Jahia CMS AND Portal Version 5.0

Developing JSR 168 portlets

with Jahia CMS and Portal Version 5.0

English v1.0

jahia

Jahia Ltd
Swizerland
9, route des jeunes
1227 Carouge

TERMS AND CONDITIONS

LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

6. GOVERNING LAW.
Any action derived from or related to this agreement will be governed by the Swiss Law and shall be of the competence of the judicial authorities of the Canton of Vaud, Switzerland.

7. TERMINATION.
This agreement is effective until terminated.

END OF TERMS AND CONDITION

PREFACE

Jahia is an enterprise-level Internet portal and content management system (CMS and Enterprise Portal). Written entirely in Java J2EE, Jahia was designed to run on Tomcat servers, but can be implemented on other application servers like IBM WebSphere.

One of Jahia's primary features is its ability to integrate the Web applications called *portlets*. This guide explains how to develop a JSR 168 portlet for JAHIA.

Chapter 1: Introduction to Portlet Development. This chapter covers definitions and important concepts for developing Jahia portlets.

Chapter 2: Developing a Basic Portlet. This chapter covers a simple portlet example called *HelloWorld!*

Chapter 3: Developing an Advanced Portlet. This chapter covers an advanced portlet example.

Chapter 4: Portlets, Frameworks and Bridges. This chapter covers various J2EE frameworks that can be used to develop portlets.

Chapter 5: Converting JahiaWebApp to Portlets. This chapter covers converting a JahiaWebApp (Jahia 4 portlet) to a JSR 168 portlet (Jahia 5 portlet).

Chapter 6: Useful Links. This chapter includes some useful links to the JSR 168 community.

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are used in this document:

Italic formatting is used for:
• comments

`Fixed width formatting` is used for:
• JAVA, HTML, and JSP code examples.

Tips and Tricks *Notes* *Warnings*

This document has been written with the goal of being as accurate as possible. However, it may not cover new features available in the most recent versions of JAHIA.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION TO PORTLET DEVELOPMENT

This chapter covers various definitions and concepts for developing Jahia portlets.

1 DEFINITIONS

1.1 CMS AND PORTAL

A content management system (CMS) is a computer tool dedicated to publishing and organizing texts, information, reports, and multimedia items in an information system. A CMS provides the following services:

- Shared processing of content by different users in different places (collaboration)
- Implementation of content management processes (workflow)
- Restriction of content to designated users (security)
- Revision management (statistics, version management)
- Determining when to publish content (planning)
- Presentation of content in the right format (templates)
- Publishing content on different sites (syndication)
- Display customization based on the visitor (personalization)
- Broadcasting content on different media (multimedia)

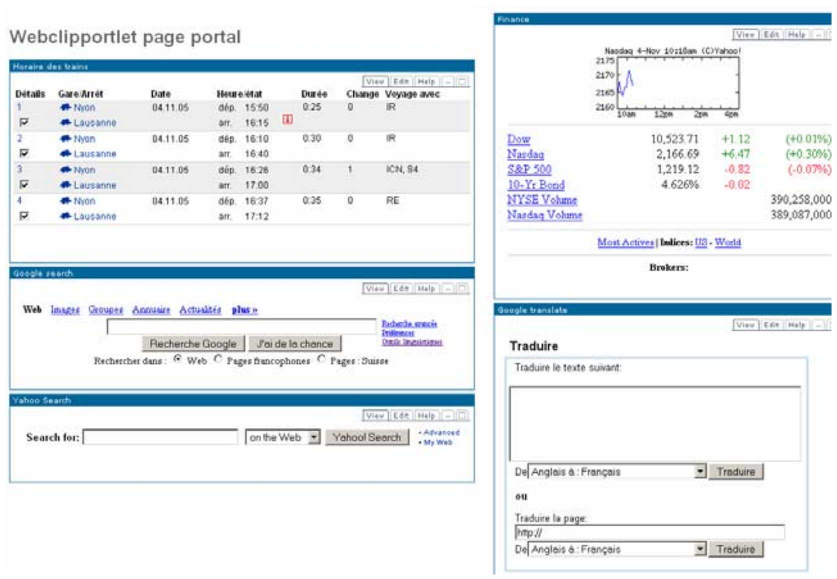
A portal is an exclusively Internet-based CMS with the ability to integrate content from external sources into a distributed environment (databases, integration of XML and XLS, etc.). In addition, portals generally provide end users with simple search and publishing features on the Web. Portals can be used by entire companies (e.g., Jahia Solutions), or by company departments such as Marketing, Human Resources, or Sales. Portals can also be used by individual users in the form of customized interactive dashboards.

Note

Jahia is an enterprise-level Internet portal and content management system (CMS and Enterprise Portal).

1.1.1 PORTLET

Portlets are reusable components integrated into a corporate portal. They enable a user to have centralized and user-friendly access to different resources such as data, applications, and websites from the same window, and to modify the portal interface for a personalized working environment. Portlets simplify access to information that is of interest to a group of users. From a technical standpoint, portlets are the applications that generate active and dynamic parts of the HTML aggregated by the portal.



