, so that cannot be the only reason to assume that the workflow is trying to interact with the user.

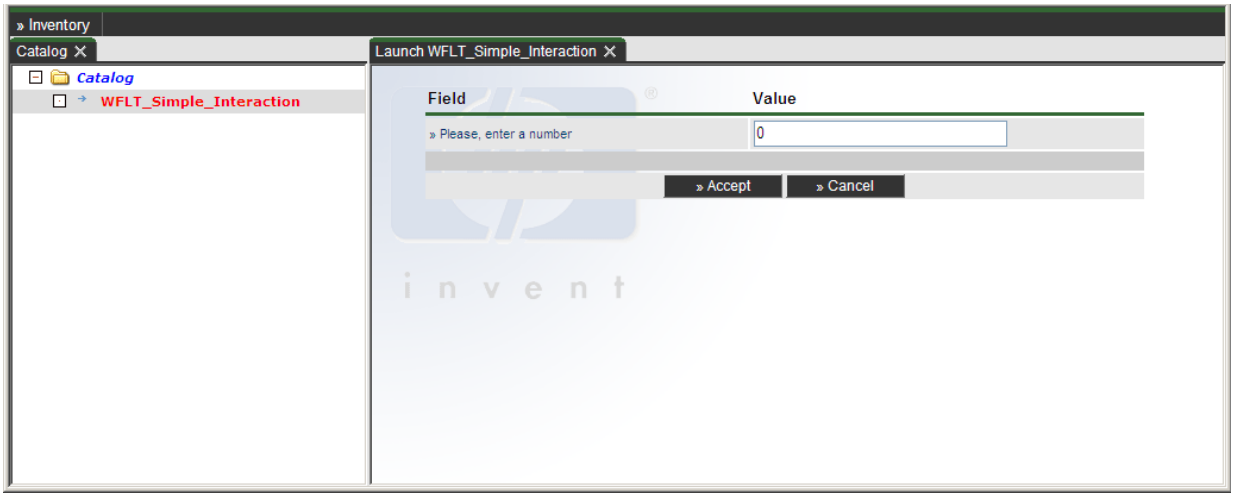
As it is explained in the Configuration section, in the *wflt.properties* file there can be specified several step names that will not be considered as interactive nodes, so those step names starting with any of these configured names will never be considered as interactive nodes.

The typical interactive nodes used for user interaction are the *AskFor* nodes. If a workflow has one *AskFor* node and its name is does not start with any of the configured step names it will show one screen like the one below.

When an interactive node is found, a JSP file is generated and placed below the *customJSP* directory.

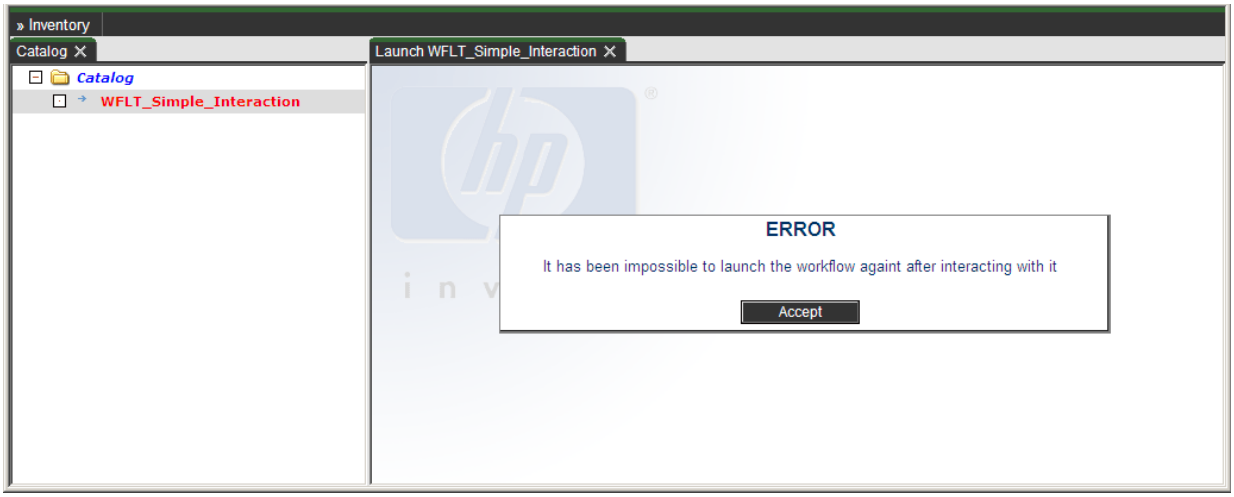
The custom JSP file will only be generated once, so if a previous custom JSP file already exists that will be used. This way, a custom JSP file can be changed and new functionality can be added to it.

The figure below shows an example of a custom JSP.



10.1.4.3 Tracking error

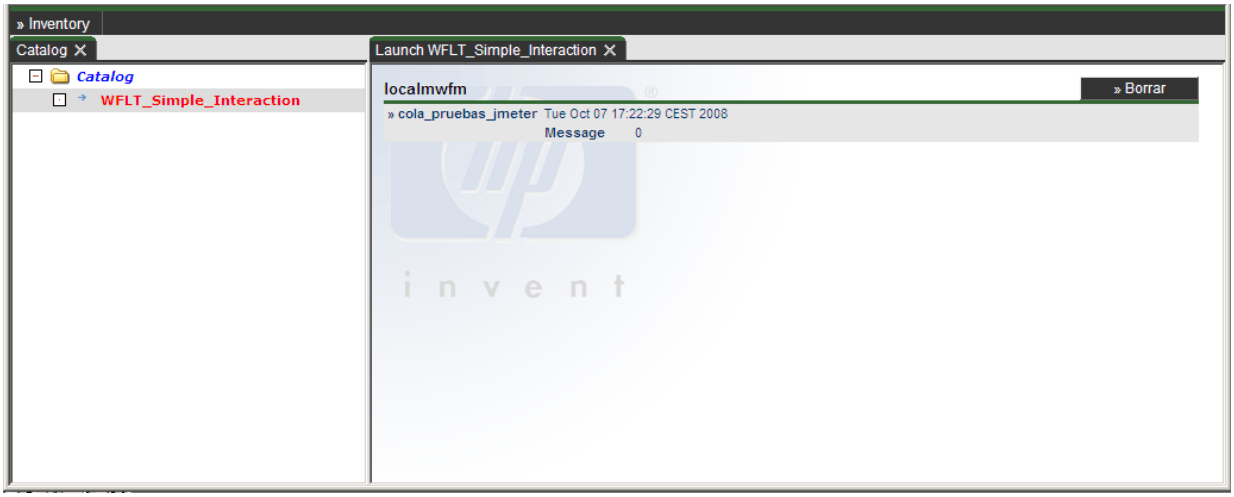
If a workflow execution fails, the WFLT will warn the user with a message like the one below.



10.1.4.4 Ending messages

The WFLT assumes that a workflow has finished when there cannot be found any workflow in the specified MWFM engine with the given job id.

When a workflow execution ends, every message belonging to the workflow and its children is displayed. Messages are typically thrown using the *PutMessage* node.


» Inventory
Catalog
Launch WFLT_Simple_Interaction
Catalog
WFLT_Simple_Interaction
localmwfm
» Borrar
» cola_pruebas_jmeter Tue Oct 07 17:22:29 CEST 2008
Message 0
i n v e n t

11 Concurrent Workflows Module

The CCWF (Concurrent Workflows Module) was designed with the aim of making possible for the user to launch concurrent workflows, capable of being executed concurrently in the MWFM. This can be achieved by using a new set of nodes in the execution of a common workflow.

Thanks to the CCWF it will no longer be necessary to define a large workflow to perform a task, the user can develop a few smaller jobs, each of them performing a small portion of the original task and working concurrently. Thus, it is possible to start up several children workflows (which can start up some other grandchildren workflows, and so on) and remain working concurrently until the children workflows become synchronized with their parent.

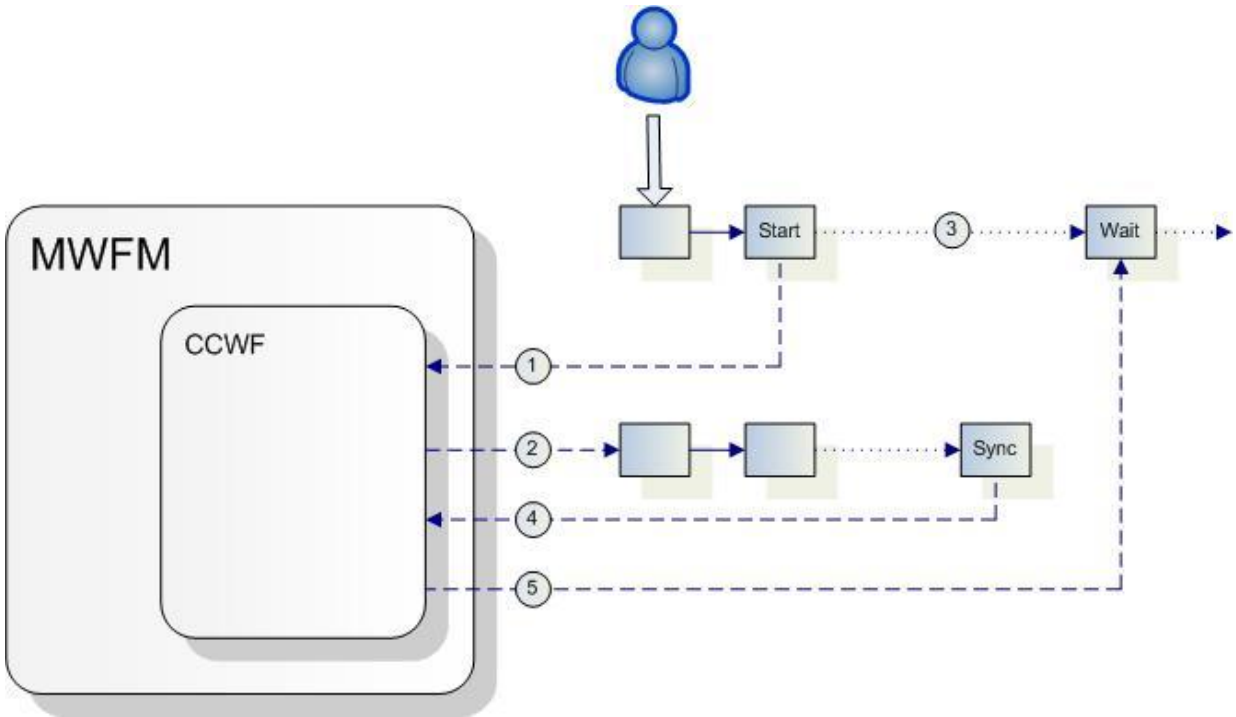
Synchronization among workflows allows sending data through the case packet, so children workflows are able to send information to their parent once they synchronize and the parent may have a different behavior depending on the information obtained from its children.

The tool provides three main features:

- To start up a new concurrent job.
- To synchronize with its father (awaking him if it is waiting).
- To wait for its children to finish.

11.1 Scenarios

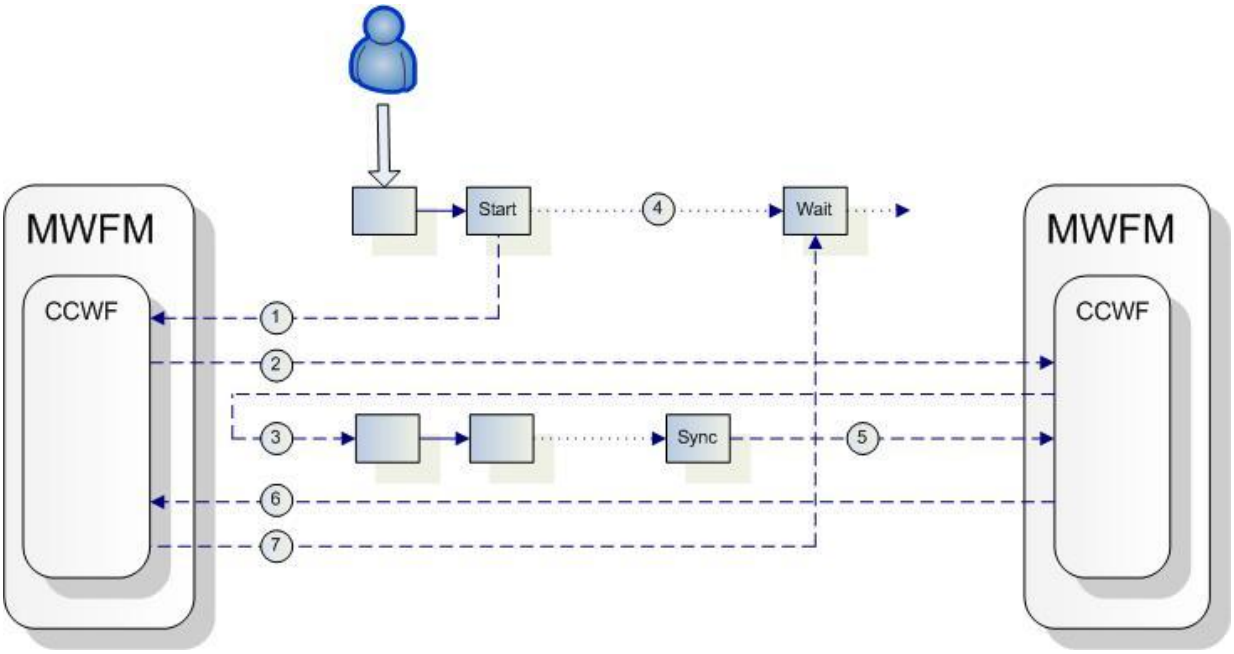
The figure below shows the whole process for starting up a child workflow and waiting until it is synchronized in a single CCWF. There, a parent workflow is started up by the user and then the next steps are followed:



1. The parent workflow reaches a *StartJobConcurrent* node. This node invokes the CCWF module to start the specified child workflow in the same MWFM engine of the parent.

2. The CCWF invokes the MWFM and starts up the child workflow. Once this workflow is started up, the parent workflow continues its execution, leaving the *StartJobConcurrent* node.
3. The parent workflow reaches the *WaitJobConcurrent* node. It will remain waiting here while the child workflow does not synchronize. This step may be reached after the step 4 (the child workflow may reach the *SyncConcurrent* node before the parent workflow reaches the *WaitJobConcurrent* node or it could be the other way around), but the whole process is not affected.
4. The child workflow reaches the *SyncConcurrent* node. This node invokes the CCWF for synchronization with its parent node.
5. The CCWF tries to synchronize with the parent workflow. Since the parent workflow is waiting for its children and this is the only children it has, it is awakened and its execution continues.

The figure below shows the same process described above, but this time two different CCWF modules are involved, that is, the child workflow is started up in a different MWFM engine.



1. The parent workflow reaches a *StartJobConcurrent* node. This node invokes the CCWF module to start the specified child workflow in a different MWFM engine than the parent's.
2. The CCWF notices that the child workflow has to be started up in another MWFM engine, so it invokes the remote CCWF to start up there the child workflow.
3. The remote CCWF invokes the MWFM where it is running and starts up the child workflow. Once this workflow is started up, the parent workflow continues its execution, leaving the *StartJobConcurrent* node.
4. The parent workflow reaches the *WaitJobConcurrent* node. It will remain waiting here while the child workflow does not synchronize. This step may be reached after the step 5 (the child workflow may reach the *SyncConcurrent* node before the parent workflow reaches the *WaitJobConcurrent* node or it could be the other way around), but the whole process is not affected.
5. The child workflow reaches the *SyncConcurrent* node. This node invokes the CCWF for synchronization with its parent node.

6. Since the parent node is not running in this MWFM, the remote CCWF notices this and invokes the local CCWF for synchronization.
7. The local CCWF tries to synchronize with the parent workflow. Since the parent workflow is waiting for its children and this is the only children it has, it is awakened and its execution continues.

11.1.1 Cleaning process

The cleaning process is started up when the configured interval gets finished. Each CCWF must take care of the jobs started up through it and is in charge of clean the database when they are finished.

The cleaning process algorithm is based on the next steps:

1. Get all the top workflows (those who has no parent workflow defined) which are running in the current MWFM. This means that each CCWF takes care of their top workflows.
2. Check if each top job is still running. This is made in three substeps: a) check if the workflow is already set as finished in database; b) check if the workflow is still running in the MWFM; and c) look for it in the Scheduler Module. If a) is true b) is not reached, and if b) is true c) is not reached either. If the job is finished, it is set as finished in database and its end date is filled with the current date. Note that the end date will not be probably the real end date of the workflow.
3. If the top workflow has been finished all its children are obtained, not taking care now of the children hierarchy. This is made in a single query to database.
4. Each child is checked as it was described in the step 2. If any child is still running the process is stopped for this hierarchy because the deletion and/or copy to historical must be done for the whole structure, beginning from the top workflow and ending with the last child. Children workflows are checked even if they are running on a different MWFM than the current one, opposite of what is made with the top workflows.
5. If the whole hierarchy from the top workflow to the last child is finished they are stored in a Vector to remove and copy to historical later on, depending on the configuration.
6. When all the hierarchies have been checked they are optionally removed and/or copied to historical. The removal is made comparing the end date of each top workflow with the current one, and those which difference is greater or equal to the configured value will be removed (the cascade deletion in database will remove also the children workflows).

11.2 Module Configuration

The CCWF is a module for the MWFM and must be configured in the *mwfm.xml* file. There can only be one CCWF module per MFWM. The name must be *ConcurrentWorkflowsModule*.

It is very important to notice that the CCWF may need several database connections at the same time, so the database module configured for it must be able to manage at least 10 connections to avoid undesired waiting times.

It is also very important to assure that if there are more than one CCWF configured in different engines, all of them must use the same database, so the *db* parameter, as it is going to be explained next, has to be pointing to the same database in the same IP.

There are some parameters that may be configured for this module:

Parameter	Mandatory	Description
remote_url	Yes	URL where the RMI service will be published.
mwfm_name	Yes	Name of the MWFM where the module is running.

		This name must be the same as the one configured in the auth.properties file of the Service Container. It will be used to know where has been started up each workflow and thus, it will be possible for the WFLT to track any workflow on any MWFM configured in the auth.properties file
Db	Yes	Name of the database module is going to be used by the CCWF. It must be configured in the same mwfm.xml file. It is very important to assure that the number of connections managed by this database module is at least 10, because the CCWF will manage multiple connections at the same time
cleaning_interval	Yes	Interval, in milliseconds, between cleaning checks. The default value is one minute. A value of 0 will indicate that no cleaning process will be performed anytime
move_to_historical	No	Boolean value which indicates if the finished jobs must be moved to the historical table. Note that the historical table will never be cleaned up, so it is the responsibility of the customer application to assure that the historical jobs table does not reach a heavy size. The default value is true
days_for_deletion	No	Number of days a finished workflow will remain in the running jobs table until it is removed. The default value is 3 days
check_finished_jobs_on_startup	No	Boolean value which indicates if the jobs set as finished in the database must be also checked or not on start up. Thus, if any persisted job removed previously has been restored while the MWFM remained stopped, it will be checked again. The default value for this parameter is false.

The example below shows how to configure the CCWF with a 60 minutes cleaning interval and deleting the jobs from active running workflows as soon as they are copied to the historical (note that the module must be called *ConcurrentWorkflowsModule*:

```
<Module>
  <Name>ConcurrentWorkflowsModule</Name>
  <Class-Name>
    com.hp.spain.engine.module.concurrentworkflows.RemoteAsynchronousWorkflowLockImpl
  </Class-Name>
  <Param name="mwfm_name"         value="primary_mwfm"/>
  <Param name="remote url"        value="//localhost:2000/concurrent workflows"/>
  <Param name="db"                value="db"/>
  <Param name="cleaning_interval" value="3600000"/>
  <Param name="days_for_deletion" value="0"/>
</Module>
```

11.3 Nodes

The CCWF provides three nodes which can be used to start up children workflows, wait for them until they finished, and synchronize children workflows with their parent workflow.

11.3.1 StartJobConcurrent

This node starts up a child workflow in a specified MWFM engine. By default, if no *remote_ip* parameter is specified, the child workflow will be started up in the current MWFM.

Since there is no limitation on the number of children workflows a parent workflow can start concurrently, the CCWF provides an optional mechanism for setting a limitation on this using the *max_wf_number* parameter. This means that a parent workflow that has to start up, for instance, 100 children workflows and has a *max_wf_number* limitation of 10 will remain starting up its children assuring that there are never more than 10 running children at the same time, and once that 10 children workflows has been started up it will be kept waiting until one of them becomes finished and another child may be started up.

Once the child workflow has been started up, its job id may be optionally stored in a Vector defined in the parent workflow case packet under the key *object_child_id*. Thus, in case of the father job suffering an error the *WFTransactionHandler* will be able to kill all his children. See the information about the *Workflow Transaction Module* for further information.

The parameters accepted by this node are:

Parameter	Mandatory	Description
Job_name	Yes	Name of the child workflow to be started up. The value specified is treated as a constant if you don't specify *variable:* prefix
max_wf_number	No	Maximum number of children workflows the parent workflow can start up concurrently. If it is not specified, there is no limitation on this
variable+i	No	Numbered parameter, starting from 0, with the case packet variable names whose values must be taken and sent to the child workflow.
case_packet_var+i	No	Numbered parameter, starting from 0, with the new name of the variable "i" in the case packet of the child workflow. If the name is the same in the parent workflow and the child one, this parameter is not needed.
remote_ip	No	IP or the whole RMI URL of the CCWF where the child workflow has to be started up. If it is not specified, the child workflow will be started up in the MWFM where the parent workflow is running (most common behavior). Only should be used when we want to start the child workflow in a remote MWFM
local_ip	No	IP or RMI URL of the current CCWF. Should be used when we want to start the child workflow in a remote MWFM, because the child workflow will need this value to synchronize with the parent workflow. It is the responsibility of the child workflow to store this value in its case packet and use it later for synchronization
object_child_id	No	Object variable used to store the children JOB IDs, to allow *WFTransactionHandler* be able to kill them if the parent workflow end with an error.

The following example starts up a child workflow in the current MWFM, sending the variables *totalString*, *indexChild* and *totalChild* changing the destination of *totalChild* in the child workflow to *totalWF*.

```
<Process-Node>
  <Name>Start up child workflow</Name>
  <Description>Start up a child workflow</Description>
  <Action>
    <Class-Name>
      com.hp.spain.node.concurrentworkflows.StartJobConcurrent
    </Class-Name>
    <Param name="job_name"          value="ConcurrentChildWF"/>
    <Param name="variable0"         value="totalString"/>
    <Param name="variable1"         value="indexChild"/>
    <Param name="variable2"         value="totalChild"/>
    <Param name="case_packet_var2" value="totalWF"/>
  </Action>
</Process-Node>
```

11.3.2 SyncConcurrent

This node makes the synchronization between children workflows and their parents. When the child job reaches this node it will synchronize with its parent, sending the specified case packet variables to it. If the parent workflow is waiting for its children to finish and all the children workflows has ended, the job will be removed from the jobs queue.

Note that the synchronization is made in the parent's CCWF. This is really important if children workflows are being started up in a different MWFM engine than the parent's, because in that case the *remote_ip* and *local_ip* becomes mandatory. See the *API Reference* section for this node for more information.

The following parameters can be configured for this node:

Parameter	Mandatory	Description
variable+i	No	Numbered parameter, starting from 0, with the case packet variable names whose values must be expected from the child workflow. There is no limitation on the number of variables defined here
destination+i	No	Numbered parameter, starting from 0, with the new name of the variable "i" in the case packet of the parent workflow. If the name is the same in the parent workflow and the child one, this parameter is not needed.
remote_ip	No	IP or the whole RMI URL of the CCWF where the parent workflow is running. If it is not specified, the parent workflow is searched in the MWFM where the workflow is running (most common behavior). Only should be used when the parent workflow is running in a remote MWFM.
local_ip	No	IP or RMI URL of the current CCWF. Only should be used when the parent workflow is running in a remote MWFM.

The example below shows how to synchronize with the parent workflow in the current MWFM. As it can be seen, neither the *remote_ip* nor the *local_ip* parameters have been specified.

```
<Process-Node>
  <Name>Synchronizes with parent job</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.concurrentworkflows.SyncConcurrent
    </Class-Name>
    <Param name="variable0" value="return code"/>
    <Param name="variable1" value="error_description"/>
  </Action>
```

```
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

11.3.3 WaitJobConcurrent

This node waits until the children workflows finish their execution and receive the variables sent by the children workflows. When the parent job reaches this node it will wait for its children to terminate, and once this happens it will continue its execution.

The following parameters can be configured for this node:

Parameter	Mandatory	Description
Queue	Yes	Name of the queue where the workflow will remain waiting until all its children become finished. The value specified is treated as a constant if you don't specify variable: prefix
variable+i	No	Numbered parameter, starting from 0, with the case packet variable names whose values must be expected from the child workflow. There is no limitation on the number of variables defined here

The following example waits until in the queue stored in the case packet variable *concurrentQueue*, the children workflows finish and update the case packet variables *return_code* and *error_description* with the values provided by the children workflows.

```
<Process-Node>
  <Name>Waiting for child jobs</Name>
  <Description>Waiting until child jobs are all synchronized</Description>
  <Action>
    <Class-Name>
      com.hp.spain.node.concurrentworkflows.WaitJobConcurrent
    </Class-Name>
    <Param name="queue"     value="variable:concurrentQueue"/>
    <Param name="variable0" value="return_code"/>
    <Param name="variable1" value="error description"/>
  </Action>
</Process-Node>
```

12 Workflow Transaction Module

This module contains the classes needed for the composition of database transactions from a workflow, the insertion of operations among them and the undoing of them if it becomes necessary.

A workflow, or a workflow tree, may contain different transactions conforming a group of transactions. Each of the different workflow nodes or of their children workflows may insert their operations in one of them, which is defined in each node's configuration, and all of them will be processed by the End Handler of the workflow that has created the transaction.

To use this module in a workflow the following case packet variables must be defined in the workflow:

Case packet variable	Type	Description
WF_TRANSACTIONS_GROUP_ID	Integer	This case packet variable will be filled with information when a transaction is initialize using the *WFTransactionBegin* node and must be sent to all the workflows involved in the transaction
return_code	String	Variable used by the *WFTransactionErrorHandler* to store the unexpected error code
error_description	String	Variable used by the *WFTransactionErrorHandler* to store the exception message caused the workflow to finish
Cancel	Boolean	Indicate to the *WFTransactionHandler* whether the workflow has been cancelled by user through the GUI or the MWFM API. This variable must be initialize to true and should be changed in the last node of the workflow to true
Finish	Boolean	Indicate to the *WFTransactionHandler* whether the workflow has finished correctly and if it has to undo the tasks performed by the workflow. This variable must be initialize to *false* and should be changed to *true* if the workflow has been executed correctly.

```
<Case-Packet>
  <Variable name="Finish" type="Boolean"/>
  <Variable name="Cancel" type="Boolean"/>
  <Variable name="WF TRANSACTIONS GROUP ID" type="Integer"/>
  <Variable name="return code" type="String"/>
  <Variable name="error_description" type="String"/>
</Case-Packet>

<Initial-Case-Packet>
  <Variable-Value name="Finish" value="false"/>
  <Variable-Value name="Cancel" value="true"/>
</Initial-Case-Packet>
```

12.1 Functionality

This module provides two handlers and several nodes to ensure the transactional behavior in the workflow execution, when we work with the inventory, the lock manager, the concurrent workflows and SOSA.

The first task to do in the workflows is initialize the transaction and assign a name to it, to later give it to the different nodes to store the information. To initialize the transaction we should call the node *WFTransactionBegin* given a case packet variable to store the name of the transaction created that should be used later when we want to work with this transaction (in nodes and in the end handler). The following is an example of it:

```
<Process-Node>
  <Name>Start new transaction</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.WFTransactionBegin
    </Class-Name>
    <Param name="WF TRANSACTION NAME" value="transactionNameVar"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

NOTE: The case packet variable has to be empty before call the node and will be filled with the identifier

Once we have defined the transaction, we can start working with it in our workflow. In the following chapters, all the operations the transaction module allowed will be explained in detail and in chapter 4.3 you can access to a detailed description of all the nodes.

12.1.1 Lock functionality

The Workflow Transaction Module allow working with object locking, assuring the locks will be free-up when the workflow finalize. To do that, the Workflow Transaction Module works in conjunction with the Lock Manager Module. In the following chapters the different locking processes related to the Workflow Transaction Module are explained. For detailed information about LockManager see the "Lock Manager – Developer reference".

12.1.1.1 Object locking

With this operation a workflow can ask a lock for a specified object (through the LockManager module). If the resource is currently locked, the workflow will be queued until the resource be released and could get the lock. Afterwards, it appends the proper rollback operations in the specified transaction to be able to unlock the resource if the workflow does not end properly.

The following example, lock the bean *com.hp.example.Bean* with the primary key stored in the case packet variable *BeanId*, storing the lock identifier in the case packet variable *lock_id* and saving the information of the lock in the given transaction:

```
<Process-Node>
  <Name>Lock bean</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.LockInventory
    </Class-Name>
    <Param name="WF TRANSACTION NAME" value="transactionNameVar"/>
    <Param name="bean0" value="com.hp.example.Bean"/>
    <Param name="primary_key0" value="BeanId"/>
    <Param name="job_id" value="JOB_ID"/>
    <Param name="lock id" value="lock id"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

NOTE: Notice that the bean doesn't has to exist in the inventory to be locked and, if it exists in the inventory someone accessing directly to the inventory (using the inventory GUI or the standard HPSA nodes to work with the inventory) can modify or inclusive delete the resource when it is locked by workflow, because the LockManager is a semaphore system (see the "LockManager – Developer Reference" for more details)

12.1.1.2 Object locking without queuing

With this operation, a workflow can do the same than previous one, with the difference that if the resource is currently locked, the node will fail without been queued, setting a value different from 0 in the RET_VALUE variable and, depending on the parameter *soft*, throwing an exception.

The following example, will work like the example in the previous chapter, but will update the *RET_VALUE* with a 0 if it is done properly. If it fails because the resource is currently locked it will throw an exception stopping the execution of the workflow:

```
<Process-Node>
  <Name>Lock bean without queuing</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.LockInventoryWithoutEnqueue
    </Class-Name>
    <Param name="WF_TRANSACTION_NAME" value="transactionNameVar"/>
    <Param name="bean0" value="com.hp.example.Bean"/>
    <Param name="primary key0" value="BeanId"/>
    <Param name="job id" value="JOB ID"/>
    <Param name="lock_id" value="lock_id"/>
    <Param name="soft" value="false"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

12.1.1.3 Assigning an existing lock identifier

With this operation, a workflow can receive an existing lock identifier and establish itself as the owner of the lock, appending it to a given transaction. This operation is useful when the lock is gained for an external system and then a workflow has to continue working with the resource without releasing it in the transfer (to avoid someone could gain the lock in-between this process).

The following example, assign the lock identified stores in the case packet variable *previousLockId*, to the current workflow, appending it to the given transaction and storing the lock identifier in *lockId*:

```
<Process-Node>
  <Name>Assign Lock Identifier</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.AssignLockId
    </Class-Name>
    <Param name="WF TRANSACTION NAME" value="transactionNameVar"/>
    <Param name="job id" value="JOB ID"/>
    <Param name="lock_id" value="lockId"/>
    <Param name="value" value="previousLockId"/>
    <Param name="soft" value="true"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

NOTE: Notice that the information previously stored in the case packet variable *lockId* will be substituted with the new lock identifier.

12.1.1.4 Object unlocking

This operation allows unlocking a resource previously locked free-up it to other workflows can work with it and removing the operations previously appended to the transaction. When this operation is perform, the previous delayed operations associated to this lock will be done (see the “Delayed Delete” and “Delayed Update” operations for more information).

In the following example, the given lock is released, the lock operation is removed from the given transaction and the delayed operations done using this lock will be performed:

```
<Process-Node>
  <Name>Unlock</Name>
    <Action>
      <Class-Name>
        com.hp.spain.node.wftransaction.UnlockInventory
      </Class-Name>
      <Param name="WF_TRANSACTION_NAME" value="transactionNameVar"/>
      <Param name="lock_id" value="lock_id"/>
```

```
    </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

12.1.2 Inventory

The Workflow Transaction Module allows working with inventory objects, assuring the transactional if the workflow fails and do certain tasks when it end up properly. To do that, the Workflow Transaction Module works record information about the transaction done by some workflow nodes (that are part of the module). In the following chapters the different inventory processes related to the Workflow Transaction Module are explained.

12.1.2.1 Inserting Inventory beans

With this operation we can insert beans in the inventory, appending afterwards the proper rollback operations for the given transaction. At the same time we insert the bean, we can lock it (if we specify it in the node).