Détails	Gare/Arrêt	Date	Heure/état		Durée	Change	Voyage avec
1	Nyon	04.11.05	dép.	15:50	0:25	0	IR
✓	Lausanne		arr.	16:15			
2	Nyon	04.11.05	dép.	16:10	0:30	0	IR
✓	Lausanne		arr.	16:40			
3	Nyon	04.11.05	dép.	16:26	0:34	1	ICN, S4
✓	Lausanne		arr.	17:00			
4	Nyon	04.11.05	dép.	16:37	0:35	0	RE
✓	Lausanne		arr.	17:12			

Rechercher dans : [✓ Web] Pages francophones Pages : Suisse

Dow	10,523.71	+1.12	(+0.01%)
Nasdaq	2,166.69	+6.47	(+0.30%)
S&P 500	1,219.12	-0.82	(-0.07%)
10-Yr Bond	4.626%	-0.02	
NYSE Volume			390,258,000
Nasdaq Volume			389,087,000

Most Actives | Indices: US - World

Brokers:

Illustration 1: Sample JAHIA pages with portlets

To enable interoperability among different portals, the international Java community (Java Community Process or JCP) specified the behavior of a portlet for all of its content areas: aggregation, personalization, presentation, and security. In August 2003, this work gave rise to the JSR (Java specification request) 168 specification, which defines the portlet API and sets a standard that all Java portals in the world must comply with whether they are commercial or not. This specification is briefly covered in the following chapter.

The entire text of the JSR 168 specification is available at:
http://jcp.org/aboutJava/communityprocess/review/jsr168/

Jahia 5 integrates the open source Jetspeed 2 portal and is compatible with the JSR 168 specification.

For more information on Jetspeed 2, go to:
http://portals.apache.org/jetspeed-2/

Jahia 5 also supports JahiaWebApp portlets. These are native to Jahia and compatible with Jahia 4. For more information on JahiaWebApp portlets, see the PDF at:
http://www.jahia.org/jahia/webdav/site/jahia_org/shared/documentation/WebAppGuideDRAFT1.pdf.

In order to comply fully with the portlet specification, future versions of Jahia will no longer support JahiaWebApps portlets. For this reason, we recommend developing JSR 168 portlets instead of JahiaWebApp portlets for JAHIA 5, and converting your JahiaWebApps portlets to JSR 168 portlets when migrating from Jahia 4 to Jahia 5.

1.1.2 PREREQUISITES

This guide assumes that you are familiar with:

- The Java language
- The Servlet specification, especially HTTP servlets
- Apache Tomcat
- HTML and XML
- JavaScript, JSP, and Taglib
- The Jahia content template

2 JSR 168 SPECIFICATION AND API

2.1 PORTAL

A (Jahia) portal is a Web application that generally ensures the functions of personalization, centralized administration (single sign-on), aggregation of content from different sources, and management of information systems' presentation layer. Aggregation is defined as the action of integrating content from different sources on the same Web page. A portal can include a set of portlets in order to create content for different users.

2.2 Portlet

Portlets are reusable Web components that are controlled by a portlet container and that process requests and produce dynamic content. Used by portals as components of the user interface, portlets provide a presentation layer for information systems.

Content produced by a portlet is called a *fragment*. A fragment is a piece of coding (e.g., HTML, XHTML, WML) that is governed by specific rules and can be aggregated with fragments of other portlets to create a complete document. The content produced by a portlet can change from one user to another depending on user preferences (see section "2.5 Concepts" below).

Technically, portlets are `war` files that are deployed as servlets. Similar to a servlet, a portlet is defined for the application server using the `web.xml` servlet deployment descriptor. In addition to the servlet descriptor, portlets must also provide a portlet deployment descriptor (`portlet.xml`) to define the portlet functions for the portal server. The diagram that follows illustrates the structure of a *war* file representing a portlet.

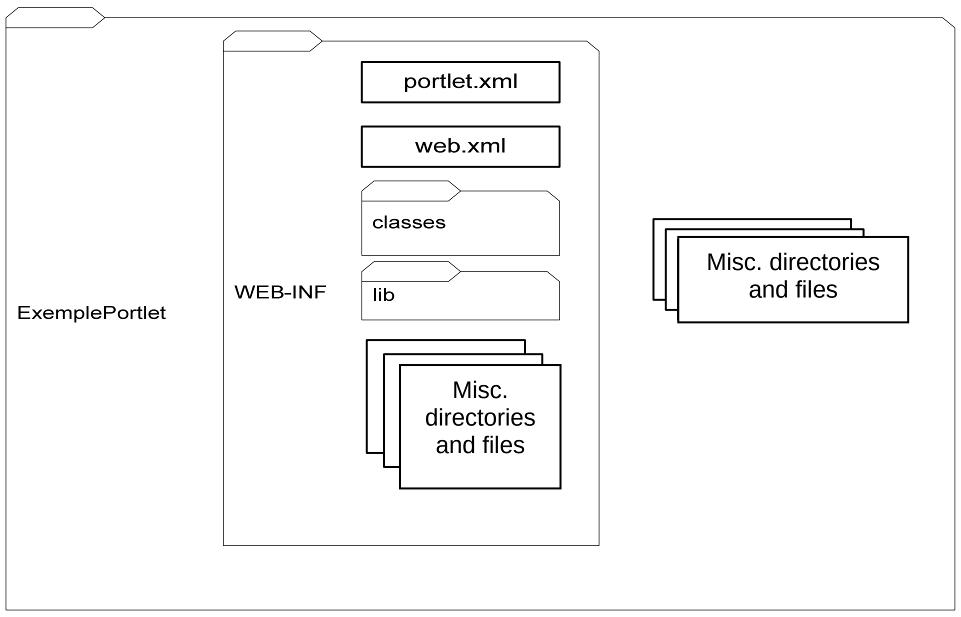


Illustration 2: Standard content of a *war* file representing a portlet

Information defined in the `portlet.xml` file includes specific configuration parameters like the type of coding processed by the portlet, initialization parameter values, and so forth. This information enables the portal server to provide services to the portlet. For example, if a portlet is set to process **Help** and **Edit** modes (see section "2.5 Concepts" below) in the portlet deployment descriptor, the portal server displays icons enabling the user to call the portlet help and modification pages.