In the following example, a bean is inserted in the inventory with the values specify in the attribute/value pairs parameters, storing the information in the given transaction to assure the rollback is something fails during the workflow execution and retrieve the complete bean to the case packet variable specified:

```
<Process-Node>
  <Name>Insert bean</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.InsertInventory
    </Class-Name>
    <Param name="WF_TRANSACTION_NAME" value="transactionNameVar"/>
    <Param name="db" value="db"/>
    <Param name="bean" value="com.hp.example.Bean"/>
    <Param name="attribute0" value="constant:Name"/>
    <Param name="value0" value="beanName"/>
    <Param name="attribute1" value="constant:Status"/>
    <Param name="value1" value="constant:NEW"/>
    <Param name="bean object" value="beanObject"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

12.1.2.2 Updating Inventory bean

With this operation we can update beans in the inventory, appending afterwards the proper rollback operations for the given transaction. Before updating the bean information, LockManager is asked to know if this workflow has the lock over the bean is trying to update.

In the following example, a bean is updated in the inventory with the values specify in the attribute/value pairs parameters, storing the information in the given transaction to assure the rollback is something fails during the workflow execution.

```
<Process-Node>
  <Name>Update bean</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.UpdateInventory
    </Class-Name>
    <Param name="WF TRANSACTION NAME" value="transactionNameVar"/>
    <Param name="bean" value="com.hp.example.Bean"/>
    <Param name="primary_key" value="beanPrimaryKey"/>
    <Param name="attribute0" value=”constant:BeanName”/>
    <Param name="value0" value=”constant:Example”/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
```

```
</Process-Node>
```

12.1.2.3 Resource reservation

This node reserves the given resources of a given pool, appending afterwards the proper rollback operations for the transaction.

For example:

```
<Process-Node>
  <Name>Reserve a resource</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.ReserveResource
    </Class-Name>
    <Param name="WF TRANSACTION NAME" value="transactionNameVar"/>
    <Param name="bean" value="com.hp.example.Bean"/>
    <Param name="pool" value="myPool"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

12.1.2.4 Delayed inventory operations

These operations allow doing tasks when the workflow finishes successfully avoiding the problems that may appear if some element is deleted or released and after that the workflow executions fails. The delayed operations available are delete, release and update.

The following examples mark a bean instance to be deleted, then release a resource previously reserved and updates a bean. These three tasks will be done at the end of the workflow if it finishes properly:

```
<Process-Node>
  <Name>Remove bean</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.DelayedDelete
    </Class-Name>
    <Param name="WF_TRANSACTION_NAME"        value="transactionNameVar"/>
    <Param name="bean"                       value="com.hp.example.Bean"/>
    <Param name="primary key"                value="beanPrimaryKey"/>
  </Action>
  <Next-Node>Release resource</Next-Node>
</Process-Node>

<Process-Node>
  <Name>Release resource</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.DelayedReleaseResource
    </Class-Name>
    <Param name="WF TRANSACTION NAME"        value="transactionNameVar"/>
    <Param name="bean"                       value="com.hp.example.ResBean"/>
    <Param name="primary_key"                value="beanPrimaryKey"/>
  </Action>
  <Next-Node>Update bean</Next-Node>
</Process-Node>

<Process-Node>
  <Name>Update bean</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.DelayedUpdate
    </Class-Name>
    <Param name="WF_TRANSACTION_NAME" value="transactionNameVar"/>
    <Param name="bean" value="com.hp.example.Bean"/>
    <Param name="primary key" value="beanPrimaryKey"/>
    <Param name="attribute0" value="constant:BeanName"/>
    <Param name="value0" value="constant:Example"/>
```

```
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

12.1.2.5 Working with historical

With these operations, in a workflow we can move and recover bean from historic, appending afterwards the proper rollback operations for the transaction if the workflows fails.

In the following example, the given object bean is moved to the historic and the information is stored in the transaction:

```
<Process-Node>
  <Name>Move bean to historical</Name>
  <Action>
    <Class-Name>
      com.hp.spain.node.wftransaction.MoveToHistory
    </Class-Name>
    <Param name="WF TRANSACTION NAME" value="transactionNameVar"/>
    <Param name="bean" value="beanObject"/>
  </Action>
  <Next-Node>DoNextTask</Next-Node>
</Process-Node>
```

NOTE: Notice that the bean must be defined with historic in order to move or recover from history. If not an exception will be thrown when execute this node.

12.2 Configuration

The WT is a HPSA module and must be defined in the *mwfm.xml* file. It accepts the following parameters:

Parameter	Mandatory	Description
save_workflows	No	Boolean parameter which indicates if the running workflow definition file should be saved for the running jobs. Thus, in case of shutdown, those jobs will be restored in the MWFM using the original and not the one currently available in the MWFM. Default value is *false*
default_persistence	No	Boolean parameter which indicates whether the running jobs must be persisted or not. Default value is *false*
persistence_dir_path	Yes	If the *default_persistence* parameter is *true*, this parameter indicates the path where the persisted workflows will be saved. It is mandatory that the directory set in this parameter contains a text file called *wftransaction.sequence* and its contents must be the number 1, just a single character.
workflow_without_persistence+i	No	Numbered parameter with the different names of those workflows which are not going to be persisted.
debug	No	Boolean parameter which indicates if the debug traces must be written or not. The default value is *false*

For example:

```
<Module>
  <Name>transaction_manager</Name>
```

```
  <Class-Name>
    com.hp.spain.engine.module.wftransaction.WFTransactionManagerModule
  </Class-Name>
  <Param name="save workflows" value="true"/>
  <Param name="default persistence" value="true"/>
  <Param name="persistence dir path" value="/tmp/wftransactions"/>
  <Param name="workflow_without_persistence0" value="MY_FIRST_JOB"/>
</Module>
```

12.3 Nodes

12.3.1 AssignLockId

`com.hp.spain.node.wftransaction.AssignLockId`

This node assigns the current workflow as the owner of a given lock, appending it to a open transaction.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
job_id	Yes	Workflow identifier. Always use JOB_ID case packet variable to avoid errors
lock_id	Yes	Variable where the lock will be stored
value	Yes	Value of the lock identifier
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.2 DelayedDelete

`com.hp.spain.node.wftransaction.DelayedDelete`

This node marks some instance bean to be delete at the end of the workflow if it ends properly.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	Name of the database module. The default value is db.
bean	Yes	Bean name
primary_key	No	Primary key
deletemethod	No	Name of the find method to remove the bean
deletevariable + i	No	Variables needed for the find method to remove the bean. Only mandatory if the deletemethod parameter is specified.
lock_id	No	Lock identifier (if the bean has been locked)
error_message	No	Variable where the error message will be returned, if any. If not specified, it will be stored in error_description
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.3 DelayedReleaseResource

`com.hp.spain.node.wftransaction.DelayerReleaseResource`

This node marks some instance bean to be release at the end of the workflow if it ends properly.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	Name of the database module. The default value is db.
bean	Yes	Bean name
primary_key+i	No	Primary keys
error_message	No	Variable where the error message will be returned, if any. If not specified, it will be stored in error_description
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.4 DelayedUpdate

`com.hp.spain.node.wftransaction.DelayedUpdate`

This node marks retrieve the updates to be done to an instance bean at the end of the workflow if it ends properly.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	Name of the database module. The default value is db.
bean	Yes	Bean name
primary_key	No	Primary key
find_method	No	Name of the bean's find method
find_variable+i	No	Variables for the find method. Orly mandatory if the find_method parameter is defined
field+i	Yes	Field names to update
variable+i	Yes	New values for the specified field names
lock_id	No	Lock identifier
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.5 InsertInventory

`com.hp.spain.node.wftransaction.InsertInventory`

The node creates instances in the inventory, appending afterwards the proper rollback operations for the transaction. The accepted parameters for this node are:

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	MWFM database module name configured in

		mwfm.xml. If no one is given, "db" module is used.
bean	Yes	Name of the JavaBean class that is used for storing the data
attribute+i	Yes	Name of a key in the JavaBean that is created. The parameter must be repeated for all attributes in the JavaBean being initially assigned.
value+i	Yes	Used in conjunction with the attribute attributes to specify the value of the individual *attributes* in the JavaBean
attributeExt+i	No	Name of an extended attribute to ne inserted. The parameter must be repeated for all extended attributes in the JavaBean being initially assigned.
valueExt+i	No	Used in conjunction with the *attributeExt* attributes to specify the value of the individual extended attributes in the JavaBean
field+i	No	Name of the field to be queried after invoke the store method
variable+i	No	Used in conjunction with the *field* attributes to specify the case packet variable where the value of the queried field will be saved
bean_object	No	Name of the variable where the created JavaBean instance is returned
lock_id	No	Name of the case packet variable where the lock identifier the insert will be store. If this parameter is specified, the bean is locked after the insertion
soft	No	Boolean value which indicates whether exceptions should be thrown. Default value is false
error_message	No	Variable where the error message will be returned, if any. If not specified, it will be stored in *error_description*

12.3.6 LockInventory

`com.hp.spain.node.wftransaction.LockInventory`

The node locks an inventory resource queuing if the resource is locked for other process, appending the information to the transaction and assigning a lock identifier.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
bean+i	Yes	Bean class
primary_key+i	No	Primary keys of the beans to lock
array	No	Array of bean to lock
string_array	No	Boolean value which indicates if the array si composed of Strings (true) or objects (false).
lock_id	No	Variable to store the lock identifier. By default, lock_id
error_message	No	Variable where the error message will be returned, if any. If not specified, it will be stored in error_description

12.3.7 LockInventoryWithoutEnqueue

`com.hp.spain.node.wftransaction.LockInventoryWithoutEnqueue`

The node locks an inventory resource appending the information to the transaction and assigning a lock identifier. If the resource is locked for another process returns an error.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
bean+i	Yes	Bean's class
primary_key+i	No	Primary keys
array	No	Array with the beans to lock
lock_id	No	Variable to store the lock identifier. By default, lock_id
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.8 MoveToHistory

`com.hp.spain.node.wftransaction.MoveToHistory`

This node moves some inventory bean to the history tables and removing it from the original one.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	Name of the database module. The default value is db
bean+i	No	Object to move
array_bean+i	No	Array with the objects to move
primary_key+i	No	Identifier of the object to move
bean_class+i	No	Class of the object. Only mandatory when the primary_key is specified
lock_id	No	Lock identifier
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.9 RecoverFromHistory

`com.hp.spain.node.wftransaction.RecoverFromHistory`

This node recovers the bean information from the history and stores it in the original table.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	Name of the database module. The default value is db
bean+i	No	Object to recover
array_bean+i	No	Array with the objects to recover
primary_key+i	No	Identifier of the object to recover
bean_class+i	No	Class of the object. Only mandatory when the primary_key is specified
lock_id	No	Lock identifier
soft	No	Boolean value which indicates whether exceptions should be thrown

12.3.10 ReserveResource

`com.hp.spain.node.wftransaction.ReserveResource`

The node is used to reserve resources from the inventory.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
db	No	Name of the database module. The default value is db
bean	Yes	Class name
pool	No	Resources pool
pool_name	No	Name of the resources pool
primary_key	No	Identifier of the resource to lock
variable	Yes	Variable to store the pk of the locked resource
field+i	No	Field name to query
field_var+i	No	Variable to store the value of the field

12.3.11 UnlockInventory

`com.hp.spain.node.wftransaction.UnlockInventory`

The node is used to unlock a previously locked bean from the inventory.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
lock_id	No	Lock identifier

12.3.12 UpdateInventory

`com.hp.spain.node.wftransaction.UpdateInventory`

The node updates instances in the inventory, appending afterwards the proper rollback operations for the transaction. The accepted parameters for this node are:

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable with the transaction name to use
Db	No	MWFM database module name configured in mwfm.xml. If no one is given, “db” module is used.
Bean	Yes	Name of the JavaBean class that is used for storing the data
primary_key	Yes	Primary key of the bean to update
attribute+i	Yes	Name of a key in the JavaBean that is update. The parameter must be repeated for all attributes in the JavaBean being updated.
value+i	Yes	Used in conjunction with the attribute *attributes* to specify the value of the individual attributes in the JavaBean
attributeExt+i	No	Name of an extended attribute to be updated. The parameter must be repeated for all extended attributes in the JavaBean being updated.
valueExt+i	No	Used in conjunction with the *attributeExt* attributes to

		specify the value of the individual extended attributes in the JavaBean
lock_id	No	Name of the case packet variable with the lock identifier
bean_object	No	Variable to store the updated object
error_message	No	Variable where the error message will be returned, if any. If not specified, it will be stored in *error_description*

12.3.13 WFTransactionBegin

`com.hp.spain.node.wftransaction.WFTransactionBegin`

Begin a new workflow transaction to store operations done in a workflow or a tree-workflows with the nodes related with the Workflow Transaction Module.

Parameter	Mandatory	Description
WF_TRANSACTION_NAME	Yes	Case packet variable where the transaction name created will be stored

12.4 Handlers

The handler is the responsible to assure the transactional of the task performed by the workflow. To do it, we have to specify for each transaction created what are the operations to be performed. The different options are the following:

- *rollback_beans*: if the workflow finishes with failure, it indicates whether it has to be performed a rollback of any insertion or update made in database.
- unlock_beans: indicates whether the locked beans have to be unlocked once the workflow is finished.
- *release_resources*: when the workflow finished with failure, it indicates whether the resources which have been reserved during the transaction must be released.
- *delete_delayed*: when the workflow finishes without failure, it indicates whether the delayed beans have to be removed.
- *update_delayed*: when the workflow finishes without failure, it indicates whether the delayed beans have to be updated.
- *release_delayed*: when the workflow finishes without failure, it indicates whether the delayed beans have to be released.
- *rollback_moved_beans*: the workflow finished with failure, it indicates whether it has to be performed a rollback of those beans moved to historical.
- *rollback_recovered_beans*: the workflow finished with failure, it indicates whether it has to be performed a rollback of those beans recovered from the historical.
- *remove_history_beans*: when the workflow finishes without failure, it indicates whether i those beans moved to historical have to be removed.

The way to do it is defining Case Packet variables to control the operations that will be taken in the handler and assigning them the values *true* or *false*, respectively.

It is very important that only these variables that are going to be used are defined. Thus, if a workflow must not undo the database modifications there must neither be defined nor used in the End handler the variable *rollback_beans*.

These is an example on how to define the handler and the case packet variables

```
<End-Handler>
   <Class-Name>com.hp.spain.node.wftransaction.WFTransactionHandler</Class-Name>

   <Param name="WF_TRANSACTION_NAME0"      value="wf_trans_lock"/>
   <Param name="rollback_beans0"           value="rollback_beans"/>
   <Param name="release_resources0"        value="release_resources"/>
   <Param name="delete_delayed0"           value="delete_delayed"/>
   <Param name="update_delayed0"           value="update_delayed"/>
   <Param name="release_delayed0"          value="release_delayed"/>
   <Param name="rollback_moved_beans0"     value="rollback_moved_beans"/>
   <Param name="rollback_recovered_beans0" value="rollback_recovered_beans"/>
   <Param name="remove_history_beans0"     value="remove_history_beans"/>
   <Param name="concurrent_sync"           value="constant:true"/>
   <Param name="async handler"             value="constant:false"/>
</End-Handler>

<Case-Packet>
   <Variable name="rollback beans"           type="Boolean"/>
   <Variable name="unlock beans"             type="Boolean"/>
   <Variable name="release resources"        type="Boolean"/>
   <Variable name="delete_delayed"           type="Boolean"/>
   <Variable name="update_delayed"           type="Boolean"/>
   <Variable name="release_delayed"          type="Boolean"/>
   <Variable name="rollback moved beans"     type="Boolean"/>
   <Variable name="rollback recovered beans" type="Boolean"/>
   <Variable name="remove_history_beans"     type="Boolean"/>
</Case-Packet>
```

12.4.1 End handler

This handler is invoked every time the workflow gets finished, regardless of whether it finishes with failure or not.

The actions taken for each of the transactions depend on the following parameters and the value of the *Finish* and *Cancel* case packet variables:

Parameter	Mandatory	Description
WF_TRANSACTION_NAME+i	No	Name of each workflow transaction to be processed
rollback_beans+i	No	Boolean value which indicates if the rollback of the inserted or updated beans has to be performed. This parameter is taking into consideration if the case packet variable Finish is false and the case packet variable Cancel is true
rollback_moved_beans+i	No	Boolean value which indicates if the rollback of the beans moved to historical has to be performed. This parameter is taking into consideration if the case packet variable Finish is false and the case packet variable Cancel is true
rollback_recovered_beans+i	No	Boolean value which indicates if the rollback of the beans restored from the historical has to be performed. This parameter is taking into consideration if the case packet variable Finish is false and the case packet variable Cancel is true
remove_history_beans+i	No	Boolean value which indicates if the beans that has

		been moved to historical must be removed. This parameter is taking into consideration if the case packet variable Finish is true and the case packet variable Cancel is false
unlock_beans+i	No	Boolean value which indicates if the locked beans has to be unlocked.
release_resources+i	No	Boolean value which indicates if the reserved resources must be released. This parameter is taking into consideration if the case packet variable Finish is false and the case packet variable Cancel is true
delete_delayed+i	No	Boolean value which indicates if the delayed beans (See *DelayedDelete* node) must be removed. . This parameter is taking into consideration if the case packet variable Finish is true and the case packet variable Cancel is false
update_delayed+i	No	Boolean value which indicates if the delayed beans must be updated. This parameter is taking into consideration if the case packet variable Finish is true and the case packet variable Cancel is false
release_delayed+i	No	Boolean value which indicates if the delayed reserved beans must be released. This parameter is taking into consideration if the case packet variable Finish is true and the case packet variable Cancel is false
force_rollback+i	No	Boolean value which indicates if the rollback tasks must be performed even though the workflow finishes correctly, ignoring the parameter *finish*.

NOTE: The following parameters of the *WFTransactionHandler* are related to the CCWF, and only have sense if the workflow is a child workflow started using the *StartJobConcurrent* node explaining previously in this document. This handler adds the variables *return_code* and *error_description* to the ones configured with standard values (OK and ERROR depending of the values the case packet variables Cancel and Finish had) when do the synchronization.

Parameter	Mandatory	Description
concurrent_sync	No	Boolean value which indicates if this workflow has to synchronize with its parent workflow using the CCWF. It is done if parameter *finish* is true and cancel is false
concurrent_sync_finish	No	Boolean value which indicates if this workflow has to synchronize with its parent workflow using the CCWF. It is done if parameter *finish* is true and cancel is false
variable+i	No	Variable to send to the parent workflow when synchronizing
destination+i	No	Destination variable of the parent workflow where the value of each synchronized variable has to be stored
remote_ip	No	Remote Ip (needed when synchronization must be done remotely)
local_ip	No	Local Ip (needed when synchronization must be done remotely)

There is another parameter to control if the workflow must answer to SOSA (for complete information about SOSA synchronization read SOSA3 documentation):

Parameter	Mandatory	Description
async_handler	No	Boolean value which indicates if this workflow has to answer SOSA, because is has been executed by it in an asynchronous way.

following parameters of the *WFTransactionHandler* are related to the CCWF, and only have sense if the workflow is a child workflow started using the *StartJobConcurrent* node explaining previously in this document. This handler adds the variables *return_code* and *error_description* to the ones configured with standard values (OK and ERROR depending of the values the case packet variables Cancel and Finish had) when do the synchronization.

13 Audit Action Module

The HPSA Extension Pack includes a new audit system through which the different modules and applications can store information about the activities performed in the System. By default, all the core actions will leave an entry in the audit system. The audit system of the EP tries to give the user a service-oriented view of the activities performed in the system, complementing the audit system of the HPSA.

13.1 Module Configuration

The audit module must be configured in the *mwfm.xml* file.

This module attends the requests from the workflows to access the audit system. The audit system is integrated in the User Management System. If the connectivity with the UMM module is lost, the module stores the requests in a queue until the connectivity is recovered.

There are some parameters that may be configured for this module:

Parameter	Mandatory	Description
audit_error_persistence_file	Yes	Name of the file where the unattended request will be stored. If the file doesn`t exists, it will be created.
max_audit_error_number	Yes	Max number of entries in the file.
retry_timeout	Yes	Time in milliseconds to retry to connect to the UMM module, if the connectivity is lost.
spi_user_management_url	Yes	RMI URL where the SC publishes the remote method for user management. The value for this parameter must be the same as the one specified for the parameter with the same name for the Login Servlet set in the web.xml file. See the SC documentation for more information.

The example below shows how to configure the audit module:

```
<Module>
  <Name>AuditActionModule</Name>
  <Class-Name>
    com.hp.ov.activator.mwfm.ep.engine.module.AuditActionModuleImpl
  </Class-Name>
  <Param name="audit_error_persistence_file"
         value="C:/hp/OpenView/ServiceActivator/var//tmp/error_audit_actions.dat"/>
  <Param name="max_audit_error_number" value="1000"/>
  <Param name="retry timeout" value="5000"/>
  <Param name="spi_user_manager_rmi_url" value="//localhost:2001/userrmi"/>
</Module>
```

13.2 Nodes

The audit module provides one node which can be used to introduce a new entry in the audit system.

13.2.1 AuditAction

The parameters accepted by this node are:

Parameter	Mandatory	Description
messageType	Yes	Type of the audit message (INFO or ERROR).

userName	Yes	Name of the user who performed the action.
sourceComponent	Yes	Name of the affected component.
actionPerformed	Yes	Name of the action performed.
detail	No	Description containing the action details.
processId	No	Name of the process this action belongs to. It can be used to group different actions associated to a certain process.

The following example shows how to invoke the audit node.

```
<Process-Node>
  <Name>Audit example</Name>
  <Description> Audit example</Description>
  <Action>
    <Class-Name>
      com.hp.ov.activator.mwfm.ep.component.builtin.AuditAction
    </Class-Name>
    <Param name="messageType"     value="constant:INFO"/>
    <Param name="userName"        value="constant:jlopez"/>
    <Param name="sourceComponent" value="constant:VPN"/>
    <Param name="actionPerformed" value="constant:configure"/>
    <Param name="detail"          value=”vpnDetails”/>
    <Param name="processId"       value=”vpnProcessId”/>
  </Action>
</Process-Node>
```

14 TMN Inventory

The TMN Inventory is a library that can be used to organize and manage the complete set of networks and equipments of an organization. It is an Inventory Builder created project, that is, a set of XML entities describing the relationships and attributes of the elements involved in the network. These entities are transformed with the IB tool into database tables, and java classes that provide tools to use these entities.

The TMN Inventory comprises many entities, which can be network based: *Network*, *Path*, *TerminationPoint…*; or equipment based: *NetworkElement*, *EquipmentComponent*, *EquipmentFunction*, *Manufacturer…* Each one will be described in detail in the following chapters.

14.1 TMN Inventory Entities

The TMN Inventory is basically a description of a network and its elements and relationships; therefore, we will describe each element in turn, and explain the relations.

14.1.1 Colour

This is one of the simpler entities; it provides an RGB value and its identification for further use.

14.1.2 ElementTypes

It provides a list of possible types of element that can appear in the network. An example element type could be a Router.

14.1.3 EquipmentFunction

It presents a list of possible functions that a piece of equipment may have. An example could be a WIMAX converter.

14.1.4 EquipmentOS

This entity contains all the possible Operating Systems that can be installed in the inventory's system. An example Operating System could be '1.0.2.0 ciscoVersionFile1.0.2.0'

14.1.5 EquipmentStatus

It represents the status of a piece of equipment. The table contains all possible status that a piece of equipment may have. A possible status could be 'Active'.

14.1.6 PathStatus

It provides the list of all possible status a Path may have. A Path status could be AVAILABLE.

14.1.7 Provinces

This is a list of provinces of regions to locate the networks situation in a map.

14.1.8 Location

These are locations that belong to a particular province. For example a city: Madrid.

14.1.9 Manufacturers

These are the names of the manufacturers of the equipment. For example: HP.

14.1.10 Network

This table represents a Network. A Network can belong to another network. It also contains X Y parameters so that it can be located in a map. An example Network could be: 'Jonquera', which belongs to its parent network: 'Telefónica'.

14.1.11 ElementModels

These are the different element models available. Each ElementModel is of a particular ElementType and built by a Manufacturer. For example: RS3000, a model built by Riverstone.

14.1.12 EquipmentFunctionModel

This entity provides a relation between EquipmentFunctions and ElementModels.

14.1.13 EquipmentOSModel

This entity provides a relation between EquipmentOS and ElementModels

14.1.14 NetworkElement

This is perhaps the most important entity in the Inventory. It tries to describe an element inside a Network. This entity has a Name, Description and IP and has relations to the following entities:

- Status, to show the Status of the Element.
- Network to show the Network the Element belongs to.
- Parent Network of this Network.
- Manufacturer, the Manufacturer who built it.
- ElementType, to show the type of this Network Element.
- X and Y axes, to give the position inside a map.
- ElementModel, the element model of this Network Element.
- ElementFunction, the function this element provides.
- Localizaciones, to show the Location where this element resides.
- Internal Function, to show the function of this element inside the network.
- OSVersion, the version of the Operating System.

An example could be a Router inside a network.

14.1.15 ElementComponent

This entity is given a name and status, and provides relations with the NetworkElement it belongs to and, if there are any, to the parent's ElementComponent. Examples of element components could be: 'chassis', 'rack' or network card.

14.1.16 Path

A Path is a representation of the path between two NetworkElements. It contains links to the origin and destination Network Elements and to the PathStatus. A path is a virtual connection.

14.1.17 PathComponent

This entity is a component belonging to a Path, it therefore contains a relation with the Path, an index of the order within the Path whether the jump is loose or strict, and a relation with the Colour table. A path is made up of components, such as a switch or a router.

14.1.18 PathNE

This is a path to a Network Element.

14.1.19 TerminationPointID

This is an endpoint of an ElementComponent or NetworkElement. It has a name and links to the Network and Element Component it belongs to. It also has a description of its use, and a relation with its parent TerminationPointID. An example could be a Port.

14.1.20 TMNConnection

This entity describes a physical connection. It provides relations between the network of origin and the destination network, of the NetworkElement of origin and also destination, and finally the TrminationPointIDs of origin and destination. It is also related to a Path, has X-Y coordinates. An example is a real physical connection.

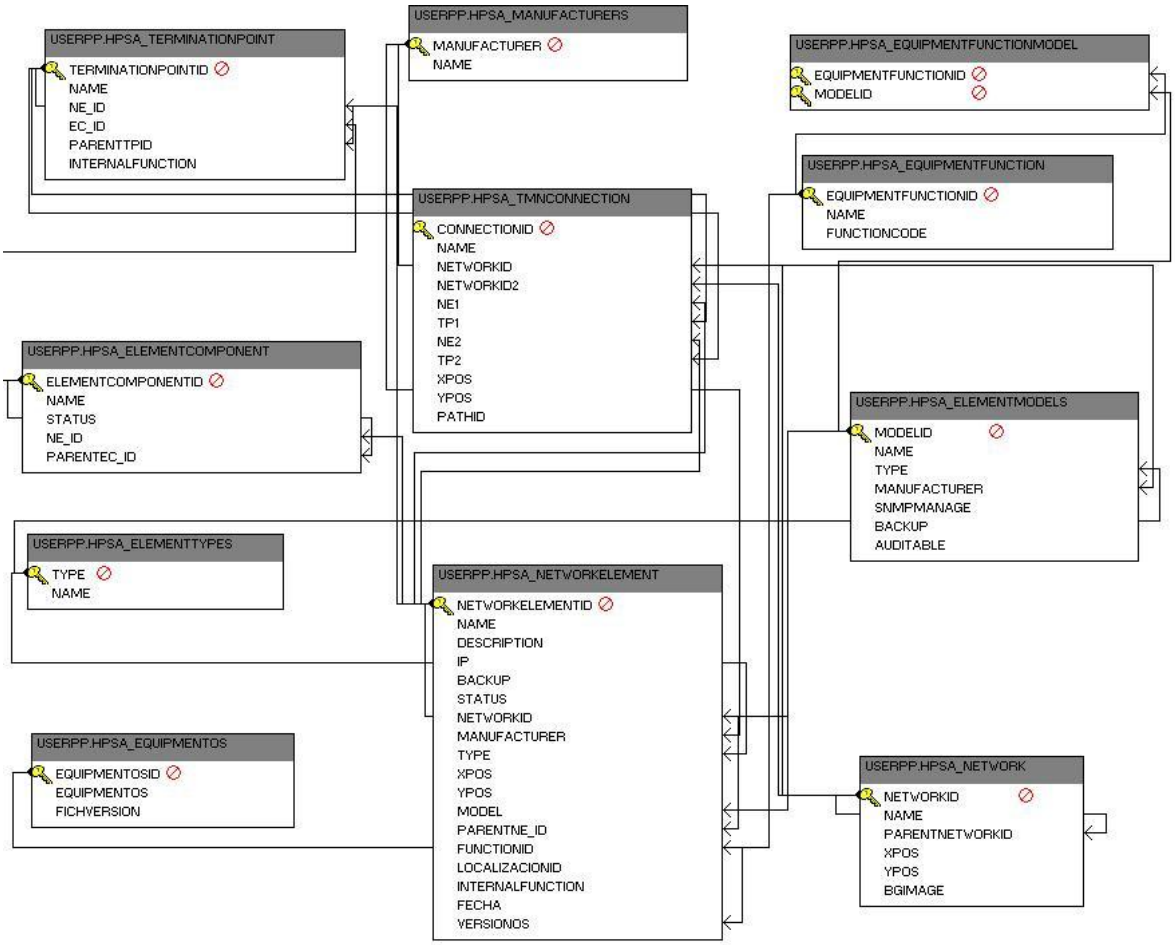
14.1.21 PathConnection

It provides a relation to a TMNConnection; and therefore ties a virtual connection to the physical connection that supports it.

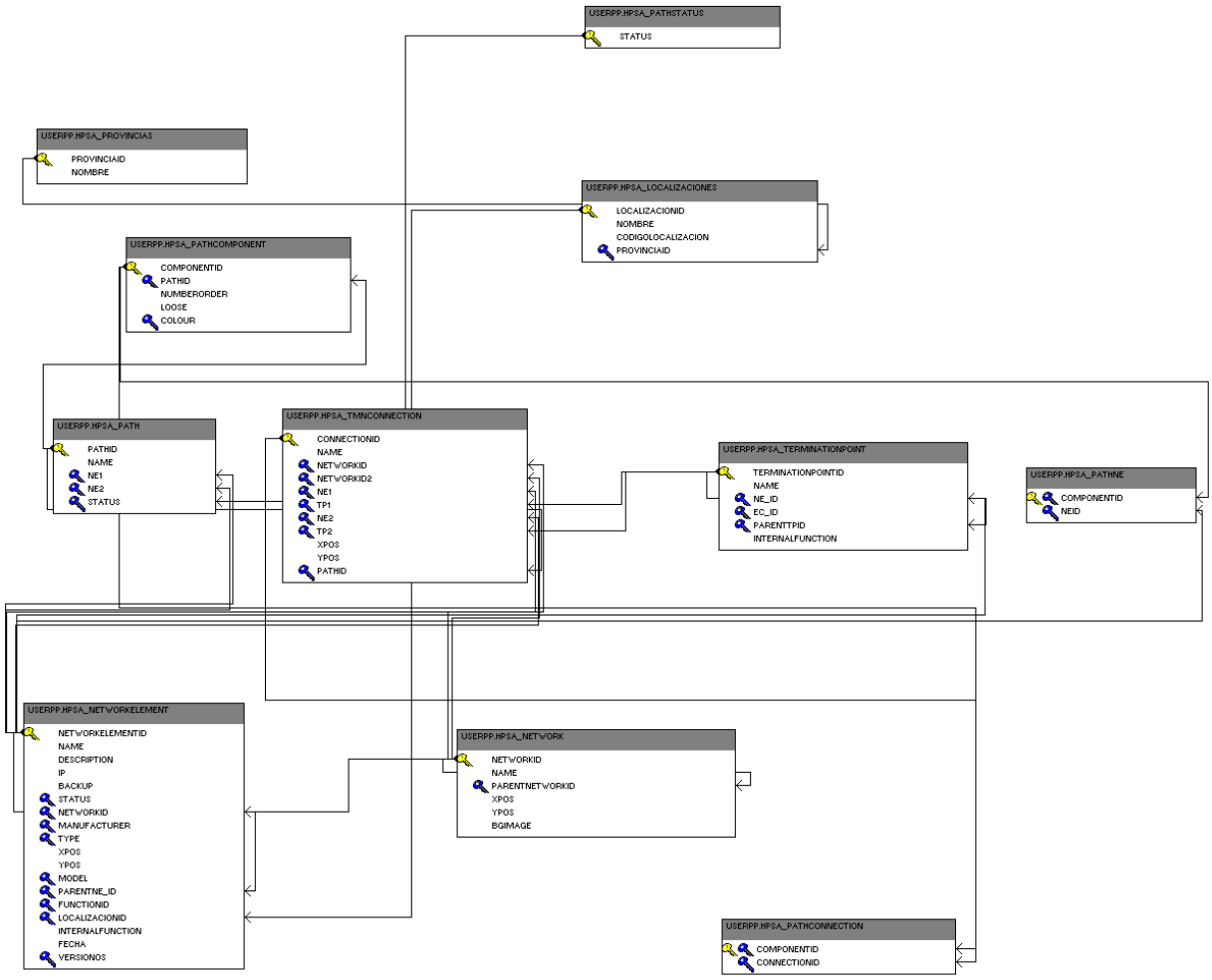
14.2 TMN Inventory Structure

The structure will be shown with a subset of the classes, leaving the location and path entities out of the diagram, to better illustrate the Network entities.

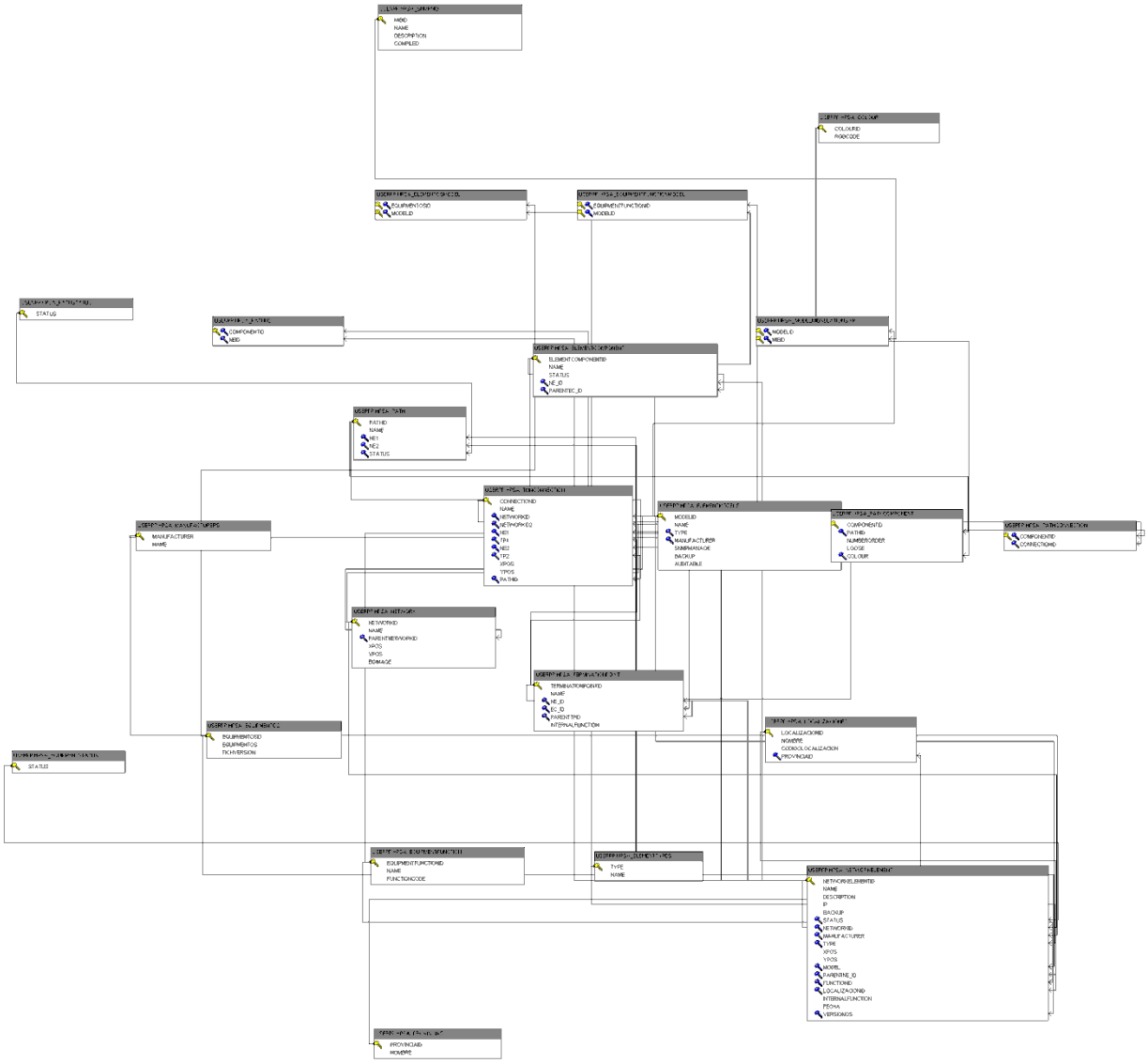
14.2.1 Network Entities Diagram



14.2.2 Path Diagram



14.2.3 Full Diagram



15 SNMP Tool

SnmpTool(*) is an application that manages Snmp requests to Network Elements. It consists of a web interface which is used to manage the MIB files and turns them into easily accessible and useful properties (allowing the user to create sets of favourites), and HPSA nodes and plugins to manage the network elements via SNMP.

* This application uses the library Westhawk's SNMP stack.

15.1 SNMP and MIB background

SNMP (Simple Network Management Protocol) is a protocol built for network management technology. SNMP defines a universal way that management information can be easily defined for any object and then exchanged between that object and a device designed to facilitate network management. Each device that participates in network management using SNMP runs a piece of software, generically called an *SNMP entity*.

The SNMP entity is responsible for implementing all of the various functions of the SNMP protocol. It comprises of two main parts, the SNMP Agent, which is a software program that implements the protocol, and sends and receives requests

Secondly, the SNMP Management Information Base (MIB) which defines the types of information stored about the node that can be collected and used to control the managed node. Information exchanged using SNMP takes the form of objects from the MIB. The MIB is written using SMI. Each managed device contains a set of variables that are used to manage it. These variables represent information about the operation of the device. The MIB is the full set of these variables that describe the management characteristics of a particular type of device.

15.1.1 Snmp version

There are several different SNMP versions that have appeared as the standard has evolved. The first two are basically similar, the difference being mostly in the use of resources, v2 being much more efficient. SNMP version 3 addresses the problem of security. The previous versions sent the requests in clear text, and having as only protection against unauthorized usage a community name, shared between all clients, that has to match the request, and gives read or read-write permissions to the client.

Version 3 is built with security issues in mind. It provides support for authentication and privacy, using any number of encryption standards. This Tool is programmed to use authentication if needed, which can be used with SHA1 or MD5 encryption algorithms and can also use privacy.

Bear in mind that the target of the request has to be configured with the proper values to the requests, and read/write permissions have to be allocated to the users.

15.1.2 TMN Inventory

The SnmpTool uses the TmnInventory project extensively. The TmnInventory provides all the information about a Network, and describes all the elements involved inside a Network. The SnmpTool expects the TmnInventory to be installed and needs to have access to its Database and its classes to access these entities. The database itself needs to be properly set up with network elements, models and manufacturers in order to send snmp requests to the network elements. For further information about the TmnInventory please read the document 'OVSA SPI for Service Providers – TmnInventory – User Reference'.

15.2 SNMP nodes

15.2.1 General Introduction

The SNMPTool Package includes a node which provides access to the *RemoteSnmpTool*, allowing inclusion of SNMP operations as part of a workflow.

The operations implemented by the node do not provide access to the full *RemoteSnmpTool*. Only the *SnmpRequest* methods are accessible. On the other hand, bulk properties operations are provided.

Some of the node operations rely on the files generated through MIB compilation processes. Additionally, helper nodes are provided to extract information from the favourites files which can be then used as input to the SNMP Node.

15.2.2 Node Class