The following is an example of the `portlet.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app>
  <portlet>
    <description>query_portlet</description>
    <portlet-name>query_portlet</portlet-name>
    <display-name>query_portlet</display-name>
    <portlet-class> sql.portlet.QueryPortlet</portlet-class>
    <init-param>
      <name>view_url</name> </value>
    </init-param>
    <init-param>
      <name>help_url</name>
      <value>/templates/help.jsp</value>
    </init-param>
    <expiration-cache>-1</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <supported-locale>en</supported-locale>
    <portlet-info>
      <title>Query Portlet</title>
      <short-title>Query Portlet</short-title>
    </portlet-info>
    <portlet-preferences>
      <preferences-validator>
           sql.portlet.QueryPreferencesValidator
      </preferences-validator>
    </portlet-preferences>
  </portlet>
</portlet-app>
```

On the user side, a portlet is a window on a portal site where the site provides a service or specific information such as a planner or news.

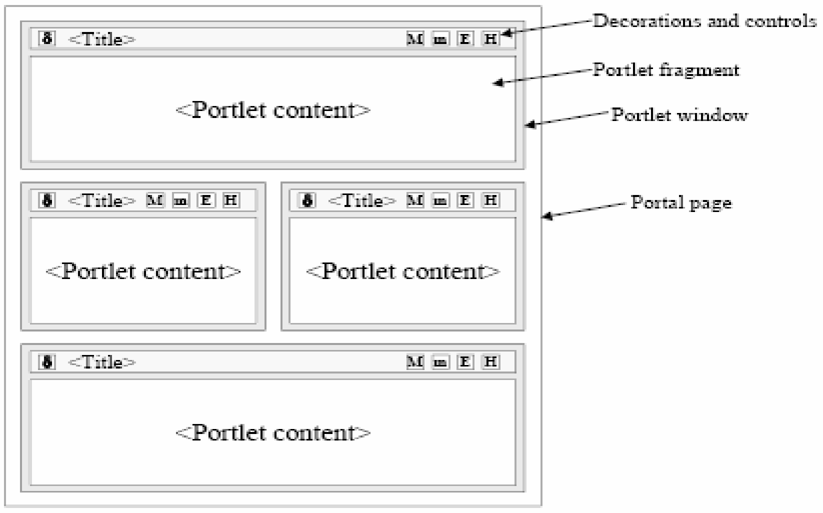


Illustration 3: Portal page with portlets

From the perspective of application development, portlets are extensions designed to run in a portlet container belonging to a portal server.

Note

Jahia 5 uses Jetspeed as its portlet container.

2.3 Portlet Container

The portlet container (Jetspeed) provides an execution environment in which portlets are instantiated, used, and ultimately destroyed. Portlets depend on the portal infrastructure to access user profile attributes, participate in action and window events, communicate with other portlets, access remote content, consult authorizations, and store persistent data. The portlet API provides standard interfaces for these functions.

However, the container is not responsible for aggregating content generated by portlets. This is managed by the portal.

2.4 Generation of the Portal Page

Portlets run in a container. Containers receive content generated by portlets and send it to the portal server. Finally, the portal server builds the portal page and sends it to the client (e.g., to the browser).

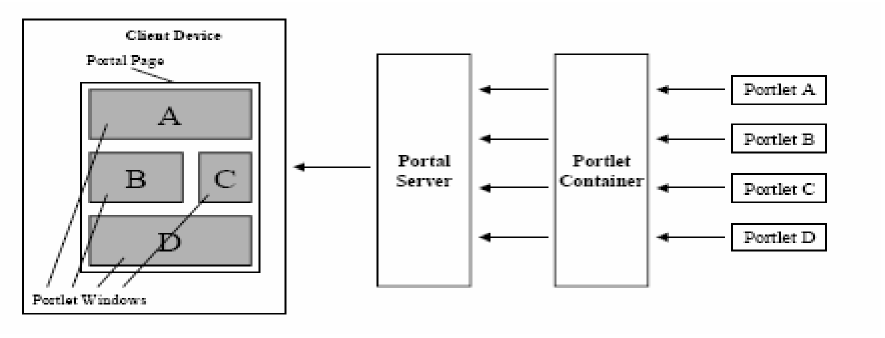


Illustration 4 : Basic portal architecture

2.5 Concepts

2.5.1 Lifecycle

A portlet lifecycle is very similar to a servlet lifecycle. It consists of three stages:

- Initialization
- Request processing
- Destruction

The portlet receives requests based on user interaction with the portlet or the portal page. Request processing consists of two stages:

- Action request processing (`processAction(…)`): An action is requested when a user clicks on a link generated by the portlet. Action request processing terminates before the portlet begins render request processing.
- Render request processing (`renderAction(…)`): During this step, the portlet generates the content sent to the portal page. All portlets on the portal page can execute this step in parallel.

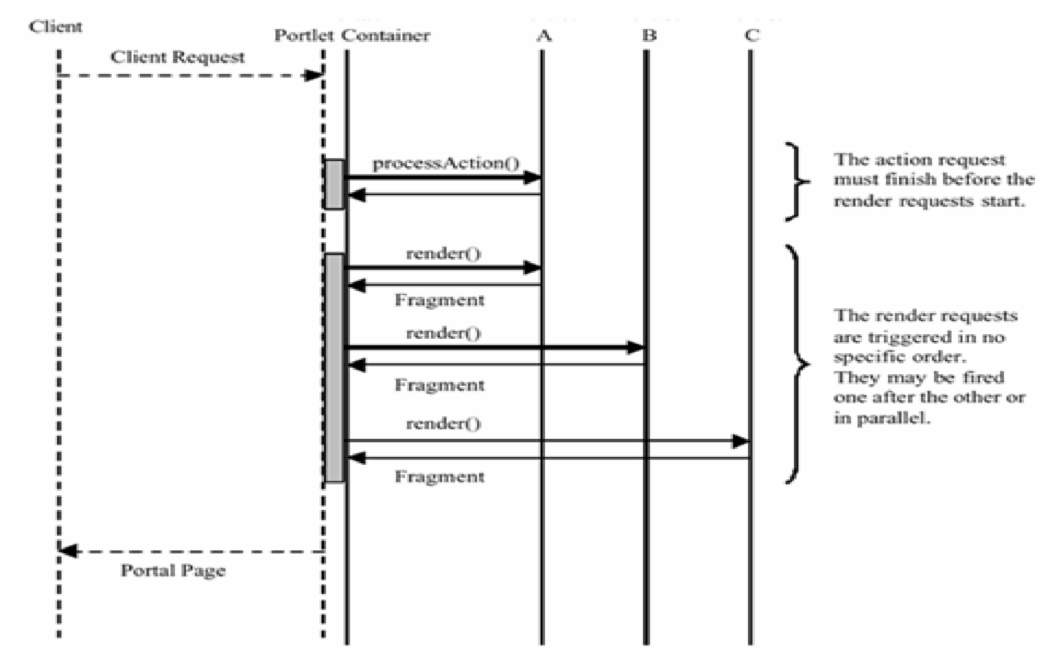

Client
Portlet Container
A
B
C
Client Request
processAction()
render()
Fragment
render()
Fragment
render()
Fragment
Portal Page
The action request must finish before the render requests start.
The render requests are triggered in no specific order. They may be fired one after the other or in parallel.

Illustration 5: Portlet request processing

A portlet is a class that must implement the `javax.portlet.Portlet` interface and that provides four methods (`init(…)`, `processAction(…)`, `render(…)`, and `destroy(…)`) for proper management by the container.

For more information on the javax.portlet.Portlet interface, go to: http://portals.apache.org/pluto/multiproject/portlet-api/apidocs/javax/portlet/Portlet.html

The javax.portlet.GenericPortlet class provides an implementation for the Portlet interface. The render(…) method consists of three methods: doView(…), doEdit(…), and doHelp(…). We recommended subclassing this class to develop a portlet. For more information, go to: http://portals.apache.org/pluto/multiproject/portlet-api/apidocs/javax/portlet/GenericPortlet.html

2.5.2 Session Management

In the same way that `HttpSession` objects are used by servlets, portlets define the `PortletSession` object, which enables objects to be saved during the client session. Two ranges are defined when saving an object in a session:

- `PORTLET_SESSION:` the object can be accessed by the portlet and the current user
- `APPLICATION_SESSION`: the object can be accessed by the whole application (portlets, servlet, and JPSs) and only by the current user

2.5.3 PortletURL

Contrary to servlets, a portlet is not directly linked to a URL. The JSR 168 specification defined the concept of PortletURL, which enables the generation of URLs pointing to the portlet (through the portal entry/exit point).

There are two URL types:

- `actionsURL`: the `processAction(…)` and `renderAction(…)` methods are executed in this order
- `renderURL`: only the render method is executed

2.5.4 Security

2.5.4.1 Authentication

The portal (Jahia) manages authentication.

2.5.4.2 Authorization

Portlets are based on the roles template defined by the `J2EE/Servlet` specification. The role of a user who is logged in can be verified using the `isUserInRole()` method. The user name of a user can be acquired using the `getUserPrincipal()` and `getRemoteUser()` methods.

2.5.5 Portlet modes

Through different modes, a portlet can display different user interfaces based on the task it will perform. A portlet has three display modes that are defined in the portlet deployment descriptor. The portlet container stores the portlet mode in the `PortletMode` object. The following modes are provided by the portlet API:

- **View**: A portlet appears in **View** mode when it is initially created on a portal page for a user. This is the default portlet display mode.
- **Edit**: When this mode is processed by the portlet, users can personalize the portlet based on their needs. For example, a portlet can provide a page that enables users to specify their location so they can get information on local weather or events. Users must be logged into the portal to access **Edit** mode.