```
com.hp.spain.node.GenericSnmpRequester
```

15.2.3 Functionality

The Node provides four main operations:

a) Reading SNMP properties values
b) Setting a specific set of properties values
c) Comparing two specific sets of properties values and setting the new or modified ones
d) Setting properties values to their reset values

15.2.4 Parameter Formats

The Node allows performing bulk operations, reading or setting multiple variables in a single process node. To allow those operations the following formats are used:

15.2.5 String Bulk Parameters

Parameters of this type contain SNMP properties as a `HashMap` with the following structure:

- Keys: Must always be a `Vector`, whose first position will contain the SNMP property name (as a `String`) and second position the OID (as a `String`), that is, entry at index zero contains the SNMP Object label and entry at index 1 contains the SNMP Object OID, both as `String`. Can't be null, and must have 2 elements which must be populated.
- Values: Values structure may vary, depending on whether the SNMP Objects are indexed or not.
    - If the SNMP properties are scalar, values must be `String`.
    - If the SNMP properties are tabular, values must be a `HashMap` instance where:
        - Keys: must be the tabular property value Index as a `String`.
        - Values: must be the value corresponding to that Index as a `String`.

The node parameter `index` will determine whether the String Bulk Parameters will be treated as Scalar or Tabular SNMP Properties.

15.2.5.1 SnmpProperty Bulk Parameters

Parameters of this type contain SNMP properties as a HashMap with the following structure:

- Keys: Must always be a Vector, whose first position will contain the SNMP property name (as a String) and second position the OID (as a String), that is, entry at index zero contains the SNMP Object label and entry at index 1 contains the SNMP Object OID, both as String. Can't be null, and must have 2 elements which must be populated.
- Values: Values structure may vary, depending on whether the SNMP Objects are indexed or not.
    - If the SNMP properties are scalar, values must be SnmpProperty instance.
    - If the SNMP properties are indexed, values must be a HashMap instance where:
        - Keys: must be the tabular property value Index as a String.
        - Values: must be the value corresponding to that Index as a SnmpProperty instance.

The node parameter index will determine whether the SnmpProperty Bulk Parameters will be treated as Scalar or Tabular SNMP Properties.

15.2.6 SNMP Versions

As the RemoteSnmpTool, the SNMP Node is able to handle SNMPv1, SNMPv2c and SNMPv3 PDUs.

To indicate the version of the PDU to send, the version parameter may be included. Depending on the version value different additional parameters must be included.

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
Version	no	yes	1	n/a	n/a	no

- *version*: Version of the PDU to send (may be 1, 2 or 3). If this parameter value is 3, the SNMPv3 authentication/privacy parameters must be used. In other case, the read and write community must be used.

15.2.6.1 SNMPv1 and SNMPv2c

All Node Actions must include an additional set of parameters if SNMPv1 or SNMPv2c PDUs are desired.

These parameters will be ignored if the version parameter value is not 1 or 2.

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
read_community	Yes	yes	n/a	String	n/a	yes if version<3
write_community	Yes	yes	n/a	String	n/a	yes if version<3

- *read_community*: Community name on behalf of which the GET or GET-NEXT PDU is sent.
- *write_community*: Community name on behalf of which the SET PDU is sent.

15.2.6.2 SNMPv3

The Node allows performing authenticated and encrypted requests, as described by the SNMP v3 specification. All Node Actions may include an additional set of parameters if SNMPv3 PDU is desired.

These parameters will be ignored if the version parameter value is not 3.

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
Username	yes	yes	n/a	String	n/a	yes if version==3
use_authentication	yes	yes	false	String	n/a	no
Password	no	yes	n/a	String	n/a	yes if use_authentication ==true
Authprotocol	yes	yes	false	String	n/a	no
use_privacy	yes	yes	n/a	String	n/a	no
priv_passwd	yes	yes	n/a	String	n/a	yes if use_privacy==true

- *username*: the user name on behalf of which the Node will operate.
- *use_authentication*: Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in *java.lang.Boolean.valueOf(Sting)*.
- *password*: The password for authentication.
- *authprotocol*: Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1.
- *use_privacy*: Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in *java.lang.Boolean.valueOf(Sting)*.
- *priv_passwd*: The privacy password.

15.2.7 Get Action

This operation will send a GET or GET-NEXT PDU for a specified set of properties. The node parameters must specify a MIB compilation file and a filter to determine the properties in the file to process. The following table describes the node parameters.

15.2.7.1 Get Action Parameters

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
action	yes	yes	n/a	String	n/a	yes
snmp_tool	no	yes	n/a	n/a	n/a	yes
hostname	yes	yes	n/a	String	n/a	yes
snmpPropertiesFile	yes	no	no	String	n/a	yes
branch	yes	yes	no	String	n/a	no
property	yes	yes	no	String	n/a	no
array_properties	yes	no	no	Object	String []	no
index	yes	yes	false	Boolean	n/a	no
index_value	yes	yes	no	String	n/a	no
vector_index	no	yes	no	Object	Vector of String	no
snmpProperties	yes	no	no	Object	SnmpProperty Bulk Parameter	n/a

Input Parameters:

- *action*: Always "GET".
- *snmp_tool*: The name under which the SNMP Tool Module to use is registered.
- *hostname*: SNMP Agent IP/hostname to send the PDU to.
- *snmpPropertiesFile*: The path to the compiled MIB file whose properties will be read.
- *branch*: Name of the branch of properties in the compiled MIB file to read.
- *property*: Name of the property in the in the compiled MIB file to read.
- *array_properties*: List of properties in the compiled MIB file to read.
- *index*: Whether the properties in the compiled MIB file should be treated as indexed or not.
- *index_value*: Index of the value to read.
- *vector_index*: List of Index values to read for each property.

Output Parameters

- *snmpProperties*: Context variable where the read variables will be returned as a SnmpProperty Bulk Parameter.

15.2.7.2 Get Action Functionality

The Get Action Node operation will perform one or more GET o GET-NEXT PDU to the SNMP Agent in the host specified by hostname using read_community as community. The version of the PDU will be determined by the version parameter.

15.2.7.3 Indexed Properties

The index parameter will determine whether the properties should be treated as indexed properties or not.

c. Properties and Values Affected

The Node will filter which of the properties present in the snmpPropertiesFile will be requested to the SNMP Agent, depending on the parameters branch, property, array_properties and index:

- If the parameter array_properties is not present.
    - If property is present
        - If it is a scalar property, that property will be requested.
        - If it is an indexed property, that property and all its sibling properties will be requested.
    - If property is not present but branch is, all the properties pertaining to that MIB branch will be requested.
    - If neither branch neither property is present, all the properties in the compiled MIB file will be requested.
- If the parameter array_properties is present, all the properties in the array will be requested, including their siblings if the properties are indexed.

d. GET-NEXT PDU

If the properties are indexed, but the parameter vector_index is not received, an SNMP Tool getVars will be performed for each property.

e. GET PDU

If the properties are indexed, and the parameter vector_index is received, an SNMP Tool get will be performed for each index it contains for all the properties.

If the properties are not indexed, but the parameter index_value is received, an SNMP Tool get will be performed for that index value for all the properties.

If the properties are not indexed, and the parameter vector_index is not received, an SNMP Tool get will be performed for all the properties.

f. Output

As a result, a SnmpProperty Bulk Parameter will be returned through the snmpProperties context attribute, containing the SNMP Agent answers.

If no error is encountered, 0 will be the Node execution return value.

If any error is encountered -1 will be the Node execution return value. A String describing the error will be returned in error_description context attribute.

15.2.8 Update Action

This operation will send a SET PDU for a specified set of properties. The node parameters must specify the properties and values to set. The following table describes the node parameters.

15.2.8.1 Update Action Parameters

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
action	yes	yes	n/a	String	n/a	yes
snmp_tool	no	yes	n/a	n/a	n/a	yes
hostname	yes	yes	n/a	String	n/a	yes
read_community	yes	yes	n/a	String	n/a	yes
write_community	yes	yes	n/a	String	n/a	yes
snmpProperties	yes	no	no	Object	String Bulk Parameter	no
index	yes	yes	false	Boolean	n/a	no

Input Parameters:

- *action*: Always “UPDATE”.
- *snmp_tool*: The name under which the SNMP Tool Module to use is registered.
- *hostname*: SNMP Agent IP/hostname to send the PDU to.
- *read_community*: Community name on behalf of which the GET or GET-NEXT PDU is sent.
- *write_community*: Community name on behalf of which the SET PDU is sent.
- *snmpProperties*: String Bulk Parameter containing the properties and their values to set.

- *index*: Whether the properties in snmpProperties parameter should be treated as indexed or not.

15.2.8.2 Update Action Functionality

The Update Action Node operation will perform one or more SET PDU to the SNMP Agent in the host specified by hostname using write_community as community. The version of the PDU will be determined by the version parameter.

a. Indexed Properties

The index parameter will determine whether the properties should be treated as indexed properties or not.

b. Properties and Values Affected

All the properties present in the String Bulk Parameter in the snmpProperties value with a neither null nor empty value will be affected by the set.

c. SET PDU

If the properties are indexed an SNMP Tool set will be performed for each index value for each property. If the properties are not indexed an SNMP Tool set will be performed for each value of each property.

d. Output

If no error is encountered, 0 will be the Node execution return value. If some error is encountered while setting a property, a String containing the property names of the failed properties will be returned in the update_snmp_error context attribute. If the properties are not indexed will have de format:

```
\t\t\Property1Name\n
…
\t\t\Property2Name
```

If the properties are not indexed will have de format:

```
\t\t\Property1Name\n
…
\t\t\Property2Name
```

If any other error is encountered -1 will be the Node execution return value. A String describing the error will be returned in error_description context attribute. Additionally, if any property set has failed, the update_snmp_error context attribute will be set, as described before.

15.2.9 Reset Action

This operation will send a SET PDU for the properties specified in a compiled MIB file which specify a reset_value. The following table describes the node parameters.

15.2.9.1 Reset Action Parameters

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
action	yes	yes	n/a	String	n/a	yes
snmp_tool	no	yes	n/a	n/a	n/a	yes
hostname	yes	yes	n/a	String	n/a	yes
read_community	yes	yes	n/a	String	n/a	yes
write_community	yes	yes	n/a	String	n/a	yes

snmpPropertiesFile	yes	no	no	String	n/a	yes
index	yes	yes	false	Boolean	n/a	no
index_value	yes	yes	no	String	n/a	if index is true

Input Parameters:

- *action*: Always "RESET".
- *snmp_tool*: The name under which the SNMP Tool Module to use is registered.
- *hostname*: SNMP Agent IP/hostname to send the PDU to.
- *read_community*: Community name on behalf of which the GET or GET-NEXT PDU is sent.
- *write_community*: Community name on behalf of which the SET PDU is sent.
- *snmpPropertiesFile*: The path to the compiled MIB file whose properties will be set.
- *index*: Whether the properties in the compiled MIB file should be treated as indexed or not.
- *index_value*: Index of the value to reset.

15.2.9.2 Reset Action Functionality

The Update Action Node operation will perform one or more SET PDU to the SNMP Agent in the host specified by hostname using write_community as community. The version of the PDU will be determined by the version parameter.

a. Indexed Properties

The index parameter will determine whether the properties should be treated as indexed properties or not.

b. Properties and Values Affected

All the properties which include a reset_value in the compiled MIB file. Additionally, if the properties are indexed and index_value is specified, only that value of the indexed properties will be reset.

c. SET PDU

If the properties are indexed an SNMP Tool set will be performed for the index value indicated by index_value. If the properties are not indexed an SNMP Tool set will be performed for each value of each property.

d. Output

If no error is encountered, 0 will be the Node execution return value.

If any error is encountered -1 will be the Node execution return value. A String describing the error will be returned in error_description context attribute.

15.2.10 Set Action

This operation will send a SET PDU for the properties which differ between two sets of properties values. The following table describes the node parameters.

15.2.10.1 Set Action Parameters

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
action	yes	yes	n/a	String	n/a	yes
snmp_tool	no	yes	n/a	n/a	n/a	yes
hostname	yes	yes	n/a	String	n/a	yes
read_community	yes	yes	n/a	String	n/a	yes
write_community	yes	yes	n/a	String	n/a	yes
snmpProperties	yes	no	no	Object	SnmpProperty Bulk Parameter	yes
snmpProperties_old	yes	no	no	Object	SnmpProperty Bulk Parameter	yes
index	yes	yes	false	Boolean	n/a	no
index_value	yes	yes	no	String	n/a	no

Input Parameters:

- *action*: Always "SET".
- *snmp_tool*: The name under which the SNMP Tool Module to use is registered.
- *hostname*: SNMP Agent IP/hostname to send the PDU to.
- *read_community*: Community name on behalf of which the GET or GET-NEXT PDU is sent.
- *write_community*: Community name on behalf of which the SET PDU is sent.
- *snmpProperties*: String Bulk Parameter containing the properties and their values to set.
- *snmpProperties_old*: String Bulk Parameter containing the properties and their values to set.
- *index*: Whether the properties in the compiled snmpProperties and snmpProperties_old should be treated as indexed or not.
- *index_value*: Index of the value to reset.

15.2.10.2 Set Action Functionality

The Update Action Node operation will perform one or more SET PDU to the SNMP Agent in the host specified by hostname using write_community as community. The version of the PDU will be determined by the version parameter.

a. Indexed Properties

The index parameter will determine whether the properties should be treated as indexed properties or not.

b. Properties and Values Affected

The property values affected will be those considered different between snmpProperties and snmpProperties_old.

A scalar value will be considered new if its hashmap key can be found in snmpProperties but not in snmpProperties_old.

A scalar value will be considered modified if its corresponding SnmpProperty values in snmpProperties and snmpProperties_old differ.

A tabular value will be considered new if its hashmap key can be found in snmpProperties but not in snmpProperties_old, or if its indexed value key can be found in snmpProperties but not in the corresponding snmpProperties_old element.

A tabular value will be considered modified if its corresponding SnmpProperty values in snmpProperties and snmpProperties_old differ.

c. SET PDU

The properties and values result of the comparing process will be set or created with their new values.

d. Output

If no error is encountered, 0 will be the Node execution return value.

If any error is encountered -1 will be the Node execution return value. A String describing the error will be returned in error_description context attribute.

15.3 Helper Nodes

15.3.1 Favourites Node

15.3.1.1 General Introduction

The Favorites Node eases the use of the SNMP Node, allowing the use of user favorites instead of complex field lists.

15.3.1.2 Node Class

```
com.hp.spain.node.GetSnmpAttributesFromFavorites
```

15.3.1.3 Functionality

Will extract all the properties names included in a favorite and return them as an array which can be then used as array_properties parameter for the SNMP Node Get Action.

15.3.1.4 Parameters

Parameter	Variable Reference	Constant	Default Value	Case Packet Type	Java Type	Mandatory
favoriteFile	yes	yes	n/a	String	n/a	yes
user	yes	yes	n/a	String	n/a	yes
favorite	yes	yes	n/a	String	n/a	yes
array_properties	no	yes	n/a	Object	String []	n/a

Input Parameters:

- *favoriteFile*: The path of the MIB favorites file to be read.
- *user*: The name of the user whose favorite will be read.

- *favorite*: he name of the user favorite to read.

Output Parameters:

- *array_properties*: The context attribute name where the array containing the properties names in the favorite will be stored.

15.3.1.5 Output

As a result, all the property names contained in the favorite specified by favoriteFile, user and favorite will be returned as an String [] in the context attribute named array_properties.

If an error is encountered, the node return value will be -2, and the context attribute error_description will be set to the error description.

If any error is encountered -1 will be the Node execution return value. A String describing the error will be returned in error_description context attribute.

15.4 Nodes Examples

15.4.1 Get of scalar properties in favorite

```
<Workflow Init-On-Startup="false">
  <Name>WFLT SNMPGetFavorite</Name>
  <Description>
    Obtains the values of the set of SNMP Objects defined in a favorite
  </Description>
  <Start-Node>LoadFavouriteSNMPObjectsLabels</Start-Node>
  <Nodes>
    <Process-Node>
      <Name>LoadFavouriteSNMPObjectsLabels</Name>
      <Description>
        Load the Labels of the SNMP Objects defined in a favorite into an array
      </Description>
      <Action>
        <Class-Name>
          com.hp.spain.node.GetSnmpAttributesFromFavorites
        </Class-Name>
        <Param name="favoriteFile"
               value="constant:<<ACTIVATOR ETC>>/config/snmp/RFC1213-MIB favorites.xml"/>
        <Param name="user" value="constant:admin"/>
        <Param name="favorite" value="constant:InterfacesData"/>
      </Action>
      <Next-Node>SNMPRequest</Next-Node>
    </Process-Node>
    <Process-Node>
      <Name>SNMPRequest</Name>
      <Description>Performs an snmp request</Description>
      <Action>
        <Class-Name>com.hp.spain.node.GenericSnmpRequester</Class-Name>
        <Param name="snmp tool" value="snmp tool"/>
        <Param name="hostname" value="constant:127.0.0.1"/>
        <Param name="action" value="constant:GET"/>
        <Param name="read_community" value="constant:public"/>
        <Param name="write_community" value="constant:public"/>
        <Param name="snmpProperties" value="output"/>
        <!-- Populated by LoadFavouriteSNMPObjectsLabels -->
        <Param name="array properties" value="array properties"/>
        <Param name="snmpPropertiesFile value="snmpPropertiesFile"/>
      </Action>
    </Process-Node>
  </Nodes>
  <Error-Handler>
    <Class-Name>com.hp.spain.node.wftransaction.WFTransactionErrorHandler</Class-Name>
  </Error-Handler>
  <End-Handler>
```

```
    <Class-Name>com.hp.spain.node.wftransaction.WFTransactionHandler</Class-Name>
    <Param name="finish" value="Finish"/>
    <Param name="cancel" value="Cancel"/>
    <Param name="async handler" value="constant:true"/>
  </End-Handler>
  <Case-Packet>
    <!-- GetSnmpAttributesFromFavorites-->
    <Variable name="error_description"   type="String"/>
    <Variable name="array_properties"    type="Object"/>
    <!-- GenericSnmpRequester -->
    <Variable name="update snmp error"   type="String"/>
    <Variable name="error_description"   type="String"/>
    <Variable name="output"              type="Object"/>
    <Variable name="hostname"            type="String"/>
    <Variable name="action"              type="String"/>
    <Variable name="read community"      type="String"/>
    <Variable name="write_community"     type="String"/>
    <Variable name="snmpProperties"      type="Object"/>
    <Variable name="array_properties"    type="Object"/>
    <Variable name="snmpPropertiesFile"  type="String"/>
    <Variable name="username"            type="String"/>
    <Variable name="password"            type="String"/>
    <Variable name="authprotocol"        type="String"/>
    <Variable name="priv_passwd"         type="String"/>
    <Variable name="context_engine_id"   type="String"/>
    <Variable name="context_name"        type="String"/>
    <Variable name="use privacy"         type="String"/>
    <Variable name="use authentication"  type="String"/>
    <!-- WFTransactionHandler -->
    <Variable name="Finish"              type="Boolean"/>
    <Variable name="Cancel"              type="Boolean"/>
  </Case-Packet>
  <Initial-Case-Packet>
    <Variable-Value name="snmpPropertiesFile"
                    value="<<ACTIVATOR_ETC>>/config/snmp/RFC1213-MIB_props.xml"/>
  </Initial-Case-Packet>
</Workflow>
```

15.4.2 Get of Indexed properties in favorite with SNMPv3

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Workflow SYSTEM "workflow.dtd">
<Workflow Init-On-Startup="false">
  <Name>WFLT SNMPGetFavouriteTabular</Name>
  <Description>
    Obtains the values of the set of SNMP Objects defined in a favorite
  </Description>
  <Start-Node>CreateIndexesNode</Start-Node>
  <Nodes>
    <Process-Node>
      <Name>CreateIndexesNode</Name>
      <Description>Creates the indexes vector</Description>
      <Action>
        <Class-Name>com.hp.spain.node.CreateNewArray</Class-Name>
        <Param name="variable0"   value="snmpindexes"/>
      </Action>
      <Next-Node>AddIndexesNode</Next-Node>
    </Process-Node>
    <Process-Node>
      <Name>AddIndexesNode</Name>
      <Description>Adds indexes to the vector</Description>
      <Action>
      <Class-Name>com.hp.spain.node.AddToArrayMultiple</Class-Name>
      <Param name="array0"     value="snmpindexes"/>
      <Param name="element0"   value="constant:1"/>
      <Param name="array1"     value="snmpindexes"/>
      <Param name="element1"   value="constant:2"/>
      <Param name="array2"     value="snmpindexes"/>
      <Param name="element2"   value="constant:3"/>
    </Action>
    <Next-Node>LoadFavouriteSNMPObjectsLabels</Next-Node>
    </Process-Node>
```

```
    <Process-Node>
      <Name>LoadFavouriteSNMPObjectsLabels</Name>
      <Description>
        Load the Labels of the SNMP Objects defined in a favorite into an array
      </Description>
      <Action>
        <Class-Name>com.hp.spain.node.GetSnmpAttributesFromFavorites</Class-Name>
        <Param name="favoriteFile"
               value="constant:<<ACTIVATOR_ETC>>/config/snmp/RFC1213-MIB_favorites.xml"/>
        <Param name="user"              value="constant:admin"/>
        <Param name="favorite"          value="constant:Tabular"/>
      </Action>
      <Next-Node>SNMPRequest</Next-Node>
    </Process-Node>
    <Process-Node>
      <Name>SNMPRequest</Name>
      <Description>Performs an snmp request</Description>
      <Action>
        <Class-Name>com.hp.spain.node.GenericSnmpRequester</Class-Name>
        <Param name="snmp_tool"       value="snmp_tool"/>
        <Param name="hostname"              value="constant:16.38.0.136"/>
        <Param name="action"          value="constant:GET"/>
        <!-- "array properties" is populated by LoadFavouriteSNMPObjectsLabels -->
        <Param name="array properties" value="array properties"/>
        <Param name="snmpPropertiesFile" value="snmpPropertiesFile"/>
        <!-- Obtain multiple indexed values for all SNMP Objects -->
        <Param name="index"           value="constant:true"/>
        <!-- "vector index" is populated by CreateIndexesNode, AddIndexesNode  -->
        <Param name="vector index"    value="snmpindexes"/>
        <!-- SNMPv3 Auth and Priv -->
        <Param name="version"            value="3"/>
        <Param name="username"                  value="username"/>
        <Param name="use_authentication" value="use_authentication"/>
        <Param name="password"                  value="password"/>
        <Param name="authprotocol"       value="authprotocol"/>
        <Param name="use_privacy"        value="use_privacy"/>
        <Param name="priv_passwd"        value="priv_passwd"/>
        <!-- OUTPUT -->
        <Param name="snmpProperties"     value="output"/>
      </Action>
    </Process-Node>
  </Nodes>
  <Error-Handler>
    <Class-Name>com.hp.spain.node.wftransaction.WFTransactionErrorHandler</Class-Name>
  </Error-Handler>
  <End-Handler>
    <Class-Name>com.hp.spain.node.wftransaction.WFTransactionHandler</Class-Name>
    <Param name="finish" value="Finish"/>
    <Param name="cancel" value="Cancel"/>
    <Param name="async_handler" value="constant:true"/>
  </End-Handler>
  <Case-Packet>
    <!-- CreateIndexesNode -->
    <Variable name="snmpindexes" type="Object"/>
    <!-- GetSnmpAttributesFromFavorites -->
    <Variable name="error description" type="String"/>
    <Variable name="array properties" type="Object"/>
    <!-- GenericSnmpRequester -->
    <Variable name="hostname" type="String"/>
    <Variable name="action" type="String"/>
    <Variable name="read_community" type="String"/>
    <Variable name="write_community" type="String"/>
    <Variable name="array properties" type="Object"/>
    <Variable name="snmpPropertiesFile" type="String"/>
    <Variable name="username" type="String"/>
    <Variable name="use_authentication" type="String"/>
    <Variable name="password" type="String"/>
    <Variable name="authprotocol" type="String"/>
    <Variable name="use privacy" type="String"/>
    <Variable name="priv_passwd" type="String"/>
    <Variable name="update_snmp_error" type="String"/>
    <Variable name="error description" type="String"/>
     <Variable name="output" type="Object"/>
```

```
    <!-- WFTransactionHandler -->
    <Variable name="Finish" type="Boolean"/>
    <Variable name="Cancel" type="Boolean"/>
  </Case-Packet>
  <Initial-Case-Packet>
    <Variable-Value name="snmpPropertiesFile"
                    value="<<ACTIVATOR_ETC>>/config/snmp/RFC1213-MIB_props.xml"/>
    <Variable-Value name="username" value="usr"/>
    <Variable-Value name="use_authentication" value="true"/>
    <Variable-Value name="password" value="pwdauth"/>
    <Variable-Value name="authprotocol" value="MD5"/>
    <Variable-Value name="use_privacy" value="true"/>
    <Variable-Value name="priv_passwd" value="pwdpriv"/>
  </Initial-Case-Packet>
</Workflow>
```

15.5 SNMP Generic Plug-in

15.5.1 General Introduction

The SNMP Plug-in allows atomic execution of the SNMP Set and Get requests. It also allows to perform bulk Set and Get request atomically, that is, operating atomically over a whole group SNMP variables. Additionally, tabular SNMP variables may be treated as indexed values.