- **Help**: When this mode is processed by the portlet, a help page with more information on the portlet is provided to users.

The portlet API provides several methods to the portlet to determine the current mode. All portlets must process **View** mode. The portal includes commands that enable users to change the current mode. The following is an example of a portlet in **View** mode:

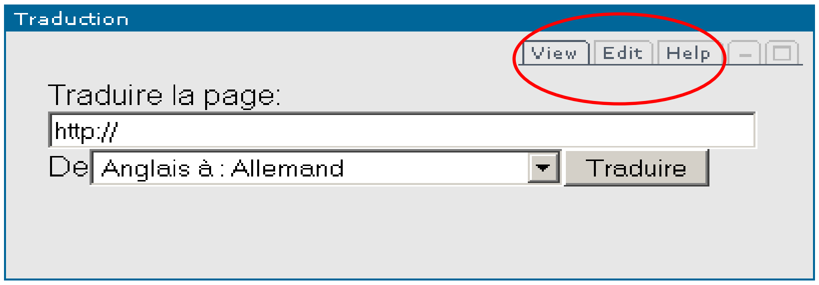


Illustration 6: Portlet mode

This bar displays the ***View***, ***Edit***, and ***Help*** modes. The portlet switches to **Edit** mode when the user clicks on the Edit tab.

The API provides the *javax.portlet.PortletMode* PortletMode class to manage the portlet mode.

For more information on the javax.portlet.PortletMode class, go to:
http://portals.apache.org/pluto/multiproject/portlet-api/apidocs/javax/portlet/PortletMode.html

It is only possible to save preferences during the processAction (…) stage.

2.5.6 Portlet states

The different states of a portlet enable users to change how the portlet window is displayed on the portal page. From a browser, users access these different states using tabs on the title bar just like in Windows applications. Portlet states are stored in the `PortletWindowState` object as a Boolean value. The portlet API provides the following states:

- **Normal**: When it is initially created on the portal page, the portlet appears in Normal mode and is organized on the page with the other portlets.
- **Minimized**: When the portlet is minimized, only its title bar is displayed on the portal page.

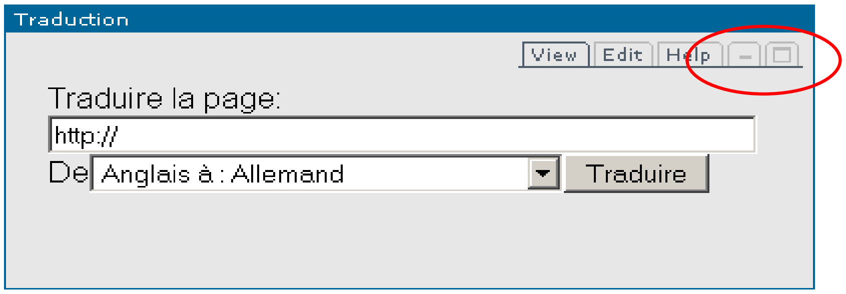


Illustration 7: Portlet state

The portlet API provides several methods to the portlet to determine the current state.

For more information on the javax.portlet.WindowState class, go to: [*http://portals.apache.org/pluto/multiproject/portlet-api/apidocs/javax/portlet/WindowState.html*](http://portals.apache.org/pluto/multiproject/portlet-api/apidocs/javax/portlet/WindowState.html)

2.5.7 Portlet Preferences

The portlet can store permanent data for a specific user with the `PortletPreferences` object. Preferences can use default values, which are defined in the deployment descriptor (*portlet.xml* file).

*The mode recommended for defining preferences is **Edit** mode. This mode provides the user with a personalization screen.*

The container can use validators to ensure that the values of saved preferences meet certain criteria.

2.5.8 Restrictions

Portlets generate aggregated HTML fragments on a portal page. As a result, the JSR 168 specification prohibits a portlet from generating `Base`, `body`, `frame`, `frameset`, `head`, `html`, and `title` tags.

3 Jahia and Portlet JSR 168

3.1 Request Attributes

Jahia inserts attributes into the request object that can be very interesting to applications developers. Note that the use of these attributes will make the application Jahia specific, whereas most of the other interceptors are not. The table below describes the name, type, possible values of the attributes set by Jahia and that may be accessed by web applications through the renderRequest.getAttribute(String attributeName) Portlet API call:

Attribute name	Type	Value	Description
"org.portletapi.portal"	String	“true”	Always set to value “true” in Jahia. This is the recommended way to determine whether it is running in standalone or within a Jahia Portal
"org.portletapi.userlist"	Vector of String objects	List of user names	Contains a list of user names that correspond to all the users that are in any role of the application. Useful to present a list of users for application-specific user assignments. Be warned though that it usually is a better idea to use roles for this but it may not always be possible.
"org.portletapi.contextid"	String	== fieldID	The contextid is a String that is useful to have an identifier to separate different instances of the same application. Jahia allows the same application to be instantiated multiple times on the same page. The contextID will be an identifier that will change among these instances, so that instance specific data may be stored in the application. In the current implementations of Jahia, the contextID is equivalent to the fieldID that contains the application. It’s also possible to get a unique String identifier thanks to getNamespace() method on a renderResponse object (JSR168 specific).
"org.portletapi.fullscreen"	String	“true” / “false”	Not yet fully implemented. This attribute tells the application whether it is running in full screen mode or not. Basically if this attribute is true, the application may take over the whole screen, including HTML headers, etc.
"org.jahia.sitekey"	String	jParams.getSiteKey ()	The unique String identifier for the current site.
"org.jahia.siteurl"	String	jParams.getSiteURL ()	A String object that contains the siteURL parameter entered that usually corresponds to the domain name for the site, i.e.: www.jahia.org, www.jahia.com, etc....
"org.jahia.siteid"	String	Integer.toString (jParams.getSiteID ())	A String object that contains an integer value that corresponds to the database identifier for the site.
"org.jahia.pageid"	String	Integer.toString (jParams.getPageID ())	A String object that contains an integer value for the page number that the application is being displayed on.
"org.jahia.operationmode"	String	jParams.getOperation-Mode ()	The current operation mode of the page. Possible values for this attribute in Jahia 4/5 are: “normal” (also called Live in our templates), “edit” (content edition mode), “debug” (not used, but still accepted in page URLs), “preview” (preview modifications to a page before it gets published in live mode), “compare” (a special mode to compare two versions of the content on the page)

3.2 SESSION ATTRIBUTES SET BY TEMPLATES

By default, Jahia does not share it's session attributes with application session attributes. This means that if you set an attribute in Jahia's session (through a scriptlet or tag in a template) using a call similar to this:

```
request.getSession().setAttribute("name", "value");
```

You will not be able to access it from your application through the following call:

```
request.getSession().getAttribute("name");
```

This limitation exists for security reasons, as application should be as separates as possible in order to avoid namespace conflicts, as well as attribute corruption, etc. But it is possible to desactivate this limitation by using a special setting in the jahia.properties configuration file.

```
webapps.dispatcher.inheritJahiaSessionAttributes = true
```

The default setting is false, and settings this parameter to true will authorize web applications to access Jahia's session attributes.

CHAPTER 2: DEVELOPING A BASIC PORTLET

This chapter covers an example of developing and deploying a portlet with Jahia 5 using a basic template.

1 INTRODUCTION

The aim of this portlet is to display the message "Hello world!" The development consists of the following steps:

- Creating the *HelloWorldPortlet* class that implements the *javax.portlet.Portlet* interface
- Creating the *portlet.xml* and *web.xml* files
- Archiving the portlet in a `war` file
- Deploying the portlet
- Displaying it in a Jahia page

Developing a portlet requires the portlet API library, which is available at: http://jcp.org/aboutJava/communityprocess/final/jsr168/index.html.

2 *HELLOWORLDPORTLET* CLASS

As with any portlet, the class must implement the *javax.portlet.Portlet* interface. The portlet API provides a class called *GenericPortlet*, which implements this interface. This class breaks down the *render(…)* method into three methods:

- `doView(…)`: portlet in **View** mode
- `doHelp(…)`: portlet in **Help** mode
- `doEdit(…)`: portlet in **Edit** mode

The example portlet works only in **View** mode. It uses the `doView(…)` method.