The SNMP Plug-in allows SNMP version 1, version 2c and version 3 operations.

The plugin (SNMPPlugin.par) can be found under the 'OpenView/ServiceActivator/SPI' directory, and it needs to be deployed before it can be used.

15.5.2 Locking Arguments

Parameter: the machine destination of the SNMP request.

15.5.3 Class Name

```
com.hp.spain.plugin.snmp.SNMPPlugin
```

15.5.4 Pre-provisioning tasks

The SNMP Plug-in relies on RemoteSnmpTool Workflow Module to operate. The RemoteSnmpTool Module must be configured prior to use the SNMP Plug-in.

15.5.5 Single Value Atomic Tasks

15.5.5.1 task_SNMPGetUnsec

This task reads an SNMP Scalar Variable value performing an SNMP v1 or SNMP v2c GET request.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Variable OID to read
Version	"1" for an SNMP v1 request or "2" for an SNMP v2c request.
sCommunity	Community name on behalf of which the GET PDU is sent

The value of the requested SNMP Object variable will be returned through the Plug-in Context Data Uploader associated to the key "SNMP_RET".

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.5.2 task_SNMPGetUnsecIndexed

This task reads an SNMP Tabular Variable value performing an SNMP v1 or SNMP v2c GET request which can be treated as an indexed value.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Tabular Variable OID to read. The actual SNMP Value read will be: sObjectOID + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to read. The actual SNMP Value read will be: sObjectOID + "." + sIndex
version	"1" for an SNMP v1 request or "2" for an SNMP v2c request.
sCommunity	Community name on behalf of which the GET PDU is sent

The value of the requested SNMP Object variable will be returned through the Plug-in Context Data Uploader associated to the key "SNMP_RET".

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.5.3 task_SNMPGet

This task reads an SNMP Scalar Variable value performing an SNMP v3 GET request.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Variable OID to read
sUserName	the user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String ("").
sAuthProtocol	Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String ("").
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sUserPrivacyPasswd must be an empty String ("").
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String ("").

sContextEngineID	The Context Engine ID. Can be empty String (“”).
sContextName	The Context Name. Can be empty String (“”).

The value of the requested SNMP Object variable will be returned through the Plug-in Context Data Uploader associated to the key “SNMP_RET”.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.5.4 task_SNMPGetIndexed

This task reads an SNMP Tabular Variable value performing an SNMP v3 GET request which can be treated as an indexed value.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Tabular Variable OID to read. The actual SNMP Value read will be: sObjectOID + “.” + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to read. The actual SNMP Value read will be: sObjectOID + “.” + sIndex
sUserName	the user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If “false”, sPasswd and sAuthProtocol must be empty Strings (“”).
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String (“”).
sAuthProtocol	Protocol to be used for authentication. “MD5” for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String (“”).
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If “false”, sUserPrivacyPasswd must be an empty String (“”).
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String (“”).
sContextEngineID	The Context Engine ID. Can be empty String (“”).
sContextName	The Context Name. Can be empty String (“”).

The value of the requested SNMP Object variable will be returned through the Plug-in Context Data Uploader associated to the key “SNMP_RET”.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.5.5 task_SNMPSetUnsec

This task sets an SNMP Scalar Variable value performing an SNMP v1 or SNMP v2c SET request. The current value of the variable to set will be read and stored as part of the persistent Transaction State before it is modified. If a Transaction must be rolled back, the stored previous value will be set again, restoring the SNMP variable previous state.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Variable value OID to set
sValue	The value to set the SNMP Variable to.
version	"1" for an SNMP v1 request or "2" for an SNMP v2c request.
sCommunity	Community name on behalf of which the SET PDU is sent

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variable value the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can't be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.5.6 task_SNMPSetUnsecIndexed

This task sets an SNMP Tabular Variable value performing an SNMP v1 or SNMP v2c SET request which can be treated as an indexed value. The current value of the variable to set will be read and stored as part of the persistent Transaction State before it is modified. If a Transaction must be rolled back, the stored previous value will be set again, restoring the SNMP variable previous state.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Tabular Variable OID to set. The actual SNMP Value set will be: sObjectOID + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to set. The actual SNMP Value set will be: sObjectOID + "." + sIndex
sValue	The value to set the SNMP Variable to.
version	"1" for an SNMP v1 request or "2" for an SNMP v2c request.
sCommunity	Community name on behalf of which the SET PDU is sent

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variable value the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can't be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.5.7 task_SNMPSet

This task sets an SNMP Scalar Variable value performing an SNMP v3 SET request. The current value of the variable to set will be read and stored as part of the persistent Transaction State before it is modified. If a Transaction must be rolled back, the stored previous value will be set again, restoring the SNMP variable previous state.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Variable value OID to set
sValue	The value to set the SNMP Variable to.
sUserName	The user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String ("").
sAuthProtocol	Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String ("").
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sUserPrivacyPasswd must be an empty String ("").
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String ("").
sContextEngineID	The Context Engine ID. Can be empty String ("").
sContextName	The Context Name. Can be empty String ("").

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variable value the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can't be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.5.8 task_SNMPSetIndexed

This task sets an SNMP Tabular Variable value performing an SNMP v3 SET request which can be treated as an indexed value. The current value of the variable to set will be read and stored as part of the persistent Transaction State before it is modified. If a Transaction must be rolled back, the stored previous value will be set again, restoring the SNMP variable previous state.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname to send the PDU to.
sObjectOid	The SNMP Tabular Variable OID to set. The actual SNMP Value set will be: sObjectOID + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to set. The actual SNMP Value set will be: sObjectOID + "." + sIndex
sValue	The value to set the SNMP Variable to.
sUserName	the user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String

	(“”).
sAuthProtocol	Protocol to be used for authentication. “MD5” for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String (“”).
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If “false”, sUserPrivacyPasswd must be an empty String (“”).
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String (“”).
sContextEngineID	The Context Engine ID. Can be empty String (“”).
sContextName	The Context Name. Can be empty String (“”).

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variable value the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can’t be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.6 Multiple Value Atomic Tasks

15.5.6.1 task_SNMPMultipleGetUnsec

This task reads multiple SNMP Scalar Variables values performing SNMP v1 or SNMP v2c GET requests.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Variables to read in the format specified by the file MultipleSNMPVars.dtd. See 15.5.7.1 MultipleSNMPVars.dtd
Version	“1” for SNMP v1 requests or “2” for SNMP v2c requests.
sCommunity	Community name on behalf of which the GET PDUs are sent

The values of the requested SNMP Object variables will be returned through the Plug-in Context Data Uploader associated to the key “SNMP_RET” as an XML String in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.6.2 task_SNMPMultipleGetUnsecIndexed

This task reads multiple SNMP Tabular Variables values performing SNMP v1 or SNMP v2c GET requests which can be treated as an indexed value.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Tabular Variables to read in the format specified by the file MultipleSNMPVars.dtd. See 15.5.7.1 MultipleSNMPVars.dtd. The actual SNMP Values

	read will be: <OID present in XML> + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to read. The actual SNMP Value read will be: <OID present in XML> + "." + sIndex
Version	"1" for SNMP v1 requests or "2" for SNMP v2c requests.
sCommunity	Community name on behalf of which the GET PDUs are sent

The values of the requested SNMP Object variables will be returned through the Plug-in Context Data Uploader associated to the key "SNMP_RET" as an XML String in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.6.3 task_SNMPMultipleGet

This task reads multiple SNMP Scalar Variables values performing SNMP v3 GET requests.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Variables to read in the format specified by the file MultipleSNMPVars.dtd. See 15.5.7.1 MultipleSNMPVars.dtd
sUserName	The user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String ("").
sAuthProtocol	Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String ("").
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sUserPrivacyPasswd must be an empty String ("").
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String ("").
sContextEngineID	The Context Engine ID. Can be empty String ("").
sContextName	The Context Name. Can be empty String ("").

The values of the requested SNMP Object variables will be returned through the Plug-in Context Data Uploader associated to the key "SNMP_RET" as an XML String in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.6.4 task_SNMPMultipleGetIndexed

This task reads multiple SNMP Tabular Variables values performing SNMP v3 GET requests which can be treated as an indexed value.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Tabular Variables to read in the format specified by the file MultipleSNMPVars.dtd. See 15.5.7.1 MultipleSNMPVars.dtd. The actual SNMP Values read will be: <OID present in XML> + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to read. The actual SNMP Value read will be: <OID present in XML> + "." + sIndex
sUserName	The user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String ("").
sAuthProtocol	Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String ("").
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sUserPrivacyPasswd must be an empty String ("").
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String ("").
sContextEngineID	The Context Engine ID. Can be empty String ("").
sContextName	The Context Name. Can be empty String ("").

The values of the requested SNMP Object variables will be returned through the Plug-in Context Data Uploader associated to the key "SNMP_RET" as an XML String in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT.

15.5.6.5 task_SNMPMultipleSetUnsec

This task sets multiple SNMP Scalar Variables values performing SNMP v1 or SNMP v2c SET requests. Each individual current value of the variables to set will be read and stored as part of the persistent Transaction State before each variable is modified. If a Transaction must be rolled back or the task fails, the stored previous values will be set again, restoring the SNMP variables previous states.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Variables and values to set in the format specified by the

	file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd.

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variables values the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can't be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.6.6 task_SNMPMultipleSetUnsecIndexed

This task sets an SNMP Tabular Variable value performing an SNMP v1 or SNMP v2c SET request which can be treated as an indexed value. Each individual current value of the variables to set will be read and stored as part of the persistent Transaction State before each variable is modified. If a Transaction must be rolled back or the task fails, the stored previous values will be set again, restoring the SNMP variables previous states.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Tabular Variables and values to set in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd. The actual SNMP Values set will be: <OID present in XML> + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to set. The actual SNMP Value set will be: <OID present in XML> + "." + sIndex
version	"1" for SNMP v1 requests or "2" for SNMP v2c requests.
sCommunity	Community name on behalf of which the SET PDUs are sent

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variables values the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can't be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.6.7 task_SNMPMultipleSet

This task sets an SNMP Scalar Variable value performing an SNMP v3 SET request. Each individual current value of the variables to set will be read and stored as part of the persistent Transaction State before each variable is modified. If a Transaction must be rolled back or the task fails, the stored previous values will be set again, restoring the SNMP variables previous states.

Parameters

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Variables and values to set in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd.
sUserName	The user name on behalf of which the Node will operate.

useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String ("").
sAuthProtocol	Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String ("").
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sUserPrivacyPasswd must be an empty String ("").
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String ("").
sContextEngineID	The Context Engine ID. Can be empty String ("").
sContextName	The Context Name. Can be empty String ("").

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variables values the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can't be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.6.8 task_SNMPMultipleSetIndexed

This task sets an SNMP Tabular Variable value performing an SNMP v3 SET request which can be treated as an indexed value. Each individual current value of the variables to set will be read and stored as part of the persistent Transaction State before each variable is modified. If a Transaction must be rolled back or the task fails, the stored previous values will be set again, restoring the SNMP variables previous states.

Parameters:

Parameter	Description
sMachine	SNMP Agent IP/hostname destination of the requests.
sObjectsXML	An XML String containing the SNMP Tabular Variables and values to set in the format specified by the file MultipleSNMPValues.dtd. See 15.5.7.2 MultipleSNMPValues.dtd. The actual SNMP Values set will be: <OID present in XML> + "." + sIndex
sIndex	The OID of the SNMP Tabular Variable Value to set. The actual SNMP Value set will be: <OID present in XML> + "." + sIndex
sUserName	The user name on behalf of which the Node will operate.
useAuthentication	Whether the authentication facilities of the SNMPv3 should be used. Its value will be interpreted as described in java.lang.Boolean.valueOf(Sting). If "false", sPasswd and sAuthProtocol must be empty Strings ("").
sPasswd	The password for authentication. If authentication facilities are not used, must be empty String ("").
sAuthProtocol	Protocol to be used for authentication. "MD5" for MD5 protocol. Any other value for SHA1. If authentication facilities are not used, must be empty String ("").
usePrivacy	Whether the privacy facilities of the SNMPv3 should be used. Its value will be interpreted as

	described in java.lang.Boolean.valueOf(Sting). If “false”, sUserPrivacyPasswd must be an empty String (“”).
sUserPrivacyPasswd	The privacy password. If privacy facilities are not used, must be empty String (“”).
sContextEngineID	The Context Engine ID. Can be empty String (“”).
sContextName	The Context Name. Can be empty String (“”).

This task does not return any Context data.

If the atomic task is successful, the ExecutionDescriptor returned has majorCode = OK and minorCode = NONE. Otherwise, if the Do/Undo operation does not change the current SNMP Variables values the ExecutionDescriptor returned has majorCode = ERROR and minorCode = CONSISTENT. If the Do/Undo operation result can’t be determined, ExecutionDescriptor returned has majorCode = ERROR and minorCode = INCONSISTENT.

15.5.7 Files

15.5.7.1 MultipleSNMPVars.dtd

This file contains the specification for an XML file containing a set of SNMP Object Variables OID.

```
<!--
  ============================================================================
  SNMP Variables for HP OV Service Activator SNMP Plug-in
  ============================================================================
  Copyright (c) 2001-2004 Hewlett-Packard Company. All Rights Reserved
  ============================================================================
-->

<!--
  ** Mandatory root tag
-->
<!ELEMENT SNMPObjects (SNMPObjectInstance*)>

<!--
  ** SNMP Object Instance descriptor element
-->
<!ELEMENT SNMPObjectInstance   (OID)>

<!--
  ** OID of the corresponding SNMP Object Instance element
-->
<!ELEMENT OID (#PCDATA)>
```

15.5.7.2 MultipleSNMPValues.dtd

This file contains the specification for an XML file containing a set of SNMP Object Variables OID and their values.

```
<!--
  ============================================================================
  SNMP Variables for HP OV Service Activator SNMP Plug-in
  ============================================================================
  Copyright (c) 2001-2004 Hewlett-Packard Company. All Rights Reserved
  ============================================================================
-->

<!--
  ** Mandatory root tag
-->
<!ELEMENT SNMPObjects (SNMPObjectInstance*)>

<!--
  ** SNMP Object Instance descriptor element
-->
```

```
<!ELEMENT SNMPObjectInstance   (OID,Value)>

<!--
  ** OID of the corresponding SNMP Object Instance element
-->
<!ELEMENT OID (#PCDATA)>

<!--
  ** Value of the corresponding SNMP Object Instance element
-->
<!ELEMENT Value (#CDATA)>
```

16 Configuration Management

This chapter explains how to implement a backup driver for the Configuration Management solution. A backup driver is an application that knows how to connect to certain type of equipments and make a backup of their configuration.

16.1 Configuring the memory types

The equipment backups are literally a copy of the status of the memory of the equipment. The problem is that there are equipments that have more than one type of memory. To solve this two database tables are created:

HPSA_MemoryType: Stores the different type of memories.

HPSA_ModelMemTypeRel: Relates them to given equipment.

16.1.1 HPSA_MemoryType

The fields of this table are:

- *MEMORYID*: Unique id identifying the memory.
- *MEMORYNAME*: Name of the memory.
- *FILENAME*: It is not mandatory and can be use by the developer to get the root of the name of the file to transfer with the equipment.

16.1.2 HPSA_ModelMemTypeRel

The fields of this table are:

- *MODELID*: Same Id of the MODELID field of the HPSA_ElementModel of the equipment.
- *MEMORYID*: Same Id of the MEMORYID field of the HPSA_MemoryType of the memory.

This way the Configuration Management will know which memories can be backup for every equipment model.

16.2 Parameter management

For a successful connection for a backup usually several parameters are required e.g. login, password or something more complex like a template with the parameters that will be executed in the equipment for the backup creation.

16.2.1 Configuring parameters

The parameters are defined in a file called *deviceRegister.xml* that is located in the directory

<<ACTIVATOR_ETC>>/config/confMng

A *deviceRegister.dtd* file for syntax validation is located in the same directory.

A device is defined by the next parameters:

Bean: the Java bean implementing the device.

Manufacturer: the device's manufacturer.

Type: the device's type.

Parameters: eventual parameters needed for the access to the equipment. The value of each parameter is retrieved invoking to the method set as its value and which must be implemented in the Java bean.

Properties: eventual properties needed for the access to the equipment. The value of each property is the literal text typed as the value.

Even though these parameters aren't mandatory, there are four of them which are treated in a special way if defined: *login*, *password*, *logintacacs*, *passwordtacacs* and *enablePassword*.

An example of this file is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Devices SYSTEM "deviceRegister.dtd">
<Devices>

  <Device>
    <Bean>com.hp.spain.inventory.tutorial.Equipment</Bean>
    <manufacturer>Riverstone</manufacturer>
    <type>Switch</type>
    <Params>
      <Param name="login"          value="getUsername"/>
      <Param name="password"       value="getPassword"/>
      <Param name="logintacacs"    value="getLogintacacs"/>
      <Param name="passwordtacacs" value="getPasswordtacacs"/>
      <Param name="enablePassword" value="getPasswordenable"/>
      <Param name="ROUTER_NAME"    value="getName"/>
    </Params>
    <Properties>
      <Property name="TEMPLATE"
value="RIVERSTONE_SAVE_CONFIG.vm"/>
      <Property name="SEND_TEMPLATE"
value="RIVERSTONE_MANUAL_LOAD_CONFIG.vm"/>
      <Property name="HPIA_HOSTNAME"      value="localhost"/>
      <Property name="TEMPLATE_PATH"
value="<<ACTIVATOR_ETC>>/template_files"/>
      <Property name="TEMP_PATH"          value="/tmp"/>
      <Property name="ARGUMENT_SEPARATOR" value=";"/>
      <Property name="VALUE_SEPARATOR"    value=":"/>
      <Property name="ARRAY_CHARACTER"    value="[]"/>
    </Properties>
  </Device>

</Devices>
```

Any new device must insert between <Device> tags. Inside we can find the next tags:

Bean: Contains a Class extending NetworkElement or NetworkElement itself. It will be used to recover information from DB. In our example Equipment class is used because some parameters like *logintacacs* are not present in the NetworkElement class. It will be fetched form DB calling the *findByPrimaryKey* method with the associated NetworkElement primary key as parameter.

manufacturer: Name of the manufacturer of the device. Will be used to validate if the device correspond to the desired equipment.

type: Type of device. Will be used to validate if the device correspond to the desired equipment.

model: Model of device. Will be used to validate if the device correspond to the desired equipment.

Params: Container of each param.

Param: Configuration values extracted from DB. Once the Bean class is instantiated (fetched from DB), the getter method found in "value" will be executed and associated to the "name" literal and recovered with the DeviceInformation object described below.

Properties: Container of each Property.

Property: Literal configuration values. The value set in "value" will be recovered associated to the "name" literal and recovered with the DeviceInformation object described below.

16.3 Recovering parameters

The parameters configured in the *deviceRegister.xml* are stored in an object called *DeviceInformation*. This object will be available as a parameter of the interface methods.

The methods that return information of the xml file are:

- *getName*: Returns the NetworkElement ID of the equipment.

  Syntax:

```
public String getName()
```

- *getBackupURL*: Returns the class *com.hp.spain.backup.BackupURL* from which it is possible to extract the entire connection URL or every of its parts alone. See the appendix for more info about this object.

  Syntax:

```
public BackupURL getBackupURL()
```

- *getMemoryTypes*: The result is a com.hp.spain.inventory.MemoryType array bean with all the memory types that are associated (in DB) with the model of the selected equipment.

  Syntax:

```
public MemoryType[] getMemoryTypes()
```

- *getAccessProperties*: Class that provided the values referenced by the deviceRegister.xml parameters (defined in a <Param> tag), but only if these parameters are one of the following: *login*, *password*, *logintacacs*, *passwordtacacs* or *enablePassword*.

  The returned class is *com.hp.spain.backup.AccessProperties* and it is just a bean with getter methods to access the parameters commented above. It can be consulted in the Appendix.

  Syntax:

```
public AccessProperties getAccessProperties()
```

- *getExtAccessProperties*: Class that provides the values in the deviceRegister.xml properties (defined in a <Property> tag) and the values referenced by the parameter (defined in a <Param> tag) that are different from the ones of the *AccessProperties* (*login*, *password*, *logintacacs*, *passwordtacacs* or *enablePassword*).

  Syntax:

```
public ExtAccessProperties getExtAccessProperties()
```

16.4 Coding a new backup driver

The next step is to create the java application that will be the core of the backup and deploy it in the WFM classpath. If you already have an implementation of a backup for the desired equipment just jump this step.

As an example from now to the end we are going to suppose that we are implementing a driver for a Riverstone equipment.

16.4.1 Implementing the interfaces

Only the implementation of two interfaces is needed in order to create the backup driver. These are *com.hp.spain.backup.BackupDriver* and *com.hp.spain.backup.BackupConnection.*

In our example two new classes are created to implement them:

- BackupDriver implementation

```
public class RiverStoneBackupDriver implements com.hp.spain.backup.BackupDriver {

}
```

- BackupConnection implementation

```
public class RiverstoneRSBackupConnection implements com.hp.spain.backup.BackupConnection,
RiverstoneRSConstants {

}
```

RiverstoneRSConstants is a support class with useful constant only valid for our example.

The tasks to perform in each interface are:

- BackupDriver
  - Registering the driver
  - Validating the incoming petition
  - Instantiating the BackupConnection
- BackupConnection
  - Implementing the connection to the equipment

16.4.2 Tasks in BackupDriver

16.4.2.1 Registering the driver

The driver needs to self register in the configuration management application in order to be functional. This is mandatory and it is done in a static block in the *BackupDriver* implementation.

```
static {
   BackupManager.registerBackupDriver(new RiverStoneBackupDriver());
}
```

16.4.2.2 Validating the incoming petition

The next task of the driver is to check the petition and decide if accept it or not. This can be done implementing the *acceptsURL()* method in the *BackupDriver*.

Syntax:

```
public boolean acceptsURL(BackupURL bURL);
```

As you can see it accepts a *BackupURL* object as parameter. This object will inform us about the target equipment. It can be consulted in later sections.

Example:

```
public boolean acceptsURL(BackupURL backupurl) {
   boolean flag = false;
   try {
```

```
      if ((backupurl.getManufacturer().toLowerCase().equals(MANUFACTURER.toLowerCase())) &&
            ((backupurl.getType().toLowerCase().equals(EQUIPMENT.toLowerCase())) ||
             (backupurl.getType().toLowerCase().equals(EDC.toLowerCase()))))
      {
         flag = true;
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
   return flag;
}
```

16.4.2.3 Instantiating the BackupConnection

The *BackupDriver* class is instantiated in its static block but not the *BackupConnection*. This class is instantiated by the *BackupDriver* in the method *getConnection()*.

Syntax:

```
public BackupConnection getConnection(DeviceInformation deviceInformation)
throws BackupException;
```

The unique parameter is the class *com.hp.spain.backup.DeviceInformation* and it provides the parameters that the device will need for the connection as seen before.

Example:

```
public BackupConnection getConnection(DeviceInformation deviceinformation)
throws BackupException {
   RiverstoneRSBackupConnection riverstonersbackupconnection =
         new RiverstoneRSBackupConnection(
               deviceinformation, equipmentDriverClassName, protocol, port);
   riverstonersbackupconnection.open();
   return riverstonersbackupconnection;
}
```

The variables *equipmentDriverClassName*, *protocol* and *port* have been populated in the constructor.

16.4.2.4 Finishing implementing BackupDriver

A *getManufacturer()* method still remains to implement. It must return the name of the manufacturer.

Example:

```
public String getManufacturer() {
   return "Riverstone";
}
```

16.4.3 Tasks in BackupConnection

16.4.3.1 Implementing the connection to the equipment

e. Open a connection

For opening a connection the *open()* method must be implemented:

Syntax:

```
public void open() throws BackupException
```

Example:

```
private boolean open = true; // The connection with a RiverStone is always open
public void open() throws BackupException {
  this.open = true;
}
```

f. Close a connection

For closing a connection the *close()* method must be implemented:

Syntax:

```
public void close()
```

Example:

```
public void close() {
  this.open = false;
}
```

g. Check if the connection is open

For this the isOpen connection must be implemented:

Syntax:

```
public boolean isOpen()
```

Example:

```
public boolean isOpen() {
   return open;
}
```

h. Perform the backup

There are two methods that must be implemented to perform the backup:

Syntax:

```
public EquipmentConfiguration performBackup(MemoryType memType)
throws BackupException
```

and

```
public EquipmentConfiguration performBackup(MemoryType memType, String protocol)
throws BackupException
```

Returns:

Both of them return an instance of *com.hp.spain.inventory.EquipmentConfigurationExt* object. No matter if that the interface asks for a *EquipmentConfiguration* object, a *EquipmentConfigurationExt* must be return or the BACKUP WILL NOT BE SAVED. The implementer must use the next constructor:

```
public EquipmentConfigurationExt(
   String NetworkElementID,
   java.util.Date Timestamp,
   String Version,
   String ConfigurationId,
   java.util.Date LastAccessTime,
   String RetrievalName,
   String MemoryType,
   String CreatedBy,
   String ModifiedBy,
   byte[] Data)
```

In this method the backup driver must retrieve the configuration from the equipment and create the object with this information. The difference between the two methods is that the first is for the default protocol and the in the other the user can specify the protocol.

Example:

```
public EquipmentConfiguration performBackup(MemoryType memType)
throws BackupException {
   return performBackup(memType, DEFAULT PROTOCOL);
}
```

```
public EquipmentConfiguration performBackup(MemoryType memType, String sBackupProtocol)
throws BackupException {
   EquipmentConfiguration ec = new EquipmentConfiguration();
   if(SCP_PROTOCOL.equals(sBackupProtocol)){
      ec = performSCPBackup(memType);
   } else if(TFTP_PROTOCOL.equals(sBackupProtocol)){
      ec = performTFTPBackup(memType);
   }
   return ec;
}

private EquipmentConfiguration performSCPBackup(MemoryType memType) throws BackupException {
   String configuration = null;
   EquipmentConfiguration ec = null;
   FileInputStream fileInputStream = null;
   String separador = "@@";
   ExtAccessProperties eap = deviceInfo.getExtAccessProperties();
   String sHostname = deviceInfo.getBackupURL().getHost();
   String sLogin = deviceInfo.getAccessProperties().getLogin();
   String sPassword = deviceInfo.getAccessProperties().getPassword();
   // Instantiating an SCP client.
   scpClient = new AdvancedSCPClient(sHostname, sLogin, sPassword);
   try {
      scpClient.connect();
      File backupFile = null;
      String fileName= memType.getFilename();
      backupFile = this.scpClient.getFile(fileName);
      configuration = this.readFileInputStream(backupFile);
      String equipmentName = deviceInfo.getExtAccessProperties().getProperty("ROUTER_NAME");
      Date timestamp = new java.util.Date();
      String version = "1.0";
      String configurationId = deviceInfo.getBackupURL().getIdentifier();
      Date lastAccessTime = timestamp;
      String retrievalName =
         deviceInfo.getExtAccessProperties().getProperty("RETRIEVAL_METHOD");
      String memoryType = memType.getMemoryname();
      String createdBy = sLogin;
      String modifiedBy = "";
      // The object MUST be EquipmentConfigurationExt
      ec = new EquipmentConfigurationExt(
            deviceInfo.getName(),
            timestamp,
            version,
            configurationId,
            lastAccessTime,
            retrievalName,
            memoryType,
            createdBy,
            modifiedBy,
            configuration.getBytes());
      String sBaseDir = deviceInfo.getExtAccessProperties().getProperty(BASE_DIR);
      String sDirPath = "";
      if ((equipmentName != null) && (sBaseDir != null)) {
         sDirPath = sBaseDir + equipmentName;
         File fFoo = new File(sDirPath);
         if (!fFoo.exists())
            fFoo.mkdirs();
      }
      // Creating file name
      Calendar cToday = Calendar.getInstance();
      String sAbsolutePath = equipmentName + separador;
      sAbsolutePath += retrievalName + separador;
      sAbsolutePath += createdBy + separador;
      sAbsolutePath += memoryType + "_";
      sAbsolutePath += cToday.get(Calendar.DAY_OF_MONTH) + "-";
      sAbsolutePath += (cToday.get(Calendar.MONTH)+1) + "-";
      sAbsolutePath += cToday.get(Calendar.YEAR) + " ";
      sAbsolutePath += System.currentTimeMillis();
      File fFile = new File(sDirPath + "/" + sAbsolutePath);
      backupFile.renameTo(fFile);
   } catch(AdvancedSCPException scpe) {
      String msg = "Error getting file '" + "" + "' from scp server. " + scpe.getMessage();
      scpe.printStackTrace();
```

```
      throw new BackupException(msg);
   }
   return ec;
}
```

i. Upload a backup to the equipment

There two methods that must be implemented to upload a backup:

Syntax:

```
public void sendConfiguration(
      EquipmentConfiguration sConfiguration,
      MemoryType memType)
throws BackupException
```

and

```
public void sendConfiguration(
      EquipmentConfiguration sConfiguration,
      MemoryType memType,
      String protocol)
throws BackupException
```

These methods have to upload the configuration file to the equipment. No matter that in the interface the parameters are *EquipmentConfiguration*, a *com.hp.spain.inventory.EquipmentConfigurationExt* object are passed, so the implementer can cast to this object and call *getData()* to access to the configuration data.

Example:

```
public void sendConfiguration(
      EquipmentConfiguration sConfiguration,
      MemoryType memType) throws BackupException {
   sendConfiguration(sConfiguration, memType, DEFAULT PROTOCOL);
}

public void sendConfiguration(
      EquipmentConfiguration sConfiguration,
      MemoryType memType,
      String sBackupProtocol)
throws BackupException {
   if(SCP_PROTOCOL.equals(sBackupProtocol)){
      sendSCPConfiguration(sConfiguration, memType);
   } else if(TFTP PROTOCOL.equals(sBackupProtocol)) {
      sendTFTPConfiguration(sConfiguration, memType);
   }
}

private void sendSCPConfiguration(
      EquipmentConfiguration sConfiguration,
      MemoryType memType)
throws BackupException {
   FileOutputStream fos=null;
   File fileToSend = null;
   try {
      // Remember that the object is an EquipmentConfigurationExt!!!!
      byte[] totalbytes=((EquipmentConfigurationExt)sConfiguration).getData();
      String filename = null;
      Calendar cal=Calendar.getInstance();
      String curDate = String.valueOf(cal.get(Calendar.YEAR)) +
                       String.valueOf(cal.get(Calendar.MONTH)) +
                       String.valueOf(cal.get(Calendar.DATE));
      filename="hpsa_"+memType.getFilename()+".tmp";
      fileToSend = new File(BACKUP_DIRECTORY + filename);
      fos = new FileOutputStream(fileToSend);
      fos.write(totalbytes);
      fos.close();
      fos = null;
      String filepathdestino = EQUIPMENT_CONFIGURATION_ROOT_PATH + filename;
      ExtAccessProperties eap = deviceInfo.getExtAccessProperties();
      String sHostname = deviceInfo.getBackupURL().getHost();
      String sLogin = deviceInfo.getAccessProperties().getLogin();
```

```
      String sPassword = deviceInfo.getAccessProperties().getPassword();
      try{
         this.scpClient = new AdvancedSCPClient(sHostName, sLogin, sPassword);
         this.scpClient.connect();
         this.scpClient.putFile(fileToSend,sFilePathDestino);
      } catch(AdvancedSCPException e) {
         String msg = "Error putting file on scp server '" + "" + "'. " + e.getMessage();
         e.printStackTrace();
         throw new BackupException(msg);
      }
   } catch(IOException e) {
      String msg = "Error putting file on scp server '" + ""+ "'. " + e.getMessage();
      e.printStackTrace();
      throw new BackupException(msg);
   } finally {
      fileToSend.delete();
   }
}
```

j. Finishing implementing BackupConnection

Helper methods that must be implemented are:

- *getDeviceInformation*: The DeviceInformation object that the driver must have retained in the creation time.

  Syntax:

```
public DeviceInformation getDeviceInformation(MemoryType memType)
```

  Example:

```
public DeviceInformation getDeviceInformation() {
   return this.deviceInfo;
}
```

- *toString*: String with the connection session information for logging purposes.

  Syntax:

```
public String toString()
```

  Example:

```
public String toString() {
   AccessProperties aProperties = deviceInfo.getAccessProperties();
   BackupURL bURL = deviceInfo.getBackupURL();
   ExtAccessProperties eap = deviceInfo.getExtAccessProperties();
   StringBuffer sb = new StringBuffer("");
   sb.append("- RiverStone Backup Connection -");
   sb.append("\n login: " + aProperties.getLogin());
   sb.append("\n password: " + aProperties.getPassword());
   sb.append("\n enablePassword: " + aProperties.getEnablePassword());
   sb.append("\n Type: " + bURL.getType());
   sb.append("\n Model: " + bURL.getModel());
   sb.append("\n identifier: " + bURL.getIdentifier());
   sb.append("\n host: " + bURL.getHost());
   sb.append("\n port: " + bURL.getPort());
   Enumeration e = eap.getPropertyNames();
   while (e.hasMoreElements()) {
      String sKey = (String) e.nextElement();
      sb.append("\n " + sKey + ": " + eap.getProperty(sKey));
   }
   sb.append("\n open: " + open);
   return sb.toString();
}
```

Some methods are still not implement but must be implemented as are defined in the interface implement them with no code). There are:

- *getConfiguration*
- *isAudit*

- *sendFile*

Example:

```
public EquipmentConfiguration getConfiguration(MemoryType memType) {
   return null;
}

public boolean isAudit() {
   return true;
}

public void sendFile(String Filepath) throws BackupException{}

public void sendFile(byte[] Config) throws BackupException{}
```

16.5 Using an existent backup driver

Using an existent backup driver is analog to create a new one, the only difference is that the code step is not needed (a third party provides the jar and must be deployed in the MWFM classpath). The user must ask to the third party in order to fill the correct parameters and properties.

16.6 BackupURL getter methods

BackupURL is an object used to map a equipment, the getter methods informs about the type of equipment and are:

- public String getManufacturer() {…} – Returns the equipment manufacturer.
- public String getType() {…} – Returns the equipment type.
- public String getModel(){…} – Returns the equipment model.

Other methods are provided to recover extra information:

- public String getIdentifier(){…} – Returns the configuration identifier associated to this backup petition.
- public String getHost(){…} – Returns the equipment host to connect to.
- public int getPort(){…} – Returns the equipment port to connect to.

16.7 AccessProperties getter methods

- public String getLogin() {…} – Returns de login parameter
- public String getLoginTacacs(){…} – Returns the logintacacs parameter
- public String getPassword() {…} – Returns the password parameter
- public String getPasswordTacacs() {…} – Returns paswordtacacs parameter
- public String getEnablePassword() {…} – Returns the enablepasword parameter

17 Xmaps

XMaps is a java API (Application Programming Interface) designed to provide an easy way of working with diagrams. A diagram can represent a network, a workflow or any process involving several steps or layers. Using XMaps these diagrams can be created, modified and stored and can be shown graphically on screen, allowing users to interact with its components, move them to another position, change its attributes or invoke the operations assigned to them.

XMaps generates the JavaScript code representing the specified diagram and offers methods to sort the diagram's components using a specific algorithm or one designed by the users themselves. Then, this code can be shown in a JSP provided with the API.

17.1 API structure

The main component of the XMaps API is the diagram. A diagram is a set of different components which are somehow related. Therefore, using diagrams we can represent a network, a workflow or any process involving several steps or layers.

There are several kinds of components:

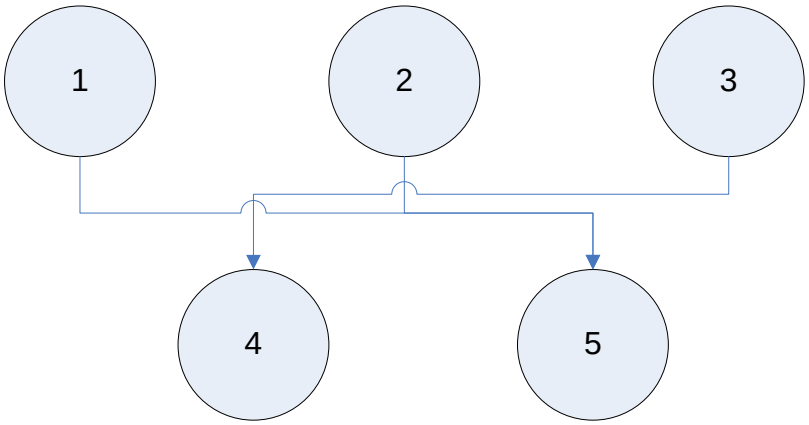
- Node: The main elements in a Diagram are the nodes. The nodes can represent a computer or a router in a network, or some action inside a process. Each node can be connected to other nodes using Ports. A node can have any number of ports.
- Port: The Ports are the components used to store the Connections between nodes. A port always belongs to a node. Each port can contain a connection to another port (which belongs to another node). A node can contain as many ports as needed.
- Connection: The connections are components used to link a pair of ports. Each connection is associated to two ports belonging to different nodes.
- Text: The user can attach text to a specific position to clarify some aspects of the diagram. These texts are not interactive. The user cannot select them nor move them.
- Image: Images can be shown, in a similar way as the texts, to add non interactive information to a diagram.

The Diagram, Nodes, Ports and Connection can have some specific graphical attributes (a node can have an image, the connections can have different colors and the user can locate the ports in a specific section of a node) and operations (the actions which will be invoked by the user using this component).

17.2 Application development

In order to describe how to develop an application using XMaps we are going to define a simple diagram step by step.

Our diagram will have the next five nodes, each one of them representing a computer in a WAN. Some of these computers are connected to each other.



After that, to make the diagram clearer, we will add a text in the upper left corner describing what this diagram is representing. The computer number 3 is connected to the WAN from the outside using the Internet, so we can add the picture of a cloud surrounding it to represent that this node is not physically connected to this WAN.

17.2.1 Nodes definition

The first step in the design of our diagram is the creation of the nodes involved in it. Each component has an id, that should be unique, and several attributes depending of its kind. The node's graphical attributes are:

- Name.
- Background Color
- Width
- Height
- Image Path
- Border color
- Text Position
- Type
- Ports Location
- Text Background color.

These parameters will be discussed in detail later in this document. At this point we are going to create a simple node with a name and an image.

```
NodeView c1 = new NodeView("c1");
NodeView c2 = new NodeView("c2");
NodeView c3 = new NodeView("c3");
NodeView c4 = new NodeView("c4");
NodeView c5 = new NodeView("c5");
c1.setName("Computer 1");
c2.setName("Computer 2");
c3.setName("Computer 3");
c4.setName("Computer 4");
c5.setName("Computer 5");
c1.setImagePath("./computer.jpg");
c2.setImagePath("./computer.jpg");
c3.setImagePath("./computer.jpg");
c4.setImagePath("./computer.jpg");
c5.setImagePath("./computer.jpg");
```

17.2.2 Ports Definition

As we said before, a port is a component contained by a node. The ports are used to link nodes using connections. Each port belongs to a single node and is connected to a single port using only one connection. There is no limit to the number of ports contained by a node.

The ports have their own attributes:

- Color.
- Type.

The type represents the shape of the node. The ports can be square-shaped, round-shaped or hidden (they will not be shown in the screen). More detail will be added later.

For our example we need to create a port for nodes from 1 to 4 and two ports for the node number 5 (it has two incoming connections). The next code will define our ports:

```
PortView p1 = new PortView(c1, "p1");
PortView p2 = new PortView(c2, "p2");
PortView p3 = new PortView(c3, "p3");
PortView p4 = new PortView(c4, "p4");
PortView p5 = new PortView(c5, "p5");
PortView p6 = new PortView(c5, "p6");
n1.addPort(p1);
n2.addPort(p2);
n3.addPort(p3);
n4.addPort(p4);
n5.addPort(p5);
n5.addPort(p6);
```

At this point we don't care about its shape or color. By default the color will be yellow and the shape round.

17.2.3 Connections definition

Now we need to link the nodes which have a connection between them. As we explained before, each connection will tie two ports, each one of them belonging to only one node. The connections have some attributes too:

- Color
- Weight
- Image Path
- Origin Node Id
- Destination Node Id
- Corner side
- Arrow on origin (show an arrow in the origin side of the connection)
- Arrow on destination (show an arrow in the destination side of the connection)

In our example we will use the default values for all the parameters.

We need to create three connections, one linking the node 3 with the node 4 and two of them connecting the nodes 1 and 2 with the node 5. We are going to create standard connections now and we will associate them to their ports later.

```
ConnectionView con1_5 = new ConnectionView("con1_5");
ConnectionView con2_5 = new ConnectionView("con2_5");
ConnectionView con3_4 = new ConnectionView("con3_4");
```

17.2.4 Creating a Diagram

As exposed before, the Diagram is the main component of Xmaps. It will contain all the components which define our WAN or our process, including images and texts. The Diagram, as the previous components, has some attributes which can be modified by the user:

- Name
- Border Type
- Lettering Color
- Lettering Text
- Background Color
- Origin X
- Origin Y
- Clip X
- Clip Y
- Width
- Height
- Drag Enabled
- Scale Map Active
- Zoom Active
- Right button enabled
- Back Url (The Url which will be invoked when the back button is pressed)

In our example we will use the default values for all of them.

The diagrams can be divided in layers but this is not mandatory. It will affect the way the nodes are going to be shown. If the diagram is organized in layers, all the nodes will be situated forming rows, one for each layer. The first row will be the layer number 0. Only the nodes can be associated to layers.

The API offers an algorithm to sort the diagram's nodes depending of their layer. If the diagram is not organized in layers the algorithm will try to assign one to each node. As a result of this process, the number of crossings between connections will be reduced and the representation of the diagram will be clearer. The use of the sorting algorithm will be detailed later.

In order to continue with our example, we need to add the components defined before to a new diagram. We need to add the nodes specifying its layer and associate the connections with their corresponding ports inside the diagram.

```
Diagram diagram = new Diagram();
diagram.addNodeView(c1, 0);
diagram.addNodeView(c2, 0);
diagram.addNodeView(c3, 0);
diagram.addNodeView(c4, 1);
diagram.addNodeView(c5, 1);
diagram.addConnectionByPort(con1_5, p1, p5);
diagram.addConnectionByPort(con2_5, p2, p6);
diagram.addConnectionByPort(con3_4, p3, p4);
```

17.2.5 Adding a Text

The diagram can contain as many texts as needed. The texts can't be associated to layers and it's mandatory to associate them to a specific location in the screen. When adding a text to a diagram we have to specify a pair of coordinates x and y.

The texts only have an attribute which contains the string which will be shown in the screen.

In our example, we are going to add a text explaining the purpose of the diagram. This text will be situated in the upper left side of the screen.

```
Text t = new Text("t", "This is example shows how XMaps works");
diagram.addText(t, 5, 5);
```

17.2.6 Adding a Image

In the same way a diagram can contain texts, it can also contain images. The images can't be associated to layers.

This component only has an attribute, the path to the image which will be shown on screen. Also, when we add the image to the diagram, a position (a pair of coordinates x and y) must be specified.

As we said before, we are going to add an image to our example diagram to represent that one of our computers is connected to our WAN through the Internet. To represent that, we are going to use the picture of a cloud.

```
Image i = new Image("/cloud.jpg", "i");
diagram.addImage(i, 10, 10);
```

17.2.7 Sorting the diagram

At this point all our components are included in the diagram, but the API offers some methods to clarify its structure. By examining the diagram that shows the location of the nodes, and which was shown at the beginning of this section, we can see that all connections between the nodes cross each other. We can use the sorting algorithm (Sugiyama algorithm) provided by the API to sort the nodes and eliminate the crossings.

```
Sugiyama sortS = new Sugiyama();
diagram.sortGraph(sortS, true);
```

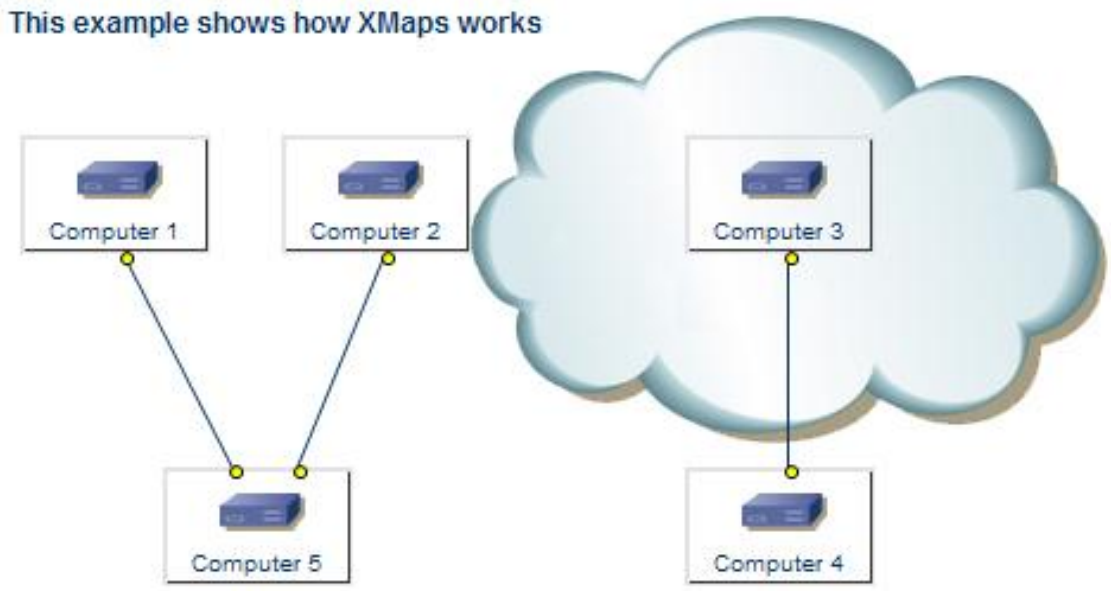
17.2.8 The resulting diagram

This is the complete code to generate our diagram:

```
NodeView c1 = new NodeView("c1");
NodeView c2 = new NodeView("c2");
NodeView c3 = new NodeView("c3");
NodeView c4 = new NodeView("c4");
NodeView c5 = new NodeView("c5");
c1.setName("Computer 1");
c2.setName("Computer 2");
c3.setName("Computer 3");
c4.setName("Computer 4");
c5.setName("Computer 5");
c1.setImagePath("./computer.jpg");
c2.setImagePath("./computer.jpg");
c3.setImagePath("./computer.jpg");
c4.setImagePath("./computer.jpg");
c5.setImagePath("./computer.jpg");

PortView p1 = new PortView(c1, "p1");
PortView p2 = new PortView(c2, "p2");
PortView p3 = new PortView(c3, "p3");
```

```
PortView p4 = new PortView(c4, "p4");
PortView p5 = new PortView(c5, "p5");
PortView p6 = new PortView(c5, "p6");
c1.addPort(p1);
c2.addPort(p2);
c3.addPort(p3);
c4.addPort(p4);
c5.addPort(p5);
c5.addPort(p6);

Diagram diagram = new Diagram();
diagram.addNodeView(c1, 0);
diagram.addNodeView(c2, 0);
diagram.addNodeView(c3, 0);
diagram.addNodeView(c4, 1);
diagram.addNodeView(c5, 1);

ConnectionView con1_5 = new ConnectionView("con1_5");
ConnectionView con2_5 = new ConnectionView("con2_5");
ConnectionView con3_4 = new ConnectionView("con3_4");

diagram.addConnectionByPort(con1_5, p1, p5, true);
diagram.addConnectionByPort(con2_5, p2, p6, true);
diagram.addConnectionByPort(con3_4, p3, p4, true);

Text t = new Text("t", "This example shows how XMaps works");
diagram.addText(t, 5, 5);

Image i = new Image("/cloud.jpg", "i");
diagram.addImage(i, 10, 10);

Sugiyama sortS = new Sugiyama();
diagram.sortGraph(sortS, true);
```

This is the diagram represented using the XMaps API:



17.2.9 Operations

Some diagram's components and the diagram itself can contain operations. The operations are actions that can be executed by the user. These operations are calls to URLs or Struts actions and can be used to fetch some information about the component from the database, to open another JSP containing another diagram or to execute any useful action designed by the developers of the application using XMaps.

When the user right clicks on a component a window will appear showing all the operations defined for that component.

The components which can contain operations are the Nodes, the Connections, the Ports and the Diagrams.

There are several possible targets for an operation:

- OPERATION_TARGET_BELOW
- OPERATION_TARGET_HIDDEN: The operation will not be shown
- OPERATION_TARGET_POPUP: The operation will be shown in a pop up window
- OPERATION_TARGET_INNERPOPUP: The operation will be shown in an iframe
- OPERATION_TARGET_SELF
- OPERATION_TARGET_BALLON: The operation will be shown in a "balloon" close to the element

Ex.

```
c5.addOperation("testOperation", "/images/operation.gif", "/Action.do",
DiagramConstants.OPERATION_TARGET_HIDDEN, true, false, null, null);
```

The fields of this method are:

- Text: The text to show associated to the operation
- Image Path: The path to the image associated to the operation
- Action: The action to execute in this operation.
- Target: The target of the operation: DiagramConstants.OPERATION_TARGET_BELOW, ..._HIDDEN, ..._POPUP, ..._INNERPOPUP, ..._SELF or ..._BALLOON
- Default operation: True if the operation should be executed by double clicking in the component.
- Activate on startup: True if the operation should be executed on startup
- Width: The horizontal coordinate associated to the balloon associated to the operation (only for DiagramConstants.OPERATION_TARGET_BALLOON)
- Height: The vertical coordinate associated to the balloon associated to the operation (only for DiagramConstants.OPERATION_TARGET_BALLOON)

There are several default operations in some of the previous components. The components Diagram, Node, Connection and Port have the next default operation:

- See attributes: all the components which can contain operations and have any attribute filled in will have this operation. The values of all the attributes entered will be shown when the user executes this action.

The component Connection has the next default operations:

- Set Corner: If the diagram's attribute Drag Enabled is set to true, this action will be available for all the connections. It allows the user to change the connection's shape, creating a corner to the right, to the left or making the connection to go straight.

The Diagrams have the next default operations:

- Lettering: If the diagram has its attribute Lettering filled this operation will be enabled, allowing the user to activate or deactivate the diagram's lettering.
- Scale Map: If the scale map is active this operation will be enabled, allowing the user to make the map visible or invisible.

- Undo: If drag enable is set to true this operation will be active, allowing the user to undo all the changes realized over the map.
- Save: If drag enable is set to true this operation will be active, allowing the user to save all the changes realized over the map.

17.2.10 On Select Operations

There is also a specific kind of operations that can be triggered by the action of selecting the associated element. These operations will work in the same way as any other operation:

```
C5.addOperationOnSelect(
    "showData()",
    DiagramConstants.OPERATION_TARGET_BALLOON,
    new Integer(10),
    new Integer(10));
```

The parameters associated to this method are:

- Action: The action to execute in this operation.
- Target: The target of the operation: DiagramConstants.OPERATION_TARGET_BELOW, ..._HIDDEN, ..._POPUP, ..._INNERPOPUP, ..._SELF or ..._BALLOON
- Width: The horizontal coordinate associated to the balloon associated to the operation (only for DiagramConstants.OPERATION_TARGET_BALLOON)
- Height: The vertical coordinate associated to the balloon associated to the operation (only for DiagramConstants.OPERATION_TARGET_BALLOON)

17.3 Sorting Algorithms

When a new diagram is created, all its nodes will be located in the position (0,0) unless another location were specified while adding the nodes to the diagram. If the user doesn't want to assign coordinates to all the nodes in the diagram, he can use the sorting algorithm to assign the node's coordinates automatically. This algorithm will try to reduce the crossings between the connections as much as possible in order to obtain a clear diagram.

The algorithm works on layered diagrams (diagrams whose nodes are organized in layers). If the diagram which the user wants to sort is not layered he can try to organize it in layers automatically:

```
  s = new Sugiyama();
  diagram.generateLayers();
  diagram.sortGraph(s, true);
```

The sorting algorithm used in this API was originally developed by Kozo Sugiyama, and therefore his method is called Sugiyama Algorithm. In order to use it is necessary to instantiate the class and use it as a parameter of the function sortDiagram(). More details about it will be provided later.

The developers of applications using the XMaps API can develop their own sorting algorithms to suit their needs by extending the interface GraphDrawer.

17.4 Solution Container Integration

Along with the API, Xmaps provides a struts action through which the developer can display the generated diagram. This action is "/DrawDiagramAction.do" and it is integrated in the solution container during the installation process.

This action invokes an internal JSP, responsible for generating the javascript code needed to represent the diagram.

The action receives a single parameter in the request:

Parameter	Mandatory	Description
DIAGRAM	Yes	Object of class com.hp.spain.xmaps.Diagram

The following example shows how to invoke it:

First, we write an action in charge of generating the diagram and invoke the generic Xmaps action:

```
import com.hp.spain.xmaps.diagram.Diagram;
import com.hp.spain.xmaps.diagram.DiagramConstants;

public class DrawExampleAction extends Action {

  public ActionForward execute(
      ActionMapping mapping,
      ActionForm form,
      HttpServletRequest request,
      HttpServletResponse response)
  throws IOException, ServletException {
    String target = FAILURE;
    Diagram myDiagram;
    DiagramManager dm;
    try {
      myDiagram = new Diagram();
      ... // code to define the diagram
      dm = new DiagramManager();
      dm.addDiagram(myDiagram, false);
      request.getSession().setAttribute(DiagramConstants.DIAGRAM MANAGER, dm);
      target = SUCCESS;
    }
    catch(Exception e) {
      e.printStackTrace();
      target = FAILURE;
    }
    return (mapping.findForward(target));
  }

}
```

Second, we add the new entry in the struts-config file:

```
<action path="/DrawExampleAction"
        type="com.hp.example.struts.action.DrawExampleAction"
        name="SimpleForm">
  <forward name="success" path="/DrawDiagramAction.do"/>
  <forward name="failure" path="/jsp/example/error.jsp"/>
</action>
```

The new diagram will be displayed every time the action is invoked.

18 ECP Console

The ECP Console is a web application that allows establishing connections with remote equipments through the ECP, using several available protocols such as telnet or SSH, for example.

The ECP Console application is prepared to analyze execution permissions on any typed command for every user and thus, the operations that can be performed on remote equipments are strictly controlled.

The ECP Console application allows the use of command scripts for each user and remote equipment.

There is also a administration GUI which allows the users to define hosts, command filters and scripts and store them in the database. The access to this GUI is associated with the roles "futuregui" and "ECP", any user accessing the GUI must have both roles.

18.1 Functionality

18.1.1 Command scripts

Command scripts are predefined command sequences which are executed using a different ECP template than typed commands do. They permit a high degree of complexity, allowing the users to define loops and create variables. Thanks to them it's possible to save complex structures and execute them with a single click.

The command scripts are stored in the database and accessible only for the desired users.

Check the document "*SPI for Service Providers – ECP Console - User reference.pdf*", section 3.2 for further information about command scripts.

18.1.2 Opening an ECP Console

In order to open a console the ECP configuration must be properly set at the *ecp.properties* file. There are three properties that must be set:

- ecpmanager.service.host: the ECP name or IP.
- ecpmanager.service.port: the ECP port number.
- ecpmanager.service.name: the ECP RMI service name.

The ECP Console must be opened through the provided *OpenConsoleECP* struts action. In order to obtain the available scripts for a given user from database and the remote host to which the commands will be send it is necessary to provide the action with the next parameters:

- *hostManufacturer*: the remote host manufacturer.
- *hostModel*: the remote host model.
- *hostVersion*: the remote host version.

This parameters constitute the necessary data to define a host (Check the document "*SPI for Service Providers – ECP Console - User reference.pdf*", section 3.1 for further information about hosts).

Depending on the connection class that will be established later in the remote host there are different parameters that must be specified. See the section below for further information.

18.1.3 Connecting to the remote equipment

The ECP provides two ways to obtain connections to a remote host: through a static pool or through a dynamic one. In this way, several parameters must be specified at the ECP Console opening action for the successful connection setup.

Needed parameters for a static pool connection:

- *poolName*: the ECP static pool name used to get an available connection.
- *queueId*: the ECP queue to be used. This parameter is optional and its default value if not specified is 1.

For example:

```
http://localhost:8089/activator/OpenConsoleECP.do?hostmanufacturer=alcatel&h
ostModel=riverstone&hostVersion=1&hostname=16.38.0.136&poolName=testingpool
```

Needed parameters for a dynamic pool connection:

- *hostname*: the remote host name or IP.
- *port*: the remote host port.
- *login*: the remote host login username.
- *password*: the remote host password.
- *passwordEnable*: the remote host password used to change the session mode once the remote host has been connected.
- *protocol*: the remote host connection protocol, typically telnet or SSH.
- *connectionResourceClassName*: the java class which implements the driver used for the remote host connection.
- *poolName*: the dynamic pool name. This is an optional parameter because dynamic pool names can be automatically generated by the ECP.
- *maxCon*: the maximum number of connections to be contained by the dynamic pool. It is an optional parameter.
- *minCon*: the minimum number of connections to be contained by the dynamic pool. It is an optional parameter.
- *initOnCreate*: boolean value which indicates if the connection must be initialized on instantiation instead of on the first time it is used. It is an optional parameter.
- *overMinimumConnTimeout*: the timout (in milliseconds) for the not used temporary connections over the minimum before they are closed. It is an optional parameter.
- *reservedConnTimeout*: the time (in milliseconds) that a connection may be in use by a single operation. It is an optional parameter.
- *poolTimeout*: the timeout (in milliseconds) for a not used dynamic pool before it is closed. It is an optional parameter.
- *additionalData*: some additional data, if needed. It is an optional parameter.
- *queueId*: the ECP queue to be used. This parameter is optional and its default value if not specified is 1.

For example:

```
http://localhost:8089/activator/OpenConsoleECP.do?hostmanufacturer=alcatel&h
ostModel=riverstone&hostVersion=1&hostname=16.38.0.136&port=23&login=guest&p
assword=gpwd&passwordEnable=egpwd&protocol=telnet&connectionResourceClassNam
e=com.hp.spain.connection.RiverstoneRSConnectionResource
```

19 Configuration

Alter installing the SC it is compulsory to check that the configuration is correct in order for all the projects that make up the SC work properly.

There are three configuration files that we must draw special attention: *web.xml*, where the servlets, taglibs and ejbs are defined among other things; *mwfm.xml*, where the MWFM is configured with all its modules, among them the user authentication module (see this module documentation for more information); and the datasources, which are configured in the *standalone.xml* file, a JBoss configuration file.

19.1 *DB* module

In the *mwfm.xml* we can define as many access modules to different databases as are needed, but one of them must be necessarily called just db, which is the one the MWFM will use.

Through this module we configure the connection pool parameters to the database the MWFM and the rest of defined modules are going to interact with.

A DB module configuration must point to one of the different datasources configured in JBoss:

```
<Module>
   <Name>
     db
   </Name>
   <Class-Name>
     com.hp.ov.activator.mwfm.engine.module.OracleDatabaseModule
   </Class-Name>
   <Param
     name="datasource_name"
     value="mwfm-default-ds.xml"/>
</Module>
```

19.2 Authentication module

In the Authentication module we indicate the login system for the users, which can depend on the operating system or any other factor.

HPSA provides three different authentication systems validating the user against a specific operating system:

- *HPUXAdvancedAuthModule*
- *SolarisAdvancedAuthModule*
- *WindowsAdvancedAuthModule*

These authentication systems validate the user against the corresponding operating system and guarantee that the user exists and belongs to a role with access permissions to HP Service Activator, but none of them consults the permissions in the User Administration Module. To do this there is another authentication system:

- *DatabaseAdvancedAuthModule*

This validates the user against a database and guarantees that its username and password are valid.

In principle we can use the one we feel is more convenient, but only the last one applies the authentication against the User Administration Module.

For more information about the Authentication module please see the HPSA documentation.

19.3 MWFM Multiple

The SC can manage different MWFM, each of them residing in a machine with an IP and with its corresponding configuration file *auth.properties*.

In order to configure the available MWFMs in the system, you have to edit the *auth.properties* file. There, the next parameters should be filled:

- *mwfm_rmi_authX*: [X takes consecutive values starting from 0 onwards]: these parameters, numbered consecutively, indicate the RMI services of the different Master MWFM. (Ej. //localhost:2000/auth). This parameter has the same meaning here as the *mwfm_rmi_auth* parameter has in the Authentication Module configuration (see the section above).
- *mwfm_nameX* [X takes consecutive values starting from 0 onwards]: these parameters, numbered consecutively, indicate the names of the different MWFM which can be accessed from the HP Service Activator. Each parameter specified here must have its corresponding URL, which is indicated in the parameter *mwfm_urlX* similarly numbered. It is necessary that at least the first MWFM is defined, so the parameter mwfm_name0 must always be indicated.
- *mwfm_urlX* [X takes consecutive values starting from 0 onwards]: these parameters, numbered consecutively, indicate the URL in which each MWFM Publisher its methods using RMI. As in previous parameter, we need at least one *mwfm_nameX* parameter with the same numbering and so we necessarily have to define a *mwfm_url0*.
- *default_mwfm*: indicates which of the MWFM specified we will use as default. This parameter is not compulsory and if is not specified the value taken as default is the one that is defined by *mwfm_name0* and *mwfm_url0*.

19.4 Session management

SC provides a new feature that allows to manage the sessions of the logged on users. This is done declaring inside the *<web-app>* tags of the *web.xml* file, just before the servlet's declaration, the next Listener:

```
<listener>
   <listener-class>
      com.hp.spain.futuregui.session.SessionManagerImpl
   </listener-class>
</listener>
```

This feature is optional and will only be performed if this listener is defined. The class which implements the listener features belongs to the SC.

It is also possible to determine the number of users with the same username who can be logged on the SC at the same time. This value is established through an attribute of the HPSA_TEAM table called *userspersession*, so it will affect to every users of the team.

A value of 1 indicates that only one user can be logged on the SC with the same username, so the last logged on user will cause the log off of the first one. A value greater than 1 will have a similar effect, but will allow the specified number of users to be logged on. A value of 0 will set no limitation on the number of users with the same username logged on the SC.

Use the User Administration GUI to set the value of this attribute.

19.5 Struts

Nowadays there exist loads of Technologies which implement the Model-View-Controller paradigm and Struts is one of them. The SC bases its functionality on Struts, so its configuration is important.

Struts provides a separation between the presentation and business layers, as is specified in the MVC model, so that the JSP must take care of the first, and the actions and forms (extensible Java classes which Struts uses) take care of the second. Struts allows, among other things, to establish validation systems and access to automatic actions, and is constituted by a servlet called *ActionServlet* which listens to all requests directing the flow of execution towards the corresponding action. All actions are mapped in configuration files whose usual name is struts-*config.xml*.

The SC provides an extension to Strut's basic servlet that searches automatically for all the configuration files where the different applications specify the actions and forms which constitute them, in such a way that all configuration files that are stored in *WEB-INF/struts-config* are mapped without any need to specify each of them in the web.xml, as happens with Strut's *ActionServlet*. In order to make this extension of the Action Servlet available it's necessary to indicate it between the *<servlet-class>…<servlet-class>* tags of the servlet definition.

The name of the Struts servlet is, by common use, action.

Also, it's necessary to indicate a series of initial parameters which determine the servlet configuration. Currently the following are necessary, although there are other which can be specified (see the Struts documentation for more information):

- *locale*: indicates whether we have to take into account the user's operating system regional configuration for the internationalization of the texts. In the SC case this parameter must have value true as it is one of the main characteristics of the new interface.
- *umm_remote_url*: indicates the URL in which the user administration module Publisher its methods using RMI. Specifying it is mandatory in order for the *RequestProcessor* can consult the execution permissions of the different actions.
- *check_permissions*: indicates if we are going to user the *RequestProcessor* before executing any action. Currently this feature is in development, so it must be given value false.

According to this, Strut's extended servlet's configuration in the web.xml is as follows:

```
<servlet>
   <servlet-name>
      action
   </servlet-name>
   <servlet-class>
      com.hp.ov.activator.mwfm.futuregui.servlet.AdvancedActionServlet
   </servlet-class>
   <init-param>
      <param-name>
         locale
      </param-name>
      <param-value>
         true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
```

```
            umm_remote_url
         </param-name>
         <param-value>
            //localhost:2000/usrmngr
         </param-value>
      </init-param>
      <init-param>
         <param-name>
            check_permissions
         </param-name>
         <param-value>
            false
         </param-value>
      </init-param>
      <load-on-start-up>
         1
      </load-on-start-up>
</servlet>
```

Alter this, in the area of the web.xml dedicated to the mapping of the servlets, we must copy the following:

```
<servlet-mapping>
   <servlet-name>
      action
   </servlet-name>
   <url-pattern>
      *.do
   </url-pattern>
</servlet-mapping>
```

This allows us to refer to Strut's actions with the .do extensions, in the same way that for the JSP we use the extension *.jsp*, for example.

19.6 Login

The user login system is carried out through a servlet called Login Servlet which gets its name from the user and his password, and communicates with the authentication module to authenticate him. Once the access is granted, it is obtained at any given moment the applications, menus, inventory views, and the operations on the views the user has access to, and other information.

When the authentication module tells the Login Servlet that the user is valid, the next information is obtained:

- Menu structure from the application environment the user has access to. This structure is kept in the session.
- Inventory views and operations the user has access to. This structure is also stored in the session.
- Datasources the user can access. Each of these datasources is stored in the session using its names as key.
- Also stored in the session is the username under the key "user".
- Apart from the previous, the session also keeps the different MWFM in a *HashMap* under the key "mwfms". Each wmfm uses as key its name (see the *auth.properties* file). Under the key "mwfm_session" is stored the default MWFM. In order to maintain backward compatibility, we also keep each MWFM in session individually, using its name as key.

To configure the Login Servlet we must indicate the name with which it will be mapped (it will necessarily be called login), the Java class that will implement the servlet (*com.hp.spain.futuregui.login.LoginServlet*) and the following parameters:

- *init_url*: indicates the URL to which we will redirect the user when the login process ends successfully. In the case of the SC, the URL by default is the one for the application environment (*/activator/jsp/future-gui/index.jsp?frst=true*). It is a mandatory parameter and cannot be null or empty.
- *superuser_init_url*: it ahs the same meaning as *init_url*, but in this case it only applies to suer and system users. If not specified, it takes the same value assigned to *init_url*.
- *future_gui_login_failure*: indicates the URL to which we will redirect the user when the login process fails. In the SC case this URL points to a JSP error page (*/activator/jsp/future-gui/loginError.jsp*). It is a mandatory parameter and cannot be null or empty.
- *future_gui_change_password*: indicates the URL to which we will redirect the user when his password has been expired.
- *use_random_color*: this parameter indicates whether or not we must use the eight colour palette of the interface. It is not mandatory and its default value is *true,* that is, the eight colour palette.
- *maxReturnedValues*: indicates the maximum number of results that a search can show. It is not mandatory and its default value is 2000 results.
- *spi_user_manager_rmi_url*: the RMI URL which provides methods to create users remotely. It is an optional parameter. Its default value if not specified is *//localhost:2000/userrmi*. This parameter becomes mandatory when there are more than one *diagnostic* JBoss instances running. When the specified URL is set to *localhost* or the equipment IP, the RMI will be started up by the Login Servlet allocated here. Since there can only be defined a single user management RMI, in this case any other Login Servlet allocated in any other IP should be configured to use the RMI configured here. The RMI service at this location provides also methods for action audit.

As is explained on top, the Login Servlet should be configured as is shown below:

```
<servlet>
   <servlet-name>
      login
   </servlet-name>
   <servlet-class>
      com.hp.spain.futuregui.login.LoginServlet
   </servlet-class>
   <init-param>
      <param-name>
         init_url
      </param-name>
      <param-value>
         /activator/jsp/future-gui/index.jsp?frst=true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         future_gui_login_failure
      </param-name>
      <param-value>
         /activator/jsp/future-gui/loginError.jsp
      </param-value>
   </init-param>
   <init-param>
```

```
      <param-name>
         future_gui_change_password
      </param-name>
      <param-value>
         /activator/SetNewUserPasswordActionFG.do
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         use_random_color
      </param-name>
      <param-value>
         true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         spi_user_manager_rmi_url
      </param-name>
      <param-value>
         //localhost:2000/userrmi
      </param-value>
   </init-param>
</servlet>
```

Once the servlet has been defined, in the section of the web.xml dedicated to the mapping of servlets we must copy the following:

```
<servlet-mapping>
   <servlet-name>
      login
   </servlet-name>
   <url-pattern>
      /login
   </url-pattern>
</servlet-mapping>
```

This mapping allows us to refer to the *Login Servlet* from JBoss's root directory (*/activator*) using the name *login*.

From this moment on the servlet invocations will be similar to:

http://localhost:8080/ep/login?username=xxx&password=yyy

19.7 Multiple JBoss instances

It is possible to start up different JBoss instances and establish different configurations for satisfying the client solutions. When there are more than one *diagnostic* JBoss instance running it is necessary to specify which one of them is going to provide the RMI used for the user management. Only one of the available JBoss instances can start it up.

Check the description of the *spi_user_manager_rmi_url* parameter specified in the *web.xml* file for the *Login Servlet* definition.

19.8 Flow interaction

The servlet which permit the interaction with the user during the flow execution is used extensively and it is a good idea to explain its configuration, in the same way we did for the Future Tree.

To perform the user interaction with a running workflow the flow of execution is paused and waits for new orders, and the *interact* servlet is invoked, whose mission is to generate a JSP where all the fields the user must fill in are shown.

To configure this servlet it is necessary to know the role of the following parameters:

- *customizeAskForNodeJSP*: indicates whether the interaction JSP should be generated or not.
- *webRoot*: indicates the directory where the interaction JSP generated by the servlet must be located.
- *fileSavedInfo*: it's only useful when *customizeAskForNodeJSP* is *true*. Indicates whether the JSP must be stored for later use.
- *mandatory*: text with which the mandatory parameters will be indicated. It is not mandatory and if no value is given the mandaroty parameters will appear in red. It generally has an asterisk (*) as value.
- *showAllInformation*: indicates whether all the information relating to the flow and the node must be shown in the JSP.
- *submit*: text that should appear in the Submit buttons. Currently this text is not used has it has become deprecated by Struts' internationalization.
- *clear*: text that should appear in the Delete buttons. Currently this text is not used has it has become deprecated by Struts' internationalization.
- *cancel*: text that should appear in the Cancel buttons. Currently this text is not used has it has become deprecated by Struts' internationalization.
- *allowCancel*: indicates whether the JSP should show the operation cancel option. If the value is false, the button will not be shown.

The servlet's configuration is as follows:

```
<servlet>
   <servlet-name>
      interact
   </servlet-name>
   <servlet-class>
      com.hp.spain.wflt.interact.PageGenerator
   </servlet-class>
   <init-param>
      <param-name>
         customizeAskForNodeJSP
      </param-name>
      <param-value>
         true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         webRoot
      </param-name>
      <param-value>
         C:/hp/jboss/server/diagnostic/deploy/hpovact.sar/activator.war
      </param-value>
   </init-param>
   <init-param>
        <param-name>
          fileSavedInfo
```

```
      </param-name>
        <param-value>
          true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         mandatory
      </param-name>
      <param-value>
         *
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         showAllInformation
      </param-name>
      <param-value>
         false
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         submit
      </param-name>
      <param-value>
         Enviar
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         clear
      </param-name>
      <param-value>
         Cancelar
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         allowCancel
      </param-name>
      <param-value>
         true
      </param-value>
   </init-param>
   <init-param>
      <param-name>
         cancel
      </param-name>
      <param-value>
         flujo_cancelado
      </param-value>
   </init-param>
</servlet>
```

Apart from the definition it is necessary to include the mapping of both servlets in the *web.xml*.

```
<servlet-mapping>
   <servlet-name>
      interact
```

```
    </servlet-name>
    <url-pattern>
       /interact
    </url-pattern>
</servlet-mapping>
```

19.9 Taglibs

A taglib allows the generation of code in a web page using user defined tags. It is formed by a TLD (Tag Library Descriptor) where the XML definition of the tags and its attributes is established, and a Java implementation for each tag, so the result is HTML code generated automatically in an easy way.

19.9.1 Taglibs belonging to Struts

The version 1.2.9 of Struts which is currently used in the SC provides various taglibs with several functionalities. For more detailed information about each of them refer to Strut's documentation.

The TLDs of these taglib are deployed in the WEB-INF and its definition inside the *web.xml* is as follows:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tags/struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-html</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-nested</taglib-uri>
    <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-tiles</taglib-uri>
    <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/table-taglib</taglib-uri>
    <taglib-location>/WEB-INF/table-taglib.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/block-taglib</taglib-uri>
    <taglib-location>/WEB-INF/block-taglib.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/button-taglib</taglib-uri>
    <taglib-location>/WEB-INF/button-taglib.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-displaytag</taglib-uri>
    <taglib-location>/WEB-INF/displaytag.tld</taglib-location>
  </taglib>
</jsp-config>
```

19.9.2 Belonging to the SC

The SC provides some taglibs that generate HTML code following the interface's own style. All the TLDs in this section can be found inside the ovsa41-utilities project and are deployed in the WEB-INF.

19.9.2.1 Button taglib

This taglib allows the generation of buttons following the style of the SC and shows the internal text internationalized.

The TLD is called *button-taglib.tld* and the necessary configuration is:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tags/button-taglib</taglib-uri>
    <taglib-location>/WEB-INF/button-taglib.tld</taglib-location>
  </taglib>
</jsp-config>
```

19.9.2.2 Table taglib

This taglib allows the generation of tables following the style of the SC.

The TLD is called *table-taglib.tld* and the necessary configuration is:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tags/table-taglib</taglib-uri>
    <taglib-location>/WEB-INF/table-taglib.tld</taglib-location>
  </taglib>
</jsp-config>
```

19.9.2.3 Block taglib

This taglib allows the generation of information request views for the application environment.

The TLD is called *block-taglib.tld* and the configuration needed is:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tags/block-taglib</taglib-uri>
    <taglib-location>/WEB-INF/block-taglib.tld</taglib-location>
  </taglib>
</jsp-config>
```

19.9.2.4 Combotext taglib

This taglib is a combination between a text field and a combo box. With it, any text may be typed into the text field, but there are some suggested options by default, as it happens with a combo box, which are displayed as they match the already typed text.

The TLD is called combotext-taglib.tld and the configuration needed is:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tags/combotext-taglib</taglib-uri>
    <taglib-location>/WEB-INF/combotext-taglib.tld</taglib-location>
  </taglib>
```

```
</jsp-config>
```

19.9.2.5 Display tag

This taglib is property of *Jakarta*, it has not been developed by HP. It allows the generation of tables with more features than the *table taglib,* as it allows the possibility of paginating the results and to order them by columns either in ascending or in descending order. Also it provides functionality to retrieve the results in different formats, such as PDF, CSV or XLS.

The TLD is called *displaytag.tld* and the configuration in the *web.xml* is:

```
<jsp-config>
  <taglib>
    <taglib-uri>/tags/struts-displaytag</taglib-uri>
    <taglib-location>/WEB-INF/displaytag.tld</taglib-location>
  </taglib>
</jsp-config>
```

19.10 Session timeout

The session timeout is defined in the *web.xml* and is the maximun amount of inactivity measured in minutes:

```
<session-config>
   <session-timeout>
      100
   </session-timeout>
</session-config>
```

19.11 Welcome page

The welcome page is configured in the *web.xml*. The system by default will search in all public directories, so that if a user types in a URL which doesn't match any specific page JBoss will try to find a page that matches the value entered here.

Here you can specify as many welcome pages as are needed.

```
<welcome-file-list>
   <welcome-file>
      login.html
   </welcome-file>
   <welcome-file>
      index.html
   </welcome-file>
</welcome-file-list>
```

19.12 Datasources

A datasource is a connection pool to the database. JBoss provides an easy way to define them in an XML file called *standalone.xml*.

By default the datasources are not kept in the user's session, they will only be accessible through the servlet context. This is due to the fact that the datasources are tied to the access permissions specified in the User Administration Module, where the mapping between datasources and applications are defined.

The user's session only stores the datasources that belong to applications that the user has been given permission to access (refer to the section on *Roles and Applications* from the *Permissions* chapter). Please refer to the documentation of the User Administration Module for more information.

It is also possible to define datasource alias in the file *alias.xml* that can be found below the directory <<ep.war>>/WEB-INF. This file allows the user to define an alias for a pre-existent datasource. This alias can be used in the same way as any other datasource defined in the system. Only the users with permissions to the original datasource can access to its alias.

The xml file has the next structure:

```
<alias-definition>
  <alias>
       <datasource-name>DatasourceOne</datasource-name>
       <datasource-alias>DatasourceTwo</datasource-alias>
  </alias>
  <alias>
       <datasource-name>DatasourceThree</datasource-name>
       <datasource-alias>DatasourceFour</datasource-alias>
  </alias>
</alias-definition>
```

19.13 Permissions

The permissions are specified in the User Administration Module and are established for each role the user belongs to.

The permissions granted to a user are verified only once: the moment when the user logs on. Therefore, if there is a change in the user's permissions while he is logged on, the user must log in again in order for the changes to take effect.

19.13.1 Users and Teams

Teams (i.e. Groups) are used to define groups of users with the same (or at least very similar) privileges. A group may have from zero to n users, and may be associated to one role (*futuregui*) or more. Each user must be assigned to a group, and there can't be any user associated to roles which are not associated to that group. If this situation happens, the user won't be able to log on the SC.

Teams may have administrators. There is no limitation on the number of administrators a team can have.

Administrators are allowed to create, update and remove users of their own group. They can manage their permissions to access roles, but they are not allowed to create, update or remove roles. This feature is only allowed for super users.

19.13.2 Roles and Teams

The relationships between roles and teams determine the roles which will be accessible for the users of the different teams. A user will have permission to access one or more of the roles belonging to his team, but at least it is necessary for him to have access to the *futuregui* role.

19.13.3 Roles and users

These are the first permissions that must be established. A user can be associated to any number of roles and vice versa. From this moment on, the rest of the permissions are established through the roles, and never through the user.

Since a user must belong to a team (and only to one team), it is not allowed to establish access to roles which are not associated to the user's team. If this situation is given, the user will not be able to log on the SC.

19.13.4 Roles and applications

The permissions between roles and applications determine the applications (and all the elements belonging to the applications, such as menus and datasources) to which the roles will have access.

From these permissions are determined the datasources that the user will keep in the session after logging in: those datasources that are associated to the applications the user has access to.

19.13.5 Roles and menus

The permissions between roles and application menus are established in two ways. One is this, making the relationship directly between the roles and the menus, but as the menus belong to the applications it is also necessary establish the permissions to access the application that corresponds to the menu, as was explained in the previous section.

These permissions only affect the application environment, which is where the application menus are used, but they are irrelevant to the inventory.

19.13.6 Roles and inventory views

The inventory views the user has access to, are defined here. Thus, a normal user will opening the inventory window will only have access to those views associated to one (or more) of his roles.

19.13.7 Roles and inventory view operations

The permissions between roles and the operation types belonging to each inventory view are set here. Thus, only those operations of those operation types associated to one (or more) of the user roles will be accessible for each user.

19.14 GUI

There are some features of the Inventory's interface which can be configured.

19.14.1 Changing view and status

The *index.jsp* file can receive two optional parameters, which are typically specified along the different mappings in the application *struts-config.xml* file:

- *viewName*: indicates the name of the new view which is going to be loaded. If this menu has any menu attached, they will be preloaded, too. The JSP file associated to this view, if any, is obtained from database.
- *fjsp*: indicates the URL of the initial status JSP file which has to be loaded in the status space, if any. If no *fjsp* parameter is specified, a default JSP file is loaded, called *blank.jsp*, and provided with the SC. This default JSP file has no representation.

19.15 Access to the Inventory UI: cross launch

The Inventory window is opened in a new navigator's window from the SC's applications environment by clicking the *Inventory → Open* menu. This option will not be available for a user if that user is not associated to the role *inventory*. Contact your system administrator to get these rights.

The Inventory UI accessed is the HPSA's using cross launch, so parameters regarding this cross launch must be configured in the *crosslaunch.properties* file:

- *hpsa.ip*: the IP where HPSA is running. Note that HPSA must be started up to access the Inventory.
- *hpsa.port*: the port where HPSA is running, typically 8080.

19.16 Workflow Launcher

The next sections explain every needed configuration for the WFLT. This configuration is set using the *wflt.properties* file, which in a Windows environment is placed below the "hp\jboss\server\diagnostic\deploy\hpovact.sar\activator.war\properties" directory.

19.16.1 SOSA Remote Interface

If any workflow has to be started up with SOSA the next parameters are needed in the *wflt.properties* in order to be able to invoke the SOSA remote interface:

- *wfltmanager.service.host*: The computer's IP where SOSA is running.
- *wfltmanager.service.port*: The port which is being used by SOSA.
- *wfltmanager.service.name*: The RMI service used by SOSA.

Here we can see an example of this configuration where SOSA is running locally:

```
wfltmanager.service.host = 127.0.0.1
wfltmanager.service.port = 1119
wfltmanager.service.name = RmiWFLTService
```

19.16.2 Not interactive step names

In the *wflt.properties* file there can be specified the different step names which shall never be considered as interactive nodes, so any node which name starts by any of this configured names will never be an interactive node.

These parameters must be numbered starting from 0:

```
wflt.not.interaction.step0 = Activate
wflt.not.interaction.step1 = Fix
wflt.not.interaction.step2 = Test
wflt.not.interaction.step3 = Lock
wflt.not.interaction.step4 = Invoke
wflt.not.interaction.step5 = Wait
```

19.16.3 ECP Command tracking configuration

Some parameters are necessary to track the ECP commands. They must be specified in the *wflt.properties* file:

- *wflt.provider.url*: The URL where the ECP JMS Server has been launched
- *wflt.max.commands*: The maximum number of commands which will be stored in each launched activation.
- *wflt.ecp.jms.connection.factory:* The JMS Connection factory, this parameter it's not mandatory. By default it will be "TopicConnectionFactory". This parameter must be the same as the one configured in the ECP.
- *wflt.ecp.jms.destination.id* = The JMS Destination Id, this parameter it's not mandatory. By default it will be "/dynamicTopics/ECP.MainTopic". This parameter must be the same as the one configured in the ECP.

Here we can see an example of the values assigned to these parameters:

```
wflt.provider.url = tcp://16.38.0.136:4001
wflt.max.commands = 20
```

As it happens in the previous section with the not interactive step names, those node names in which the ECP command tracking has to be performed must be specified using the properties file. For that, there are some numbered parameters starting from 0 (as it can be seen in the example below) where the beginning of the node names which must be considered as command tracking nodes are specified:

```
wflt.activation.step0 = ECP
wflt.activation.step1 = Command
wflt.activation.step2 = Activate
```

19.16.4 CCWF for the WFLT

Each CCWF must be noticed by the WFLT to be able to track workflows on the different modules. Thus, for each defined CCWF, there must be a properties file with the name of that CCWF where it's RMI URL will be specified using the next parameters:

- *concurrentworkflow.service.host*: the IP of the MWFM host.
- *concurrentworkflow.service.port*: the MWFM port.
- *concurrentworkflow.service.name*: the name of the CCWF remote service.

Note that these three parameters must be the same as the ones specified in the *remote_url* parameter of the CCWF.

For example, in the example of the previous section the name of the CCWF is *localmwfm*. That means that there must be a *localmwfm.properties* file in the properties directory of the JBoss with these contents:

```
concurrentworkflow.service.host = localhost
concurrentworkflow.service.port = 2000
concurrentworkflow.service.name = concurrent_workflows
```

If no properties file is found for a given MWFM name, then the default URL will be used to invoke the CCWF:

//localhost:2000/concurrent_workflows

19.17 ECP Console

19.17.1 Permissions

There are two kinds of permissions that must be managed in order to use the ECP Console: the command filters, which allow executing the different typed commands, and the command scripts, explained in previous sections.

The SC provides an administration GUI to manage these permissions. Check the sections about the ECP Console in the document *"HPSA Extension Pack – Solution Container - User Reference"* for further information.

19.17.2 Command filters

A command filter is a regular expression which matches every typed command. Only those commands that match a regular expression will be accepted. The other ones will become forbidden and an error message will be displayed for the user.

Command filters are associated to users. Any typed command is matched with every user's command filter and, if it matches one of them then it is accepted and executed.

Check the sections about the ECP Console in the document *"HPSA Extension Pack – Solution Container - User Reference"* for further information.

19.17.3 Scripts

The accessible command scripts must be associated to the user and the host. Other way they will never be displayed in the ECP Console.

Check the sections about the ECP Console in the document *"HPSA Extension Pack – Solution Container - User Reference"* for further information.

20 Start-up

The SC is started up along with HPSA, so once HPSA has started up properly the Solution Container will be available as well as HPSA.

21 API Reference

21.1 General information request views

This kind of views allows the user to compose his searches, selecting some available attributes and operations and specifying the wanted value.

There are some JavaScript objects developed for the SC which provides an easy way to generate these views. The JavaScript file where this objects are coded is imported in the *index.jsp* file, so there is no need to import again the JavaScript file in the JSP of the view.

There are four objects involved in this view: *DateFormat*, *Operation*, *Field* and *Search*.

The API of these objects is:

k. DateFormat object

This static object sets the date format which will be used if the calendar is needed. There are four possible formats:

- DDMMYYYY (default)
- DDMMMYYYY
- MMDDYYYY
- MMMDDYYYY

It is also possible to show the hour or not. The hour can be shown in "12" or "24" (default) format.

Methods:

- *showTime(boolean)*: indicates if the time must be shown with date field attributes. By default, time is not included.
- *setFormat(format)*: indicates which of the four possible formats will be used. Possible values for the parameter are *DateFormat.DDMMYYY*, *DateFormat.DDMMMYYYY*, *DateFormat.MMDDYYYY* and *DateFormat.MMMDDYYYY*.
- *setHourFormat(hourFormat)*: sets the hour format that will be used with all date field attributes. Possible values are *DateFormat.HOURS_24* (default) and *DateFormat.HOURS_12*.

l. Operation object

This object gathers possible values associated to different types of attributes. It is used to simplify the specification of the operators for a given attribute. If an attribute has no operations explicitly attached, the Operation object provides a role of default operations for it, using the attribute's type to gather them.

Defined operations are:

- Operation.LESS_THAN = "<";
- Operation.LESS_EQUAL_THAN = "<=";
- Operation.GREATER_THAN = ">";
- Operation.GREATER_EQUAL_THAN = ">=";
- Operation.EQUAL = "=";
- Operation.NOT_EQUAL = "!=";

- Operation.LIKE = "LIKE";
- Operation.P_LIKE = "%LIKE";
- Operation.LIKE_P = "LIKE%";
- Operation.P_LIKE_P = "%LIKE%";

But an attribute can have any other operation even though it is not defined here. Note that for the view an operation is only a String, it has any sense to it because this javascript code uses this operations to show them to the user, not just to perform the operation.

m. *Field* object

The Field object wraps each possible attribute the Search can manage.

To define each Field, the next parameters are needed:

- *name*: the text that will be shown
- *attName*: the name of the attribute expected by the action which is going to perform the Search.
- *type*: the type of the attribute. The possible values for this parameter are:
    - Field.SELECT: values for this Field are selected from a combo.
    - Field.STRING: value for this Field is a text.
    - Field.BOOLEAN: value for this Field is a boolean.
    - Field.NUMBER: value for this Field is a number.
    - Field.DATE: value for this Field is a date. If this is the case, the JSP must import the datetimekeeper.js file.
    - Field.IP: value for this Field is an IP.

The public methods for this object are:

- *addOperation(type)*: Adds an operation to this Field. Operations are automatically added to a Field according to it's type when no operation is attached explicitly. For instance, if a Field belongs to the Field.BOOLEAN type and no operation is attached to it using neither the addOperation(op) nor the addOperations(aOps) methods, the Field.BOOLEAN type's operations are attached automatically. The parameter *op* is the operation to add. Any String is valid because it is not checked.
- *setValidationType (type)*: Adds a possible value for this Field. This method is only allowed for Fields belonging to the Field.SELECT type. Otherwise, an error is shown. The parameter *value* is the possible value for this Field.
- *addValue(value)*: Sets the validation type of this Field. This is used to validate the format entered for the values of a Field. For instance, it allows to validate a String as a Number, or checks if a number entered by the user is really a number. The parameter *vType* the validation type for this Field. The possible values for this parameter are the same as the types of a Field.

Search **object**

The Search object stores and shows the view.

The next parameters are needed to define a Search object:

- *title*: the title of the Search.
- *action*: the action that will be invoked to perform the Search.

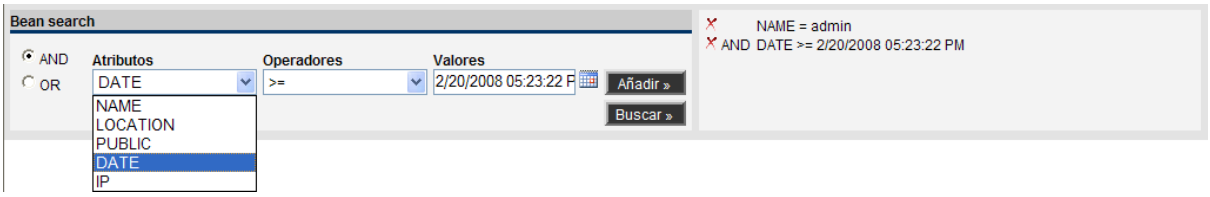
The public methods for this object are:

- setAddButtonText(text): Sets the Add button's text. This allows to internationalize texts. The parameter text is the text of the Add button.
- setSearchButtonText(text): Sets the Search button's text. This allows to internationalize texts. The parameter text is the text of the Search button.
- setFieldsText(text): Sets the text for the Fields combo. This allows to internationalize texts. The parameter text is the text for the Fields combo.
- setOperatorsText(text): Sets the text for the Operations combo. This allows to internationalize texts. The parameter *text* is the text for the Operations combo.
- setValuesText(text): Sets the text for the Values input. This allows to internationalize texts. The parameter text is the text for the Values input.
- addField(field): Adds a possible Field to the Search. The parameter field is the Field to be added.

Example

The next code generates a view like the one in the figure below.

```
var a = new Search("Bean search", "/activator/mySearch.do");
var f = new Field("NAME", "name", Field.STRING);
f.addOperation(Operation.EQUAL);
f.addOperation(Operation.LIKE);
a.addField(f);
f = new Field("LOCATION", "location", Field.SELECT);
f.addOperation("%");
f.addOperation("BETWEEN");
f.addValue("UNO");
f.addValue("DOS");
a.addField(f);
f = new Field("PUBLIC", "public", Field.BOOLEAN);
f.addOperation(Operation.LESS_EQUAL_THAN);
f.addOperation(Operation.LESS_THAN);
a.addField(f);
f = new Field("DATE", "date", Field.DATE);
a.addField(f);
f = new Field("IP", "ip", Field.IP);
a.addField(f);
a.write();
DateFormat.setFormat(DateFormat.MMDDYYYY);
DateFormat.showTime(true);
DateFormat.setHourFormat(DateFormat.HOURS_12);
```



21.2 Information request views: Block Taglib

These kinds of views, typical of element searches, consist of a form with several fields where the user must enter the information to interact with the system.

To avoid the need for the programmers to be concerned about following the style and appearance of the SC the Block taglib has been developed, which allows the creation of centred search fields, in columns of

one or two fields, and with a maximum of 5 fields (more than 5 can also be shown, but the user's screen only has 1024 pixel width resolution some of the columns will be lost).

The Block Taglib is made out of the following tags:

Space Tag

This tag indicates the beginning and end of the block space. It does not have any values; its usefulness is limited to marking the border of the taglib. Every use of the Block taglib must begin and end with this tag.

```
<block:space>
   ...
</block:space>
```

It accepts the following attributes:

- *align*: indicates the default alignment of the text. It can take the values *left* (default value) *center* and *right*.
- *title*: indicates the title of this view. The title is showed on an upper bar over the blocks of the view.
- *key*: indicates the key of the internationalization file where the title is found. This attribute has no sense if the *bundle* attribute is not specified.
- *bundle*: indicates the name of the internationalization file where the title is found. This attribute has no sense if the *key* attribute is not specified.

Wall Tag

The blocks must be placed in columns or walls, of one or two blocks. This tag also has no values; it is used to delimit the roles, but is necessary both when the role has one block or two.

```
<block:wall>
   ...
</block:wall>
```

It can have the following attributes:

- *width*: it shows the width of the column. If no value is assigned a default one is taken, but it's important to take into account that it must match the corresponding width attribute from the Block tag.

Block Tag

This is the tag that establishes the block itself. If must always appear inside a *wall* tag.

It can have the following attributes:

- *title*: the block title. It is the text that is shown just above the search field.
- *key*: it indicates the entry of a property file that contains the internationalized title for this block. If this attribute is present, then the *bundle* tag must also have a value.
- *bundle*: it indicates the package where the property file from which the *key* value is obtained. Therefore the *key* attribute must have a value.
- *verticalAlign*: the vertical position of the block. It can take three possible values: top, center or bottom. It is only useful when there is only one block inside the column. If there are two blocks this attribute is irrelevant.
- *width*: indicates the block width. If it is not indicated a default value is assigned, but if it does have a value then it's important that it corresponds to the similar attribute from the *wall* tag.

Example

The following example generates a view with two columns, two blocks in the first column and one in the second:


```
<block:space>
   <block:wall>
      <block:block title="Nombre">
         <input type=text value="">
      </block:block>
      <block:block title="Tipo de equipo">
         <select style="width:145">
            <option value="1">Riverstone</option>
            <option value="2">Alcatel</option>
            <option value="3">Otro</option>
         <select>
      </block:block>
   </block:wall>
   <block:wall>
      <block:block title="Localización" verticalAlign="center">
         <select style="width:145">
            <option value="1">Madrid</option>
            <option value="2">Valencia</option>
            <option value="3">Chimbamba</option>
         <select>
      </block:block>
   </block:wall>
</block:space>
```

21.3 Buttons: Button taglib

The application environment follows the idea that you can only use buttons in the views, but never in the status. To do any operation from a status JSP we have the status menu.

To generate buttons that blend with the SC interface the Button Taglib has been developed composed of just one tag:

***Button* tag**

It is the only tag of the taglib. It generates a button that matches the look & feel of the SC. It accepts the following attributes:

- *value*: the text that must be shown with the button. It automatically receives the » prefix (unless the string value that is given with this attribute already has the symbol). If no specific text is given for the button then the default value is the double bigger-than symbol.
- *key*: It indicates the entry for a property file that contains the internationalized text for this button. When this value is defined it is mandatory to also have a value for the *bundle* attribute.
- *bundle*: indicates whereabouts of the package where the property file from which the value set for *key* is obtained. Therefore, it's necessary to indicate a value for the *key* attribute.

- *onclick*: string with the invocation that must be produced when this event is detected on the button. If this attribute contains quote symbols (") (not apostrophes, these aren't a problem) are substituted by apostrophes.
- *width*: the button width. If none is indicated, then the button's size is resized depending on the text.
- *noRaquo*: boolean that indicates whether the prefix » must appear in front of the button's text. The default value is false, which indicates that this value must be shown.
- *type*: string that indicates the button type. It can take the values: "button", "submit" and "reset". The appearance of both is similar, but the first creates a traditional button (*<input type=button>*) and the second a submit button for the associated form (<input type=submit>). If no value is given for this attribute, then the default value is "button".

Examples

Basic button with no text

Generates a button similar to that in figure 13.

The code necessary to generate this button is:

```
<btn:button onclick="alert('Hello, world!!!');"/>
```

Button with text

Generates a button similar to that in figure 14.

The code necessary to generate this button is:

```
<btn:button value="Say Hello" width="100" onclick="alert('Hello,
world!!!');"/>
```

Button with internationalized text

We can generate a button such as the one in Figure 2 that shows the text in the language associated with the user.

Let's suppose that in the struts-config.xml file of the application we are developing we have mapped the property file with the name ApplicationResourcesEJ. The button's internationalization would be:

```
<btn:button key="button.salutation" bundle="ApplicationResourcesEJ"
width="100" onclick="alert(Hello, world!!!');"/>
```

21.4 Information Presentation Views

The information presentation views regularly show a great amount of data that due to the confinement of the space should be condensed as much as possible.

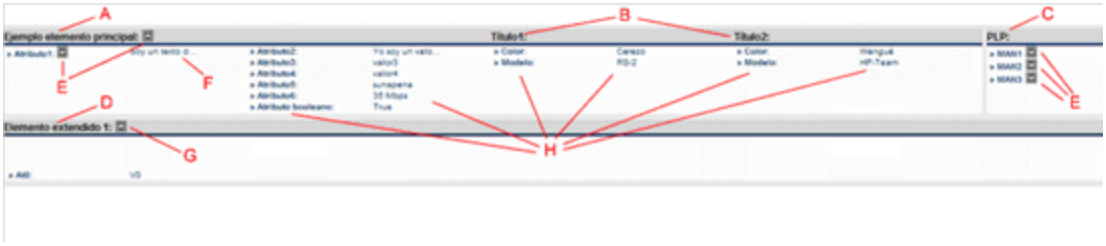
To aid with the development of these kind of views, whose design complication is quite high, an API exists (*menuInfo.js)* based on JavaScript objects that are in charge of showing the data correctly. This file is part of the *index.jsp* page and because all the views correspond to JSPs that are embedded inside *index.jsp* the access to the API objects contained in *menuInfo.js* is direct from the view JSP.

An Information Presentation View looks similar to figure 15.

As can be seen, this representation of the information is divided in three big main blocks:

- The main information is the one that will grab the user's attention initially and is shown at the beginning when the view is loaded.
- The secondary information can be composed of at most of two elements, and refer first of all to the information associated with the main information. By default the first of these is always shown first, but you can see one or the other by clicking on the column titles (in the image Titulo1 and Titulo2) of the main information. The secondary information does not have to be present necessarily, and when none is specified the main information is extended automatically to occupy the whole width of the screen.
- The extended information contains information that does not fit in any of the other or that for whatever reason is better shown in this way. It can be associated to only one attribute and not to a main or secondary element. This information is hidden when the view loads and in order to show it it is necessary to click the button for showing/hiding this particular extended information.

The figure 16 shows the view elements that interact with the user:



where:

- A: Main element title.
- B: Column titles, that show one or the other secondary information
- C: Title for the secondary element that is being shown
- D: Title for the extended element that is being shown.
- E: Buttons for the showing/hiding of the extended information.
- F: Truncated text, which can be seen completely by putting the cursor over it.
- G: Button for the hiding of the extended information that is being shown.
- H: Attributes, composed of name/value pairs.

The API for the Information Presentation View is composed of four JavaScript objects:

- *MainMenuInfo*: generates the main element.
- *SecondaryMenuInfo*: generates the secondary elements.
- *ExtMenuInfo*: generates the extended information.
- *MenuInfoWriter*: is in charge of showing everything on screen.

MainMenuInfo object

The *MainMenuInfo* object constitutes the core of the information presentation views. It is the only object, apart from *MenuInfoWriter*, that must appear necessarily. The *SecondaryMenuInfo* and *ExtMenuInfo* instances, however, can exist or not depending no the needs of the data to be shown.

This object provides a matrix representation of the name-value pairs that form the view information. Once all the matrix cells have been filled in the data will be shown on the web page as a table.

A simple example of use of the *MainMenuInfo* object is shown below. In it the object constructor is invoked, a title is assigned and several name-value attributes are established.

```
// Constructor
var mmi = new MainMenuInfo();
// Título
mmi.addTitle("My object title", null);
// Atributos
mmi.addAttribute("First name att", "First value", 0, 0, null);
mmi.addAttribute("Another name", "Another value", 2, 0, null);
...
mmi.addAttribute("Last name att", "Last value, allocated at the right side",
6, 1, null);
```

Constructors

- *MainMenuInfo()*

Methods

- *public void addTitle(String title, ExtMenuInfo extensibleObj)* – Sets the main *title* for the object.

  Parameters:

  - *title* - the object's title.
  - *extensibleObj* - the *ExtMenuInfo* object associated. If null, this object won't have any extended information associated with it. If it is not null then a button appears to the right of the object's title which will allow hiding or showing the associated extended information.
- *public void addColumnTitle(String title, String/int columnNumber)* – Sets the titles for the fourth or sixth columns of the object. The presence of these titles allows showing one of the two possible secondary information available (see SecondaryMenuInfo).

  Parameters:

  - title – the title for the fourth or sixth columns of this object. If null, the default title shown is "Column title not found".
  - columnNumber – the number of the column the title corresponds to. It cannot be null. In fact, it can only acquire two possible values: 4 or 6, as only these two columns can have titles.
- *public void addAttribute(String title, String value, String/int nameX, String/int nameY, ExtMenuInfo extensibleObj)* – Adds a name-value pair to the object. The name of the attribute is set in the matrix cell corresponding to the position indicated by the parameters *(nameX, nameY)*, while the value is situated in the cell *(nameX + 1, nameY)*. Each name-value pair occupies two consecutive cells.

  If the parameter (*title parameter*) or the value (*value parameter*) where null they would be replaced for the text "Name not found" or "Value not found", but the process does not stop, it simply shows a warning message to inform the programmer of the situation.

If the coordinates *nameX* or *nameY* where null then the process stops, any future invocation to any other method of the object is ignored and a message is shown to warn about the problem.

The same thing happens if the coordinates exceed the limits allowed. The coordinate established by *nameX* can take values from 0 to 7, both included, and *nameY* between 0 and 5, both included. This means that the matrix representations of this object have at most 8 columns and 6 rows.

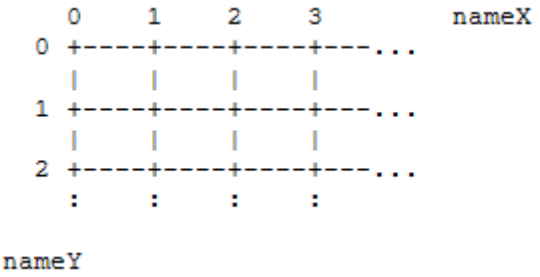
As each name-value pair occupies two cells, the *nameX* coordinate must necessarily be an even number, starting the count with 0. For example, the coordinates (0,0), (0,1) or (2,4) are valid, but the coordinate (1,2) is not valid. If this is not followed, a future invocation of this method might result in an overwriting of the name or value established here. If the following JavaScript piece of code is examined:

```
myMainMenuInfoObj.addAttribute("nameAtt0", "valueAtt0", 0, 0, null);
myMainMenuInfoObj.addAttribute("nameAtt1", "valueAtt1", 1, 0, null);
```

the result would be incorrect, as the *valueAtt0* value of the first line has been located in the (1,0) coordinate, the same coordinate that the second line sets with the name *nameAtt1*. The correct way to do it is:

```
myMainMenuInfoObj.addAttribute("nameAtt0", "valueAtt0", 0, 0, null);
myMainMenuInfoObj.addAttribute("nameAtt1", "valueAtt1", 2, 0, null);
```

The matrix that is filled with name-value pairs is similar to:



Parameters:

- *name* - the attribute name.
- *value* - the value associated with the name of the attribute.
- *nameX* - the row where the name must be situated.
- *nameY* - the column where the name must be situated.
- *extensibleObj* – the object with the extended information associated with this attribute. If the attribute does not have extended information it will be *null*. If not, to the right of the button a button will appear to the right of the attribute name that will allow the hiding/showing of the extended information.

- *public void addScrollableCell(String title, int initRow, int initColumn, int width, int height, String fromTextareaId)* – Shows a multiple line field (a non editable textarea) situated in the cells whose coordinates are given by initRow and initColumn and whose width and height are, respectively, width y height. Any name-value pair specified in the cells that this element overshadows will be hidden by it.

  This element is used when the type of information to be shown consists of a very long text and that might include any mix of characters such as colons or line feed.

  To avoid having to escape the potentially problematic characters as the ones cited before, a final parameter has been included in this method that corresponds to the identifier of the hidden *textarea* where the information we want to show here should have been stored previously.

Let's put an example. We need to show an attribute "Observations" that contains the following:

```
This text
"contains" dangerous characters
and line feeds.
```

What we will do is to write this in a *textarea* hidden from our view. Like this:

```
<textarea id="myObservations"
style="visibility:hidden"> This text
"contains" dangerous characters
and line feeds.
</textarea>
```

This, obviously, must be outside any *<script>...</script>* tags.

Later, when we specify our *MainMenuInfo* object and we invoke the *addScrollableCell* method, we pass as *fromTextareaId* parameter the value "myObservations". Like this:

```
mmi.addScrollableCell("Observations", ...,
"myObservations");
```

Parameters:

- *title* - the multi-line attribute title. This string is shown over the multi-line text. If the textbox is situated in row number 0 the title is not shown.
- *initRow* - the first row where the multi-line attribute will be shown.
- *initColumn* - the first column where we want to show the multi-line attribute.
- *width* - the number of columns (width) of the multi-line attribute.
- *initRow* – the number of rows (height) of the multi-line attribute.
- *fromTextareaId* – the hidden *textarea* identifier from with the text of this multi-line attribute will be copied.

SecondaryMenuInfo **object**

This object makes reference to the secondary information that can be shown optionally in the right panel of the information representation view. Due to the fact that it is optional, when no object of this type is specified for the view in the main information contained in MainMenuInfo it expands to occupy the whole view's width.

The secondary information is shown vertically, like a one column table.

As can be seen from the MainMenuInfo specification it is possible to establish two different instances of this object with completely separate secondary information. At each moment only one of them will be visible. The MenuInfoWriter object establishes which of them will be visible when the page loads and which will remain hidden at the beginning at the start.

A basic example of secondary information specification is:

```
<script>
// Constructor
var smi = new SecondaryMenuInfo();
// Title
smi.addTitle("Networks");
// Atributos
smi.addAttribute("Network 1", null, 0, null, null);
smi.addAttribute("Network 2", null, 1, null, null);
smi.addAttribute("Network 3", null, 2, null, null);
...
smi.addAttribute("Network N", null, N, null, null);
</script>
```

Constructors

- *SecondaryMenuInfo()*

Methods

- *public void addTitle(String title)* – Sets the main title of the secondary information.

  Parameters:

  - *title* – the secondary information title.

- *public void addAttibute(String name, ExtMenuInfo extensibleObj, int position, String action, String target)* - Adds an element to the secondary information.

  Parameters:

  - *name* - the name of the element.
  - *extensibleObj* - the object with the extended information associated with this element, if any is given. When none is specified the value for this parameter is *null*. The existence of extended information associated to an element results in the presence of a button to show/hide this information.
  - *position* – the element's position inside the element column that constitutes the extended information. It can be *null*, in which case the element is added to the end of the existing ones.
  - *action* – indicates the URL or JavaScript function that must be invoked when the user clicks on the element. It can be *null*, in which case clicking on the element will have no effect.
  - *target* – indicates the physical place where the URL must be shown when clicking on the element. It can be null, in which case the default value is "_self", that is, the same page we are in, which would be result in a change of view. The possible values for this attribute are:
    - "_self": the URL will appear in the same window we are in, which will result in a change of view.
    - "_blank": the URL will appear in a popup window.
    - "fjsp": the URL will appear in the space reserved for status. "fjsp" is the iframe name dedicated for this use. This results in a status change.
    - "_js": the URL corresponds to a JavaScript function that will be invoked when the user clicks on the element.

ExtMenuInfo object

This object allows the representation of extended information of other objects or of other object's elements.

It is not shown from the start, but is spread below the main information when it is needed.

The way to represent the information is through a matrix, in the same way as happened with the main information (see MainMenuInfo), and is also composed of name-value pairs.

A simple example for this object is:

```
var emi = new ExtMenuInfo();
emi.addTitle("System components ");
emi.addAttribute("Port 1", "Gigabyte", 0, 0);
emi.addAttribute("Port 2", "Gigabyte", 0, 1);
emi.addAttribute("Network card", "Ethernet", 0, 2);
```

Constructors

- *ExtMenuInfo()*

Methods

- *public void addTitle(String title)* – Sets the main title for the secondary information.

  Parameters:

  - *title* - the title for the secondary information.
- *public void addAttribute(String name, String value, int nameX, int nameY)* - Adds a new name-value pair to this extended information which will be put on the cell whose coordinates are *(nameX, nameY)*.

  Parameters:

  - *name* - the name of the attribute. If null, the default value is "Name not found".
  - *value* - the attribute's value. If null, the default value is "Value not found".
  - *nameX* – the *X* coordinate where the name of the attribute will be set. As the value will be set in the next cell to the right, the *X* coordinate implicit for the value will be *nameX + 1*. It cannot be null. The possible values go from 0 to 7 both included.
  - *nameY* – the *Y* coordinate is the attribute's name. As the value will be shown in the next cell, the *Y* coordinate is implicit for the attribute's value and will have the same value *nameY*. It cannot be null, the possible values go from 0 to 3, both included.
- *public void addScrollableCell(String title, int initRow, int initColumn, int width, int height, String fromTextareaId)* - Shows a multi-line field (a non editable *textarea*) situated in the cell whose coordinates are given by *initRow* and *initColumn* and whose width and height are *width* and *height*. Any name-value specified that this element will overshadow will remain hidden.

  This element is used when the information type we need to show can have a very long text that can include any kind of characters, such as line feed or double quotes.

  With the goal to escape the potentially problematic characters such as the ones mentioned before, this method includes a parameter that corresponds to the hidden *textarea* where the text to be shown here will have been previously set.

  Lets show an example. We need to show the "Observations" attribute that contains the following:

```
This text
Contains "dangerous" characters
and line feeds.
```

  What we will do is write this in a textarea hidden from our view. Like this:

```
<textarea id="myObservations"
style="visibility:hidden"> This text
Contains "dangerous" characters
and line feeds.
</textarea>
```

  This, obviously, must appear outside any *<script>...</script>* tags.

  Later, when we are specifying our ExtMenuInfo object and we invoke the *addScrollableCell* method, such as the *fromTextareaId* parameter we will send "myObservations". Like this:

```
emi.addScrollableCell("Observations", ...,
"myObservations");
```

  Parameters:

- *title* - the multi-line attribute title. This string is shown over the multi-line text. If the textbox is situated in row number 0 the title is not shown.
- *initRow* - the first row where the multi-line attribute will be shown.
- *initColumn* - the first column where we want to show the multi-line attribute.
- *width* - the number of columns (width) of the multi-line attribute.
- *initRow* – the number of rows (height) of the multi-line attribute.
- *fromTextareaId* – the hidden *textarea* identifier from with the text of this multi-line attribute will be copied.

***MenuInfoWriter* object**

This object doesn't have its own visual representation; it is in charge of showing the different objects defined.

It only has one write method that takes care of invoking in the right way the different similar names that form the View Representation.

Supposing a MainMenuInfo object has been defined, stored in mmi, and two SecondaryMenuInfo objects, stored in smi and dmi respectively, the way to use this method is:

```
new MenuInfoWriter(mmi, smi, dmi).write();
```

Constructors

- *MenuInfoWriter(MainMenuInfo mainMenuInfo, SecondaryMenuInfo secObj1, SecondaryMenuInfo secObj2)*

  Parameters:

  - *mainMenuInfo* – the *MainMenuInfo* object's view representation information.
  - *secObj1* - the *SecondaryMenuInfo* object's view representation information that must be shown initially.
  - *secObj2* - the *SecondaryMenuInfo* object's view representation information that must be hidden initially.

Methods

- *public void write()* – Shows the view's representation on screen.

Examples

Only with main information

For this example we are going to create a view's information representation made up of only main information.

Let's suppose that we simply have to show a user's data: username, description, real name, company, whether he is a restricted user or not and his preferred language.

Let's remember we are developing a view's JSP and therefore, that it must not constitute a whole web page, as it will be embedded inside the *<body>…</body>* tags of *index.jsp*. If we include these tags inside the view's JSP the final result will be an error, so it is important to assume that we are developing the body or the *index.jsp* already.

Let's remember also that *index.jsp* already includes in its header the invocation for the JavaScript file (*menuInfo.js*) that contains the necessary objects to generate the information representation views, so we don't need to include them again in our view's JPS.

Therefore, the only thing we must do in this case is to invoke the main information element's constructor (MainMenuInfo), and to establish a title for the information we want to show and to add the necessary attributes in the preferred positions.

```
<script>
// Constructor invocation
var mmi = new MainMenuInfo();
// Setting the title
mmi.addTitle("User data", null);
// We add the attributes starting from the leftmost
mmi.addAttribute("Nombre", "operador", 0, 0, null);
mmi.addAttribute("Nombre real", "John Smith", 0, 1, null);
mmi.addAttribute("Compañía", "HP", 0, 2, null);
mmi.addAttribute("Descripción", "Operador de sistemas", 0, 3, null);
mmi.addAttribute("Restringido", "No", 0, 4, null);
// We put the language in the first cell of the second column
mmi.addAttribute("Lenguaje", "Castellano", 2, 0, null);
// We invoke the object that composes and writes or view.
new MenuInfoWriter(mmi, null, null).write();
</script>
```

With this code, the final result is shown below.

Main information with the extended view

In this case we are going to complicate the view's representation a little by establishing two possible pieces of extended information.

In the previous example we were showing user's information: username, real name, etc., but now we also want to show the information that doesn't have to appear all the time but which can be consulted when needed, so we use extended information. Therefore, we can establish as extended information the user's measurements, his height, size and weight.

But we also want to be able to consult the extended information about the company he works for, so for this attribute we will associate more extended information where we will be able to consult the antiquity, achievements and things like this.

The first thing we have to do is to define both pieces of extended information. For this we make use of the ExtMenuInfo object.

```
<script>
// We define the extended information with the user's measurements
// We invoke the constructor
var emi0 = new ExtMenuInfo();
// We establish the title for this extended information
emi0.addTitle("User's measurements");
// We establish the height and weight in the first cells of the first //
column

emi0.addAttribute("Height", "185 cm", 0, 0);
emi0.addAttribute("Weight", "80 kg", 0, 1);
// We define the extended information with all the company data
// Invoke the constructor
var emi1 = new ExtMenuInfo();
// Set the title for this extended information
emi1.addTitle("Company data");
// We set the information we need for the company
emi1.addAttribute("From", "1902", 0, 0);
emi1.addAttribute("Country", "EEUU", 2, 0);
```

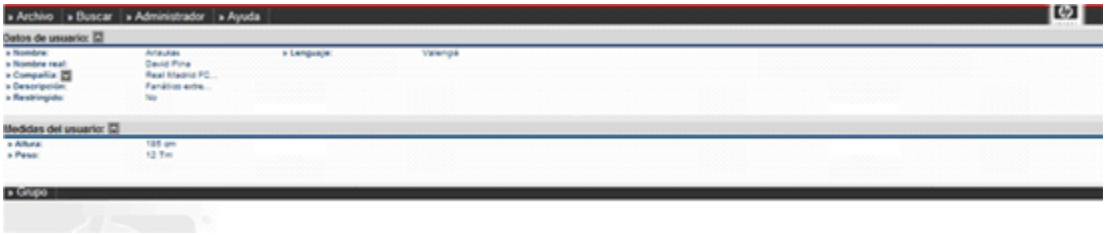
```
emi1.addAttribute("State", "California", 2, 1);
emi1.addAttribute("Profession", "Informática", 2, 2);
emi1.addAttribute("Activation", "Madrid", 4, 0);
emi1.addAttribute("City", "Las Rozas", 4, 1);
emi1.addAttribute("Address ", " Vicente Aleixandre", 4, 2);
emi1.addAttribute("Department", "Telco", 4, 3);
</script>
```

Now we will define the main information in a similar way we did for the first example, but associating the extended information we just defined.

```
<script>
// Constructor invocation
var mmi = new MainMenuInfo();
// We establish the title and we associate the extended user
// information
mmi.addTitle("User data", emi0);
// We add the attributes starting with the leftmost column
mmi.addAttribute("Username", "operador", 0, 0, null);
mmi.addAttribute("Real name", " John Smith ", 0, 1, null);
// We associate to this attribute the extended information
mmi.addAttribute("Company", "HP", 0, 2, emi1);
mmi.addAttribute("Description", "Operador de sistemas", 0, 3, null);
mmi.addAttribute("Restricted", "No", 0, 4, null);
// We set the language in the first cell of the second column
mmi.addAttribute("Language", "Castellano", 2, 0, null);
// We invoke the object that composes and shows our view.
new MenuInfoWriter(mmi, null, null).write();
</script>
```

With the previous code we create a View with the initial appearance as that of the figure below.

If the user clicks on the arrow situated to the right of the title the user's extended attributes are shown, as can be seen below.



And if we finally click on the arrow situated to the right of the "Company" attribute the extended data about the user's company is shown.

Initial information and secondary information

We are going to complicate a little the example that we are in charge with now, and apart from seeing the user's data, we are going to show as extended information the roles he belongs to.

As the definition order of the main and secondary information is unimportant, we can define any of them first.

The first thing we are going to do is to define the secondary information making use of the SecondaryMenuInfo object. Let's suppose the user belongs to three roles: Operator, Administrator and

Demo. Also, when we click over the Administrator and Operator roles we want to access the URL *getRoleData.do*, whose result will be shown in the page we are in, that is, we would jump to a completely different view, and when the user clicks on the Demo role, we want to invoke the JavaScript function called *jumpToStatus()* which will show us the user singing his companies hymn to his heart's content.

We first define the jumpToStatus() function:

```
<script>
function jumpToStatus() {
   // Code necessary to view the user singing the hymn...
   ...
}
</script>
```

Later we define the secondary information:

```
<script>
// Constructor invocation
var smi = new SecondaryMenuInfo();
// We establish the title for the secondary information
smi.addTitle("Associated Roles ");
// Setting the roles the user belongs to
smi.addAttribute("Operator", null, 0, "getRoleData.do", "_self");
smi.addAttribute("Administrator", null, 1, "getRoleData.do", "_self");
smi.addAttribute("Demo", null, 2, "jumpToStatus()", "_js");
</script>
```

Now we define the main information exactly the same as we did in the first or second examples, depending on whether we want the extended information to appear or no.

When we invoke the *MenuInfoWriter* Object we will have to indicate the presence of both the main and the secondary information.

```
<script>
// Constructor invoked
var mmi = new MainMenuInfo();
// Title establishedEstablecemos el título
mmi.addTitle("User's Data", null);
// We add the attributes starting from the leftmost column
mmi.addAttribute("Name", "operador", 0, 0, null);
mmi.addAttribute("Real name", "John Smith", 0, 1, null);
mmi.addAttribute("Company", "HP", 0, 2, null);
mmi.addAttribute("Description", "Operador de sistemas", 0, 3, null);
mmi.addAttribute("Restricted", "No", 0, 4, null);
// We set the language in the first cell of the second column
mmi.addAttribute("Language", "Castellano", 2, 0, null);
// We invoke the object that composes and shows the view.
new MenuInfoWriter(mmi, smi, null).write();
</script>
```

This code's result is shown below.

Main and secondary information with extended

We are going to complicate things further and now we are going to let the roles that are part of the secondary information to have extended information.

The first thing to do in this case is to define de extended information of the Operator, Administrator and Demo roles.

```
<script>
// We define the extended information for the Operator role
// Constructor invocation
var emi0 = new ExtMenuInfo();
// Set the title for this secondary info
emi0.addTitle("Operator Role ");
// Set the data for the Operator Role
emi0.addAttribute("Users", "2", 0, 0);
emi0.addAttribute("Performances", "5", 0, 1);
// We define the extended info for the Administration role
// Constructor invocation
var emi1 = new ExtMenuInfo();
// We set the title for this extended info
emi1.addTitle("Administrator Role");
// We set the data for the Administrator role
emi1.addAttribute("Users", "1", 0, 0);
emi1.addAttribute("Performances", "21", 0, 1);
// We set the extended information for the Demo role
// Constructor invocation
var emi2 = new ExtMenuInfo();
// We set the title for this extended information
emi2.addTitle("Demo role ");
// We set the information we need to know about this role
emi2.addAttribute("Users", "1", 0, 0);
emi2.addAttribute("Performances", "5", 2, 0);
</script>
```

Once defined we proceed like in the example number 3, but taking into account that now the roles possess extended information and we need to indicate it the code.

We first define the jumpToStatus() function:

```
<script>
function jumpToStatus() {
   // Code necessary to view the user singing the hymn...
   ...
}
</script>
```

Then we define the secondary information, associating the extended information for each role:

```
<script>
// Constructor invocation
var smi = new SecondaryMenuInfo();
// Set the title for the secondary information
smi.addTitle("Associated roles");
// Set the roles the user belongs to
smi.addAttribute("Operator", emi0, 0, "getRoleData.do", "_self");
smi.addAttribute("Administrator", emi1, 1, "getRoleData.do", "_self");
smi.addAttribute("Demo", emi2, 2, "jumpToStatus()", "_js");
</script>
```

Now we define the main information exactly the same as we did in the first or second examples, depending on whether we want the extended information to appear or no.

When we invoke the *MenuInfoWriter* Object we will have to indicate the presence of both the main and the secondary information.

```
<script>
```

```
// Construction invoked
var mmi = new MainMenuInfo();
// Setting the title
mmi.addTitle("Datos de usuario", null);
// We add the attributes starting with the leftmost column
mmi.addAttribute("Nombre", "operador", 0, 0, null);
mmi.addAttribute("Nombre real", "John Smith", 0, 1, null);
mmi.addAttribute("Compañía", "HP", 0, 2, null);
mmi.addAttribute("Descripción", "Operador de sistemas", 0, 3, null);
mmi.addAttribute("Restringido", "No", 0, 4, null);
// We put the language on the first cell of the second column
mmi.addAttribute("Lenguaje", "Castellano", 2, 0, null);
// We invoke the object that forms and composes our view.
new MenuInfoWriter(mmi, smi, null).write();
</script>
```

This code's result is shown in the figure below, where you can see that the entries for the secondary information possess a button to spread the extended information associated to them.

Main information and two secondary info

This example is an extension of the third example, very similar to it but with the existence of two secondary info instead of one.

The major difference in this case, is that apart from having to define the two secondary information, we have to establish column titles in the main information depending on the current secondary information selected. The usual in this case is that column 4 (and 5) show main information related to the associated secondary and that columns 6 (and 7) do the same for their info.

The secondary information of the roles will be identical to the one we have already seen.

First we define *jumpToStatus()*:

```
<script>
function jumpToStatus() {
   // Necessary code for the user singing the hymn...
   ...
}
</script>
```

Then secondary info is defined:

```
<script>
// Constructor invocation
var smi = new SecondaryMenuInfo();
// Title is set for the secondary info
smi.addTitle("Associated roles");
// Roles the user belong to
smi.addAttribute("Operator", null, 0, "getRoleData.do", "_self");
smi.addAttribute("Administrator", null, 1, "getRoleData.do", "_self");
smi.addAttribute("Demo", null, 2, "jumpToStatus()", "_js");
</script>
```

Now we proceed to define the secondary information. Let's say in this case we want to show the names of the applications the user has access to.

```
<script>
```

```
// Constructor invocation
var smi1 = new SecondaryMenuInfo();
// Set the title for the secondary info
smi1.addTitle("Aplications");
// We set the application the user has access to
smi1.addAttribute("GdC", null, 0, null, null);
smi1.addAttribute("Diagnostic", null, 1, null, null);
</script>
```

Now we define the main information exactly the same as we did for the first and second examples, depending on whether we want to show the secondary information.

When the object *MenuInfoWriter* is invoked we will have to indicate the presence of both the main and secondary info.

```
<script>
// Constructor invocation
var mmi = new MainMenuInfo();
// We set the title and we associate the user's extended info
mmi.addTitle("User's data", null);
// We set the title for the fourth column
mmi.addColumnTitle("Roles", 4);
// We set the title for the sixth column
mmi.addColumnTitle("Applications", 6);
// We add the attributes starting from the leftmost

mmi.addAttribute("Username", "Arlaukas", 0, 0, null);
mmi.addAttribute("Real name ", "David Phine", 0, 1, null);
// For the company attribute we associate the extended information
mmi.addAttribute("Company", "RMFC", 0, 2, null);
mmi.addAttribute("Description", "Company Description ", 0, 3, null);
mmi.addAttribute("Restricted", "No", 0, 4, null);
// We set the language in the first cell of the second column
mmi.addAttribute("Language", "Catalan", 2, 0, null);
// We invoke the object that forms and shows the view, but this time we
indicate the presence of the secondary information.
new MenuInfoWriter(mmi, smi0, smi1).write();
</script>
```

The initial result for this code is shown below.

If we now click on the "Applications" title the secondary information will be shown, the one about the applications.

21.5 Table Taglib

This taglib, designed specially for the SC, allows the generation of simple tables, that don't require ordering by columns or result pagination.

In order to use this taglib it is necessary to have defined it before in the *web.xml* (see section 4.8.3.2 for more information).

The taglibs are used assigning them a prefix such that the JSP's interpreter can recognize them each time they are found. In the case of the Table Taglib the prefix is "table", so each time the JSP's interpreter finds "<table:...>" it will know it must interpret it according to the taglib's definition. To know what it must do it is mandatory to indicate the following line at the beginning of the JSP that is going to use the taglib:

```
<%@ taglib uri="/WEB-INF/table-taglib.tld" prefix="table" %>
```

Also, the tables that are generated with the Table Taglib use the SC's style, so it is also necessary to include the "subestilosX.css" stylesheet inside the JSP (please note that the X must be substituted for the random colour of the SC).

The SC's tables possess a certain format that can have small changes, but whose final appearance is always the same. This makes all tables used in the SC to be declared initially in a way similar to this:

```
<table border="0" cellSpacing="2" cellPadding="2" width="90%">
```

As this heading is the same every single time, the best idea is to use a taglib that generates it automatically by just changing the previous line for:

```
<table:table>
```

The Table Taglib is very simple and is made up of only five tags: *table*, *header*, *row*, *cell* and *separator*. Let's see each of them together with all the possible attributes they can have and also see a few examples, which will become useful.

21.5.1 TableTag

This generates the initial code for a table. It possesses some attributes that can modify to a certain degree the basic table format of the SC. They are:

- *width*: determines the table width. Its default value is "90%". It can acquire all the values of the traditional HTML tables, allowing both percentages and pixel or point measurements. You must take into account that the tables can be deformed and if the cell width is bigger than this then the table will expand as much as necessary.
- *height*: indicates the table's height. Its default value is null and usually has no sense, because the HTML tables resize depending on the space occupied by the cells. If a value X is assigned and the content for the table needs more space then the table will expand as much as necessary even if it had been predefined to height X.
- *border*: indicates the number of pixels that the table border occupies. Its default value is 0 and is also the only value allowed for SC's tables, because in this environment the tables have no border. However, to debug the JSPs the border attribute has been allowed, so that it is possible to

check whether everything is being shown as it should, but its important to make sure the final JSP has border size 0.

- *id*: the table's identifier. By default, a table never has an id assigned so its value is "null".
- *rowsMayBeSelected*: tells whether the different rows of a table can be selected or not. Its default value is "true" and it can take the values "true" or "false". The "true" true value indicates that the rows can be selected. In this case the automatic ilumination effect of the row the mouse is over is created together with the tables. Also the row remains illuminated unless another is clicked on. If the value for this attribute is "true" all the rows must have a unique id associated to each one.
- *headerAsBody*: indicates whether the header must have the same number of columns as the rest of rows in the table. The default value is "true" and the possible values are "true" and "false". The true value indicates that the number of rows must indeed be the same.

21.5.2 Header Tag

This tag is used to declare the table header. This header can be global for the whole table or it can be local a column, so every table's column can have a different header. In the first case the number of header cells doesn't have to be the same as the number or rows in the table, whilst in the second it must necessarily be the same (see the *headerAsBody* attribute of the "table" tag).

This tag is devoid of attributes.

21.5.3 Row Tag

This tag is used to declare the begging of the new row in the table.

It possesses the following attributes:

- *width*: indicates the table width. The default value is "null", because logic indicates that a row has the same length as the table. This attribute is hardly useful.
- *height*: indicates the vertical length of the row. The default value is "null", as this value is usually resized automatically to the space needed for the table content.
- *id*: it is the row's identifier. It is necessary when the table's rows can be selected (see the attribute *rowsMayBeSelected* for the "table" tag), as it's the only way to distinguish during execution a row from another. Obviously, each row's identifier must be unique for the whole JSP.
- *onclick*: this is the action which must be invoked when an onclick event is detected on the row. Let's suppose that when a row is clicked on whose identifier is "myRow" we want to invoke a JavaScript function called "myFunction", which we have previously coded. What we will do then to declare this row is:

```
<table:row id="myRow" onclick="myFunction()">
```

Or also, supposing that when we click on the row we want to jump immediately to a certain URL, be it a JSP, a Struts Action or any other. Then the previous declaration must be:

```
<table:row id="myRow" onclick="window.location.href='URL'">
```

- *selected*: indicates whether the row must be selected from the moment the JSP is loaded for the first time. The default value is "false", which means that the row isn't selected. It can have the values "true" or "false".

21.5.4 Separator Tag

This tag introduces a row in the middle of the table that can give a new meaning to the table's rows below. The appearance of a separating row is the same one as the header's header. The effect is the same as if we had several consecutive tables, but the difference is that in this way everything forms part of the same table and we make sure that all the columns have the same width. It is a question of symmetry.

This tag has no attributes.

21.5.5 Cell Tag

This tag creates a new cell inside the table's header, inside a row or inside a separator.

The attributes this tag can have are:

- *width*: shows the cell's width. Its default value is "null", as the horizontal length for the table is usually set automatically by the browser depending on the table's needs.
- *height*: indicates the cell's vertical length. Its default value is "null".
- *id*: the cell's identifier. The cells have no default identifier, so its value is "null".
- *align*: indicates the alignment for the text inside the cell. The default value is "left".
- *colspan*: indicates the number of consecutive cells starting from this one that must be combined into one cell.
- *nobg*: indicates whether or not the background colour for this cell should be transparent. The default value is "false", in which case the cell possesses the traditional colour for the SC's cells. It can take the values "true" o "false". This attribute is hardly ever used.
- *onclick*: assigns an *onclick* event to the cell. This event doesn't usually have any meaning in cells that aren't part of the header, although it can also be used for them. Usually this event is used for the table header's cells that can be ordered by columns using struts' pagination feature.

21.5.6 Examples

Simple table with general use title

The example that follows generates a table of 500 pixel width and un-selectable rows. Also, the table will have a general use title; it won't specify the meaning of every column.

```
<table:table width="500" headerAsBody="false"
rowsMayBeSelected="false">
   <table:header>
     <table:cell>General use title</table:cell>
   </table:header>
   <table:row>
     <table:cell>a0</table:cell>
     <table:cell>a1</table:cell>
     <table:cell>a2</table:cell>
   </table:row>
   <table:row>
     <table:cell>b0</table:cell>
     <table:cell>b1</table:cell>
     <table:cell>b2</table:cell>
   </table:row>
   <table:row>
     <table:cell>c0</table:cell>
```

```
      <table:cell>c1</table:cell>
      <table:cell>c2</table:cell>
    </table:row>
    <table:row>
      <table:cell>d0</table:cell>
      <table:cell>d1</table:cell>
      <table:cell>d2</table:cell>
    </table:row>
  </table:table>
```

Table with selectable rows

This second example generates a table which will occupy 100% of the available width for the web page and where a title is set for each column. Also, the rows can be selected and when a row is selected it will remain marked with blue colour and the JavaScript *myFunction* function will be invoked, which will receive as parameter the rows identifier.

```
<script>
function myFunction(clickedRow) {
   alert(clickedRow);
}
</script>

<table:table width="100%" headerAsBody="true"
rowsMayBeSelected="true">
   <table:header>
     <table:cell>título0</table:cell>
     <table:cell>título1</table:cell>
     <table:cell>título2</table:cell>
   </table:header>
   <table:row id="row0" onclick=" myFunction(this.id)">
     <table:cell>a0</table:cell>
     <table:cell>a1</table:cell>
     <table:cell>a2</table:cell>
   </table:row>
   <table:row id="row1" onclick=" myFunction(this.id)">
     <table:cell>b0</table:cell>
     <table:cell>b1</table:cell>
     <table:cell>b2</table:cell>
   </table:row>
   <table:row id="row2" onclick=" myFunction(this.id)">
     <table:cell>c0</table:cell>
     <table:cell>c1</table:cell>
     <table:cell>c2</table:cell>
   </table:row>
   <table:row id="row3" onclick=" myFunction(this.id)">
     <table:cell>d0</table:cell>
     <table:cell>d1</table:cell>
     <table:cell>d2</table:cell>
   </table:row>
 </table:table>
```

21.6 Combotext

This taglib is a combination between a text field and a combo box. With it, any text may be typed into the text field, but there are some suggested options by default, as it happens with a combo box, which are displayed as they match the already typed text.

As with any taglib, all JSP's that use it must include the following header:

```
<%@ taglib uri = "/WEB-INF/combotext-taglib.tld " prefix = "cmbtxt" %>
```

This makes possible to use the combotext tags with the prefix *cmbtxt*.

This taglib is composed by the two tags explained in the sections below.

21.6.1 Combotext tag

Generates a combotext object.

The attributes accepted by this tag are:

- *name*: the object's name. It is a mandatory parameter. It must be a unique name inside the web page. The meaning of this attribute is the same as the *name* attribute of a common text field.
- *id*: the object's id, if any.
- *value*: the initial value for this field, if any. By default, the combotext is left empty if no initial value is specified.
- *width*: the object's width, in pixels. The default value is 140 pixels.
- *position*: the object's position. It may take only two values: *relative* and *absolute*, as it happens with any HTML element.
- *top*: the object's top position, in pixels. The default value is 0.
- *left*: the object's left position, in pixels. The default value is 0.
- *maxheight*: the maximum value for the options height, that is, the height of the displayed options shown anytime a character is typed into the combotext. The default value is 200 pixels.
- *onchange*: The javascript function to be invoked when the combotext's value is modified. Examples:

```
onchange = "myFunction()";
onchange = "myfunction('myFinalString')";
onchange = "myfunction(myVar)"; -- In this case the variable myVar must
exist.
```

21.6.2 Option tag

This tag adds an option to the combotext. Options will be displayed below the text field of this combotext anytime a character is typed, and there will only be displayed those matching with the entered text.

The attributes accepted by this tag are:

- *value*: the value and text of this option. It will be the text displayed if it matches the typed text. It is a mandatory parameter.

21.6.3 Example

The next example will create a combotext with five options.

```
<cmbtxt:combotext name="element">
  <cmbtxt:option value="users"/>
  <cmbtxt:option value="roles"/>
  <cmbtxt:option value="applications"/>
  <cmbtxt:option value="treeviews"/>
  <cmbtxt:option value="branches"/>
</cmbtxt:combotext>
```

21.7 Displaytag

This taglib generates more elaborate tables than the ones generated with the Table Taglib. It's used for tables where there is the need to paginate the results and to order them in columns. It can also be used to export the data to other formats, such as PDF, Excel, CSV or XML.

As with any taglib, all JSP's that use it must include the following header:

```
<%@ taglib uri = "http://displaytag.sf.net" prefix = "display" %>
```

It is an Open Source taglib property of *Sourceforge*, so it has not been tailor made for the SC. However, it allows us to use *decorator* classes, whose role is to provide the table the proper look for the SC, and for this the following decorators have been developed:

- *FutureGUITableDecorator*: selects a row each time and invokes a JavaScript function when the user clicks on it.
- *MultiSelectTableDecorator*: can select several rows at the same time.
- *InventoryBuilderTableDecorator*: is the decorator used in the JSPs generated by the InventoryBuilder. It should not be used for the development of applications.

The JSP used in this taglib, and the associated actions, are the only authorized to break one of the stricter rules of the SC, which is the one that forbids inserting objects in the user's session. This taglib's functioning requires the presence in the session of a collection of bean objects (it accepts several formats, such as Array, Collection, Iterator and other) from which the table's information is obtained. To avoid the clutting up of the user's session with these kinds of collections it has been decided to impose the following rule: the array or object collection must be stored in the session under the name *tmp*. This way in any session there will only be one object collection at any moment.

It will be understood that in order to get to this type of JSP a previous Struts action will have stored in the user's session the object collection under the key *tmp* with all the *beans* that the *displaytag* must display. Also, in a String *array* called *colnames* (names or titles for the columns in the table) will be indicated the names for the different *bean* attributes that we want to show, which means that the *displaytag* will invoke the *getters* for each attribute to obtain the value that will be inserted in each cell.

As this taglib's information can be consulted online (http://displaytag.sourceforge.net/11/), in this section we are going to focus on the more useful functionality for the application environment JSPs. The most important tags are therefore *table* and *column*.

21.7.1 Table Tag

This is the main taglib's tag, which can take the following attributes:

- *id*: can assign an identifier to the table.
- *style*: can set a style for the table. As the style must be the same as the one for the SC, this attribute can have value modifiers such as the table's width.
- *name*: indicates the place and name (separated with a dot. Like this: place.name) with which to find the bean collection. The place can be *sessionScope*, *requestScope* (or by default), *pageScope* and *applicationScope*. As the bean collection must be stored in the session, the value must be *sessionScope*. The name has to be *tmp*. The result will be *sessionScope.tmp*.
- *pagesize*: indicates the number of results that will be shown for each page.
- *export*: indicates whether the options for exporting the results to Excel, PDF, XML or CSV should be shown below the table. It can take the values "true" or "false".

- *sort*: indicates whether the table can be ordered by columns. It can take the values “true" or “false”.
- *requestURI*: indicates whether the URL that should be loaded every time a new page is called, the table is ordered by one of the columns or if an exporting option is selected. Usually the value is set to return to the same JSP we are in, but this doesn’t have to necessarily be so.
- *decorator*: indicates the class to be used to give the table the correct look for the SC.

21.7.2 Column tag

It is necessary to indicate a tag of this kind for every table’s column, that is, for each bean attribute we want to show.

The possible attributes are:

- *property*: indicates the bean attribute’s name whose *getter* must be invoked to get the cell’s value.
- *sortable*: indicates whether the table can be ordered depending on the values for this row.
- *titleKey*: sets the column’s title, that is, the text that must be shown in the column’s header. To internationalize it we can use the following syntax: internationalization file name followed by a dot and the key that contains the internationalized text. For example: *ApplicationResourcesUMMA.username*.
- *headerClass*: indicates the name of the style sheet class that must be assigned to this header’s cell. This class is called *tableTitle*.
- *class*: indicates the name for the stylesheet class that must be applied to this column’s cell. This class is called *tableCell*.

21.7.3 Examples

In the next example (let’s say the JSP that this code belongs to is called ejemplo.jsp and its path is precisely the one set in the attribute *requestURI* attribute) we assume the presence in the user’s session of a collection of beans stored under the key tmp. For each bean three attributes will appear: *id, name* and *description*.

```
<display:table
      id="userlist"
      style="width:98%"
      name="sessionScope.tmp"
      pagesize="20"
      export="true"
      sort="list"
      requestURI="/jsp/ej/ejemplo.jsp"
      decorator="com.hp.spain.hputils.taglib.displaytag.decorator.
FutureGUITableDecorator">
     <display:column
      property="id"
      sortable="true"
        titleKey="ApplicationResourcesUMMA:user.id"
      headerClass="tableTitle"
      class="tableCell"/>
   <display:column
      property="name"
      sortable="true"
        titleKey="ApplicationResourcesUMMA:user.name"
```

```
      headerClass="tableTitle"
      class="tableCell"/>
   <display:column
      property="description"
      sortable="true"
        titleKey="ApplicationResourcesUMMA:user.description"
      headerClass="tableTitle"
      class="tableCell"/>
</display:table>
```

21.8 FutureAlert

To invoke the *FutureAlert* from a JSP it is necessary to import the JavaScript document where the object is kept. This is done inserting it between the *<head> ... </head>* tags of the JSP the following code:

```
<script src="/activator/JavaScript/hputils/alerts.js"></script>
```

After this, the next thing we have to do is to invoke the FutureAlert's constructor. We can create initially an empty instance and establish later the title and the warning message or we can indicate them in the constructor.

The following example generates an empty instance:

```
<script>
   var fa = new FutureAlert();
</script>
```

That we can use to set the title and the message like in this example:

```
<script>
   fa.setTitle("Warning message");
   fa.setMessage("Hello, world!!!");
</script>
```

This other example creates an instance where the constructor is called specifying the title and the message:

```
<script>
 var fa = new FutureAlert("Warning for users ", "Hello, world!!!");
</script>
```

which generates an equivalent *FutureAlert* to the previous.

Let's not forget that both the title and the message can be changed at any moment by invoking as many times as is needed the *setTitle()* and *setMessage()* methods.

To show the *FutureAlert* on screen we have to invoke the *show()* method. Like this:

```
<script>
   fa.show();
</script>
```

The easiest and shortest way to set and show a *FutureAlert* with the default values is as follows:

```
<script>
   new FutureAlert("Warning for users", "Hello, world!!!").show();
</script>
```

The result for any of the previous examples is the same, as is shown below.

As can be observed, the *FutureAlert* possesses several default characteristics, and some of them can be configured.

It will automatically be shown centred in the browser's window. This property cannot be configured

The default width is of 300 pixels and the height is of 150 pixels. These dimensions can be changed at any moment by invoking the method *setBounds()*.

```
<script>
   fa.setBounds(500, 200);
</script>
```

FutureAlert is a blocking application, which means that while visible it will be impossible to click or to interact over any other element of the page. This characteristic can be changed by calling the *setBlockingAlert()* method.

```
<script>
   fa.setBlockingAlert(true);  // FutureAlert blocks
</script>
<script>
   fa.setBlockingAlert(false); // FutureAlert does not block
</script>
```

The text that appears in the *FutureAlert* button is by default "Aceptar". To establish a different text the method *setButtonText()* must be called.

```
<script>
   fa.setButtonText("OK");
</script>
```

Once visible, the *FutureAlert* will only disappear when the user clicks on the button. However, there is a *hide()* method to hide the *FutureAlert* from the code if it becomes necessary at any moment.

```
<script>
   fa.hide();
</script>
```

The *FutureAlert*'s alert versatility is superior to the JavaScript alert. Also, if in a page it is necessary to show several different *FutureAlerts* you don't have to create an instance for each of them, the same one can be used, changing the title and message as seems fit. For example, let's suppose we have shown a *FutureAlert* like the one shown below:

```
<script>
   var fa = new FutureAlert("Message for users", "Hello, world!!!");
   fa.show();
</script>
```

The user sees it and clicks on the "Accept" button, hiding the *FutureAlert*. (It is very important to take into account that the user must have already hidden the FutureAlert before changing the title or the message. If not, we take the risk of the user not having seen the initial FutureAlert.) Everything carries on as normal until the time comes to show another FutureAlert to the user. As we already had the first, instead of creating a new one we do this:

```
<script>
```

```
   fa.setTitle("Second Warning");
   fa.setMessage("Second Message!!!");
   fa.show();
</script>
```

In general, the *FutureAlert's* API is as follows:

Constructors:

- *FutureAlert()*: creates an instance with no title or message.
- *FutureAlert(String title, String message)*: creates an instance with a title and a message depending on the similar named parameters.

Methods:

- *setTitle(String title)*: sets a new title for the *FutureAlert*.
- *setMessage(String message)*: establishes a new message for the *FutureAlert*.
- *setBounds(int width, int height)*: sets a width of "width" pixels and a height of "height" pixels.
- *setBlockingAlert(boolean isBlocking)*: tells whether the *FutureAlert* will block the underlying page or not.
- *setButtonText(String buttonText)*: sets a new text to be shown inside the button that hides the FutureAlert.
- *setButtonFunction(String jsFunction)*: indicates that when the *FutureAlert's* button is clicked on the JavaScript function *jsFunction* should be called. This is a way of using the *FutureAlert* to block code execution, as the *jsFunction* won't be executed until the user clicks on the button.
- *takeUp(int numPixels)*: shows the *FutureAlert* higher up (if *numPixels* is a positive number) or further below (if negative). The vertical distance the *FutureAlert* moves depends on the *numPixels* value.
- *show()*: shows the *FutureAlert* in the centre of the browser.
- *hide()*: hides the *FutureAlert*.

21.9 FutureConfirm

To invoke the *FutureConfirm* from a jsp it is necessary to import the JavaScript document where the object is kept. This is done by inserting between the *<head> ... </head>* tags the following code:

```
<script src="/activator/JavaScript/hputils/alerts.js"></script>
```

After this, the next step is to call *FutureConfirm*'s constructor. We can create an empty instance initially and set later the title and warning message or we can indicate them in the constructor call.

The next example generates an empty instance:

```
<script>
   var fc = new FutureConfirm();
</script>
```

in which we can set the title and message in the following way:

```
<script>
   fc.setTitle("User confirmation required ");
   fc.setMessage("Do you want to say Hello, World?");
</script>
```

This other example creates an instance in which the title and message are specified in the constructor itself:

```
<script>
   var fc = new FutureConfirm("User confirmation required ", "Do you want to
say Hello, World?");
</script>
```

which generates a similar *FutureConfirm* to the previous one.

We have to note that both the message and the title can be changed at any given moment by calling as many times as needed the *setTitle()* and *setMessage()* methods.

It is also necessary to indicate the JavaScript functions that will be called when the user clicks on one of the buttons of the *FutureConfirm*. If not, the only effect will be to hide the *FutureConfirm*. These methods can be set in the constructor itself:

```
<script>
   var fc = new FutureConfirm("User confirmation required ", "Do you want to
say Hello, World?", "sayHello(true)", "sayHello(false)");
</script>
```

or the function can also be set by using the mehods *setAcceptButtonFunction() and setCancelButtonFunction()*:

```
<script>
   fc.setAcceptButtonFunction("sayHello(true)");
   fc.setCancelButtonFunction("sayHello(false)");
</script>
```

It looks obvious, but different functions can be called for each case:

```
<script>
   fc.setAcceptButtonFunction("sayHello()");
   fc.setCancelButtonFunction("sayGoodbye()");
</script>
```

and strings can also be set as parameters for the functions called:

```
<script>
   fc.setAcceptButtonFunction("say(\"Hello!!\")");
   fc.setCancelButtonFunction("say(\"Goodbye!!\")");
</script>
```

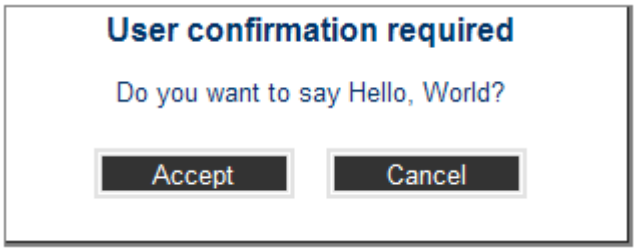
In order to show the *FutureConfirm* on screen we call the method called *show()*. Like this:

```
<script>
   fc.show();
</script>
```

The shortest way to set and show a *FutureConfirm* with default parameters is like this:

```
<script>
   new FutureConfirm("User confirmation required", "Do you want to say
Hello, World?").show();
</script>
```

The result of any of the previous examples is the same, as is shown below



As can be seen, the *FutureConfirm* possesses certain properties by default, some of them being configurable.

By default it will always be shown centred in the browser's window. This property is not configurable.

The default width is of 300 pixels and the height is of 150 pixels. These dimensions can be changed at any given moment by calling the *setBounds()* method.

```
<script>
   fc.setBounds(500, 200);
</script>
```

The *FutureConfirm* is blocking, which means that while visible it will be impossible to click or interact with the underlying window. This characteristic can be changed by calling the method called *setBlockingConfirm()*.

```
<script>
   fc.setBlockingConfirm(true);  // The FutureConfirm is blocking
</script>
<script>
   fc.setBlockingConfirm(false); // The FutureConfirm is not blocking
</script>
```

The text that appears inside the *FutureConfirm's* buttons is by default "Aceptar" and "Cancelar". To set a different text it is necessary to invoke the methods *setAcceptButtonText()* and *setCancelButtonText()*.

```
<script>
   fc.setAcceptButtonText("Yes");
   fc.setCancelButtonText("No");
</script>
```

Once it appears, the *FutureConfirm* will only disappear when the user clicks on the button. However, there exists a method called *hide()* that can hide the *FutureConfirm* from the code if it becomes necessary.

```
<script>
   fc.hide();
</script>
```

The *FutureConfirm's* versatility is superior to JavaScript's confirm. Also, if in a page it is necessary to show several different *FutureConfirms* we don't have to create a new one for each, we can use the same one and change the title and message depending as we see fit.

For example, let's suppose we have shown a *FutureConfirm* like the one below:

```
<script>
   var fc = new FutureConfirm("User confirmation required ", "Do you want to
say Hello, World?");
   fc.setAcceptButtonFunction("sayHello()");
   fc.setCancelButtonFunction("sayGoodbye()");
   fc.show();
</script>
```

The user sees it and clicks on the "Aceptar" button, hiding the *FutureConfirm*. (It is important to note that the user must have hidden the *FutureConfirm* before changing the title or message. If not, we take the risk that the user doesn't notice the initial *FutureConfirm*.) Everything carries on as normal until the time comes

to show the user the second *FutureConfirm*. As we already had the first, instead of creating a new one we do the following:

```
<script>
   fc.setTitle("Second confirm");
   fc.setMessage("Second confirm message");
   fc.setAcceptButtonFunction("say(\"Hello!!\")");
   fc.setCancelButtonFunction("say(\"Goodbye!!\")");
   fc.show();
</script>
```

In general, the *FutureConfirm's* API is as follows:

Constructors:

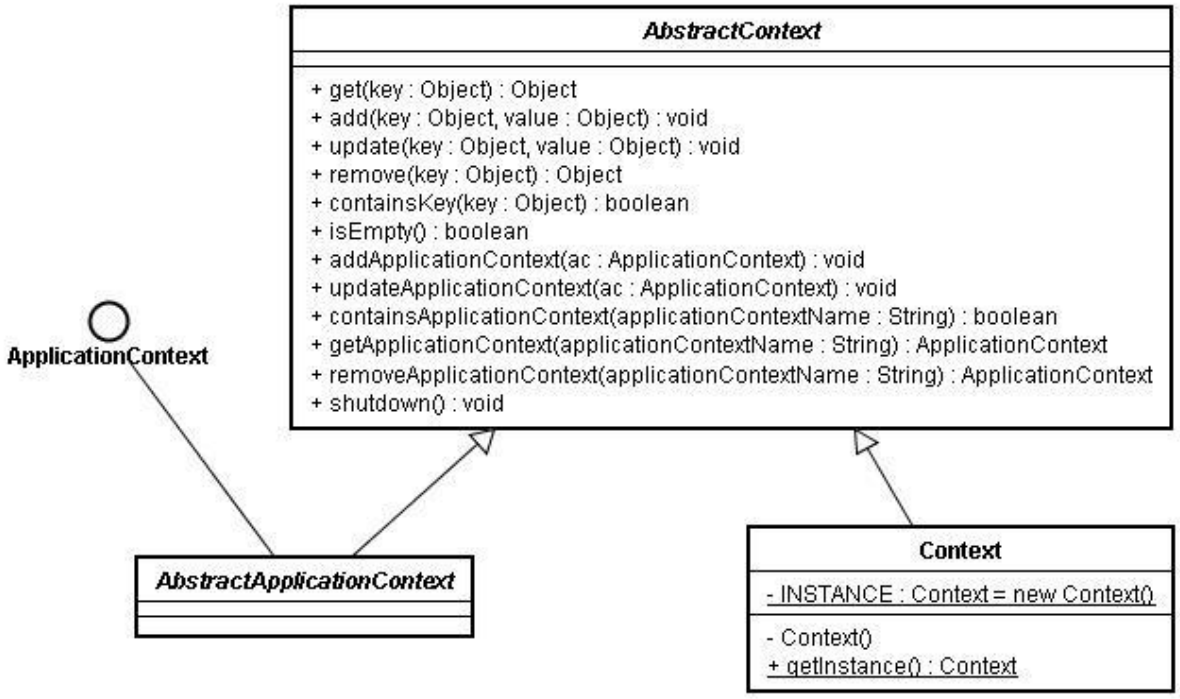
- *FutureConfirm()*: creates an instance with no title and no message.
- *FutureConfirm(String title, String message)*: creates an instance with title and message depending on the similar named parameters.
- *FutureConfirm(String title, String message, String acceptFunctionName, String cancelFunctionName)*: creates an instance with title and message, and the JavaScript functions "acceptFunctionName" and "cancelFunctionName" will be called when the user clicks on the respective buttons.

Methods:

- *setTitle(String title)*: sets the new title for the *FutureConfirm*.
- *setMessage(String message)*: sets the new message for the *FutureConfirm*.
- *setBounds(int width, int height)*: sets the width and height in pixels.
- *setBlockingConfirm(boolean isBlocking)*: indicates whether or not the *FutureConfirm* will block the underlying page.
- *setAcceptButtonText(String buttonText)*: sets the new text that will be shown inside the first button to hide *FutureConfirm*.
- *setCancelButtonText(String buttonText)*: new text that will be shown inside the second button to hide *FutureConfirm*.
- *setAcceptButtonFunction(String fnc)*: indicates that the JavaScript function called "fnc" should be called when *FutureConfirm*'s first button is clicked on.
- *setCancelButtonFunction(String fnc)*: indicates that the JavaScript function called "fnc" should be called when *FutureConfirm*'s second button is clicked on.
- *takeUp(int numPixels)*: makes the *FutureConfirm* appear higher up (if numPixels is a positive number) or further down (if negative). The vertical distance that the FutureConfirm will move coinsides with the value for numPixels.
- *show()*: makes the *FutureConfirm* appear centred on the browser.
- *hide()*: hides the *FutureConfirm*.

21.10 SC's Context and Application Context

The SC provides a static singleton class called Context where any application can store key-value pairs and *Application Context* instances. The figure below shows the UML representation of the classes involved.



21.10.1 Context class

This is a static singleton class, what means that it can be never instantiated by any other class but itself. The only one existing instance is pointed by the INSTANCE static constant, and can be obtained to operate over it using the getInstance() method.

This class implements the abstract class *AbstractContext*.

21.10.2 AbstractContext class

This abstract class manages the key-value pairs and the *ApplicationContext* instances. Provides methods to get, add, remove and update any key-value pair or any *ApplicationContext*.

21.10.3 ApplicationContext interface

This interface defines a getName() method used to identify the ApplicationContext instance.

21.10.4 AbstractApplicationContext class

This abstract class implements the *ApplicationContext* interface and defines an abstract method called *getName* as it is specified in the interface. It also extends the abstract class *AbstractContext* to inherit the methods defined there.

Any application which needs an Application Context has to define a new class which must extend this one. Once the Application Context is instantiated, it can be stored into the *Context* using the *addApplicationContext()* method.

21.11 Properties files

Some applications running under SC may need to be configured before the HPSA is started up. This can be done using properties files that must be located at this directory:

```
C:\hp\jboss\server\diagnostic\deploy\hpovact.sar\activator.war\properties
```

Along the stating up process all these properties files under the specified directory are read and stored into the SC's Context object as a key-value pair, where the key is a String with the name of the properties file (without the *.properties* extension) and the value is a *java.util.Properties* object representing the contents of the file.

This way, any application can get any configured parameter looking for the *java.util.Properties* object at the SC's Context and getting the parameter from it.

For instance, let's suppose that an application deploys a *xxx.properties* file into the specified directory. The contents of this file are:

```
equipment.ip = 11.22.33.44
equipment.port = 8080
```

When the SC starts up, a new entry is added to the Context under the key *xxx*.

Afterwards, any Struts' action of the application can get the two parameters configured into the file easily. The next code shows how:

```
java.util.Properties p =
   (java.util.Properties) Context.getInstance().get("xxx");
String ip = p.get("equipment.ip");
String port = p.get("equipment.port");
```

See the API of the Context class for further information.

21.12 Action Audit

The URL of the RMI service with the methods for action audition is stored in the SC's Context (see section 14.1 for further information). The key needed to obtain the URL from the Context is a constant defined in the com.hp.spain.futuregui.login.LoginConstants interface.

There is an example about this in the section 9 dedicated to *Action Audit*.

The parameters of the *auditAction* method provided with the RMI service are:

- *messageType*: indicates what kind of message is being audited (error, warning, info, etc.)
- *username*: the name of the user who generates the log.
- *workId*: the identifier of the task which audited this action.
- *sourceComponent*: the component where this actions was being performed.
- *actionPerformed*: the name of the action that was being performed.
- *description*: a brief description of the audit message.

21.13 WFLT

21.13.1 WFLTAction.do

This is the action which should be invoked to launch and track a workflow. The way the workflow will be launched and how it will be tracked can be specified through some configuration parameters which are

described in this section. The information necessary for the launching of a workflow will be searched in the request attributes and in the parameters. There is only one restriction, it is that all this elements must be Strings, or adaptable to Strings. All the elements will be searched in lower case too.

21.13.1.1 General parameters

These are the fundamental parameters used to launch a workflow:

- __wfname: Workflow name. Is a mandatory parameter. If it's not present an error will be thrown.
- __wfmwfmname: The name of the Mwfm in which the workflow will be launched or in which the workflow will be searched. If it's not present the default Mwfm will be used.

It's also possible to track a workflow that has been already launched. In order to use this functionality we need to specify a new parameter:

- __wfJobId: The id of the workflow which we are going to track.

21.13.1.2 Concurrent Workflows

To enable the tracking of workflows with children using the Concurrent Workflow Module the next parameter has to be used.

- __wfConcurrentCheck: Has a boolean value. This parameter is not mandatory. If its value is true the workflows will be tracked by the Concurrent Workflows application.

21.13.1.3 Database tracking

It's also possible to track workflows with children using the database. To use this functionality is necessary to use the next three parameters:

- __wfServiceName: It is the workflow's service name. Its value should be the bean package referencing the database table (Ex: com.hp.spain.inventory.Service).
- __wfServicePk: It is the workflow's service primary key. That's the primary key which will be associated with the workflow in the database.
- __wfDatasource: It is the data source name to access the database where we will store the workflow jobId.

21.13.1.4 ECP Command tracking

The activations launched by workflows can also be tracked. When this option is enabled the commands sent to the ECP will be shown in the screen. Some parameters are necessary to access to this functionality:

- __wf_command_audit_active: This parameter will enable the ECP command tracking if its value is "true". It's not mandatory and by default this option is not enabled.
- __wf_command_id. This id must be unique and will be used to filter the received messages and show only the ones related to a specific activation. At the same time, this identifier must be provided to the ECP under the same key. If no id is provided the jobId value will be taken by default.

21.13.1.5 SOSA

The workflow launcher tracker can launch SOSA 3 service orders and track them. In order to use the SOSA integration some parameters are needed.

- __wfsosatype: It corresponds to the field "service_order_name" from the table "catalog_service_order".
- __wfsosaservice: It corresponds to the field "service_name" from the table "catalog_service_order".
- __wfsosaaction: It corresponds to the field "service_operation" from the table "catalog_service_order".
- __wfsosacheck: This parameter must be true to indicate that SOSA is being used.

There are also specific SOSA parameters that are needed in the workflow's case packet. More details about them and about how to launch workflows in SOSA can be found in the document "OVSA SPI for Service Providers - SOSA - Developer Reference.doc".

21.13.1.6 Miscellaneous parameters

- next_url: It's the URL which will be invoked when the workflow finish its execution. The URL can be absolute (http://...) or relative to the base activator path (Ex. /activator/jsp/future-gui/blanck.jsp).

21.13.1.7 User parameters

The user parameters are the attributes and parameters retrieved from the request that start with the prefix "wfvar__". The next parameter:

```
wfvar__equipmentname=NT300
```

In the workflow's case packet it will be translated into this:

```
Name: equipmentname
Value: NT300
```

It is possible to make groups of attributes or parameters using String arrays. Example: if we need to launch a workflow that waits for a String array with three values whose name is "equipmentnames" we will need to use the next four parameters:

```
wfvar__arrayiterator0=wfvar__equipmentnames
wfvar__equipmentnames5=NT300
wfvar__equipmentnames22=NT400
wfvar__equipmentnames17=NT6000
```

The first one is the group's name while the others, formed using the name of the array followed by any group of numbers or chars, will contain the names which will constitute the array.

This will make the next line in the workflow's initial case packet:

```
Name: equipmentnames
Value: {NT300, NT400, NT6000}
```

If the workflow needs more arrays it will need to repeat the process adding to the first parameter's integer value (wfvar__arrayiterator0, wfvar__arrayiterator1, wfvar__arrayiterator2...). The enumeration must be consecutive.

Example:

This example is going to launch a workflow from the inventory, called EQUIPMENT_CONFIGURATION, which will receive three String arrays: the first one containing the equipments' names, the next one containing their IPs and the third containing their operating systems. Also, it will need the user name to access them. We'll assume that the user name is the same for the three of them.

```
/activator/WFLTAction.do?
__wfname=EQUIPMENT_CONFIGURATION&
__wfDatasource=confDS&
__wfservicename=confservice&
__wfservicepk=25&
__wfmwfmname=localmwfm&
wfvar__username=admin&
wfvar__arrayiterator0=equipmentnames&
wfvar__equipmentnamesA=NT300&
wfvar__equipmentnamesB5=NT400&
wfvar__equipmentnames20=NT6000&
wfvar__equipmentnamesAB=NT50&
wfvar__arrayiterator1=equipmentips&
wfvar__equipmentipsA=10.10.10.1&
wfvar__equipmentipsB=10.10.20.2&
wfvar__equipmentipsC=10.10.30.3&
wfvar__equipmentipsD=10.10.40.4&
wfvar__arrayiterator2=equipmentsos&
wfvar__equipmentsosA=HPUX&
wfvar__equipmentsosB5=HPUX&
wfvar__equipmentsos20=Windows&
wfvar__equipmentsosAB=Solaris
```

Glossary

Datasource: a factory for connections to the physical data source.

EJB (Enterprise JavaBeans): a server-side component that encapsulates the business logic of an application. The EJB specification intends to provide a standard way to implement the back-end 'business' code typically found in enterprise applications.

JSP (Java Server Page): a technology which provides a simplified, fast way to create dynamic web content. JSP technology enables rapid development of web-based applications that are server- and platform-independent.

MWFM (Micro Workflow Manager): the workflows engine provided with the HPSA.

MWFM Módule: a class which extends those provided by the MWFM to perform a certain functionality for the HPSA. Every module is started up by the MWFM and runs in the same Java virtual machine.

Servlet: a technology which provides web developers with a simple, consistent mechanism for extending the functionality of a web server and for accessing existing business systems. A servlet can almost be thought of as an applet that runs on the server side–without a face.

Taglib: a librarie which allows you to create custom actions and encapsulate functionality. Custom tags can clearly separate the presentation layer from the business logic. They are easy to maintain reusable components that have acces