```
public void doView(RenderRequest req, RenderResponse resp) {
      try {
            response.setContentType("text/html");
            PrintWriter out = resp.getWriter();
            out.print("Hello world");
      }
      catch (Exception ex) {
            ex.printStackTrace();
      }
}
```

The `doView(…)` method is similar to the `doGet(…)` and `doPost(…)` servlet methods. The `PrintWriter` object enables writing to the portal output stream. This is then aggregated with text from other portlets and the Jahia template to form a complete Jahia page.

The following is the complete portlet code:

```
package org.jahia.portlet;

import javax.portlet.*;
import java.io.PrintWriter;
import java.io.*;
import java.lang.StringBuffer;
/**
 *  HelloWorldPortlet
 *
 *@author    Khaled TLILI
 */
public class HelloWorldPortlet extends GenericPortlet {

      /**
       *  Init portlet method
       *
       *@param  config                 Description of Parameter
       *@exception  PortletException  Description of Exception
       */
      public void init(PortletConfig config) throws PortletException {
            super.init(config);
      }
      /**
```

```
         *  processAction method
         */
        public void processAction(ActionRequest request, ActionResponse response) {
                System.out.println("====== Begin process method ======");
        }
        /**
         *  render method
         */
        public void doView(RenderRequest request, RenderResponse response) {
                try {
                        response.setContentType("text/html");
                        PrintWriter out = response.getWriter();
                        PortletURL url = response.createRenderURL();
                        PortletURL actionURL = response.createActionURL();
                        out.print("Hello world!");
                }
                catch (Exception ex) {
                        ex.printStackTrace();
                }
        }

        /**
         *  destroy method
         */
        public void destroy() {
                super.destroy();
        }
}
```

To compile this portlet, add the portlet.jar library to the classpath.

*To add the **Edit** and **Help** modes, implement the* `doEdit(…)` *and* `doHelp(…)` *methods in that order and update the portlet.xml files.*

3 THE *PORTLET.XML* FILE

This file serves to describe the portlets that will be deployed in Jahia in the form of a Web application.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app version="1.0">
  <portlet>
        <description>Hello World portlet</description>
        <portlet-name>HelloWorldPortlet</portlet-name>
        <portlet-class>org.jahia.portlet.HelloWorld</portlet-class>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
        </supports>
        <portlet-info>
            <title>SimplePortlet</title>
        </portlet-info>
    </portlet>
</portlet-app>
```

The descriptor contains various information on the portlet like its name, its implementation class, the modes it supports (in our example, **View** mode only), and information describing it.

The `portlet.xml` file is located on the same level as the *web.xml* file, that is, in the WEB-INF directory.

*An application can declare several portlets. This enables having only one *.war file, or grouping similar portlets (functions, back end, etc.).*

4 THE *WEB.XML* FILE

In our example, the `web.xml` file is empty. Jahia (via Jetspeed) adds information to it after deployment.

After deployment, the content of the `web.xml` file is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <display-name>HelloWorldPortlet</display-name>
  <description>Hello World Portlet </description>
  <servlet>
    <servlet-name>JetspeedContainer</servlet-name>
    <display-name>Jetspeed Container</display-name>
    <description>MVC Servlet for Jetspeed Portlet Applications</description>
    <servlet-class>org.apache.jetspeed.container.JetspeedContainerServlet</servlet-class>
    <init-param>
      <param-name>contextName</param-name>
      <param-value> HelloWorldPortlet</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JetspeedContainer</servlet-name>
    <url-pattern>/container/*</url-pattern>
  </servlet-mapping>
  <taglib>
    <taglib-uri>http://java.sun.com/portlet</taglib-uri>
    <taglib-location>/WEB-INF/tld/portlet.tld</taglib-location>
  </taglib>
</web-app>
```

Note

If the web.xml file already contained information, it would be aggregated with the content above.

5 DEPLOYMENT

Portlets have the same structure as classic Web applications. The following tree structure must be respected:

```
+ HelloWorldPortlet
+ META-INF
+ WEB-INF
      + classes
            +…
      + lib
      porlet.xml
      web.xml
```

After it has been created, the *HelloWorld.war* file must be placed in the following directory: `${TOMCAT_HOME}\webapps\jahia\WEB-INF\var\new_webapps.`

Jahia automatically deploys the portlet. The automatic deployment performs following actions:

- Moving the `HelloWorld.war` file to:
  `${TOMCAT_HOME}\webapps\jahia\WEB-INF \etc\jetspeed\deploy`
- Adding an entry to the database
- Rewriting the `web.xml` file
- Deploying the application in Tomcat

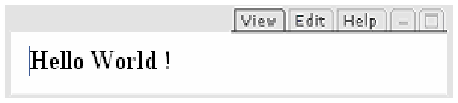
Note *The war file can be generated with the Java jar command or the Ant or Maven tools.*

Warning *The redeployment procedure is the same. However, make sure that no files are locked (jar, properties file, HSQL database file, etc.).*

Warning *NEVER PLACE THE *.WAR FILE DIRECTLY IN* `${TOMCAT_HOME}\webapps` *AS JAHIA NEEDS TO REWRITE THE* `WEB.XML` *FILE.*

6 VIEWING

In order to view the portlet, it needs to be added to a Jahia page (see "*JahiaEndUserGuide.pdf*").



Hello World !

CHAPTER 3: DEVELOPING AN ADVANCED PORTLET

This chapter covers developing an advanced portlet. It looks at how to use JSPs, attributes, preferences, and modes.

1 INTRODUCTION

The aim of this portlet is to display the message:

```
Hello <user name>!
Your favorite film is <film title>.
```

The portlet must enable the logged-in user to specify the title of his or her favorite film.

This portlet will use three modes:

- **View**: to display the above message
- **Edit**: to change preferences
- **Help**: to display a help message

Tip

*We recommend not allowing preference values to be updated by the user except in **Edit** mode.*

The development consists of the following steps:

- Creating the `HelloPortlet` class that implements the *javax.portlet.Portlet* interface
- Creating the `portlet.xml` and `web.xml` files
- Archiving the portlet in a `*.war` file
- Deploying the portlet
- Displaying it in a Jahia page

2 *HELLOPORTLET* CLASS

In contrast to the previous example, this example will use only JSPs to generate the HTML fragments. The portlet API makes the `PortletRequestDispatcher` object available to developers, which enables the generation of HTML output to be delegated to a JSP file. A JSP file is employed in the same way as the `RequestDispatcher` object of the servlet specification.

2.1 VIEW MODE

In this mode, the portlet retrieves the value of the <film> preference, saves the value in the session, and dispatches it to a JSP. The code of the `doView(…)` method is as follows:

```
public void doView(RenderRequest req, RenderResponse resp) {
        try {
                response.setContentType("text/html");

                // Get preference value
                PortletPreferences pp = req.getPreferences() ;
                String film = pp.getValue("film","valeur par défaut" );

                // Store in session
                PortletSession ss = req.getPortletSession();
                ss.setAttribute(“film”,film,PortletSession.APPLICATION_SCOPE);

                // Dispatcher
                PortletRequestDispatcher d =
                          getPortletContext().getRequestDispatcher(“/jsp/view.jsp”);
                d.include(req,resp);
        } catch(Exception e){
                // Manage exception
}
}
```

The *view.jsp* file is as follows:

```
<%@ taglib uri=“http://java.sun.com/portlet“ prefix=“p“%>
<p:defineObjects/>
<div>
Hello <%=renderRequest.getRemoteUser()%> !
</div>
<br/>
<div>
Votre film préféré est <%=renderRequest.getPortletSession().getAttribute(“film“)%>.
</div>
```

Warning

Do not forget the first two lines of the JSP file:

```
<%@ taglib uri=“http://java.sun.com/portlet“ prefix=“p“%>
<p:defineObjects/>
```

This taglib argument enables the instantiation of the renderRequest and renderResponse objects. Their use is similar to the request and response objects of the servlet specification.

2.2 EDIT MODE

Edit mode enables the portlet to store preferences. The `doEdit(…)` method dispatches only to a JSP file.

```
public void doEdit(RenderRequest req, RenderResponse resp) {
      try {
            resp.setContentType("text/html");
            PortletRequestDispatcher d =
                      getPortletContext().getRequestDispatcher(“/jsp/edit.jsp”);
            d.include(req,resp);
      } catch(Exception e){
            // Manage exception
}
}
```

The *edit.jsp* file is a form that enables a value to be entered for the film title. The *edit.jsp* file is as follows:

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="p"%>
<p:defineObjects/>
<form action="<p:actionURL/>">
<div>
Title of favorite film: <input type="text" name="film"/>
</div>
<br/>
<div>
<input type="submit"/>
</div>
</form>
```

The example used the <p:actionURL/> tag to generate an ACTION type portletURL. Thus, the processAction(…) method is executed.

Do not forget the first two lines of the JSP file:

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="p"%>
<p:defineObjects/>
```

This taglib argument enables the instantiation of the renderRequest and renderResponse objects. Their use is similar to the request and response objects of the servlet specification.

In addition to the `doEdit(…)` method, the `processAction(…)` method is executed. In our application, its main role is to save the preference value. The portlet API provides a group of methods that enable preferences to be saved.

```
public void processAction(ActionRequest req, ActionResponse resp) {
      try {
            // Identify mode
            PortletMode pm = req.getPortletMode();

            // Update preferences. Only for EDIT mode.
            if(pm.equals(PortletMode.EDIT)){
                // Retrieve value
```

```
                  String film = request.getParameter(“film”);

                  // Retrieve preference object
                  PortletPreferences pp = req.getPreferences();

                  // Update value
                  pp.setValue(“film”,film);

                  // Store permanently
                  pp.store();
              }
       } catch(Exception e){
              // Manage exception
}
}
```

It is possible to force ***View*** *mode after saving by adding resp.setPortletMode(PortletMode.VIEW) to the end of the processAction method.*

2.3 HELP MODE

In this mode, the portlet displays a message. The `doHelp(…)` method dispatches to the *help.jsp* JSP. The related code is as follows:

```
public void doHelp(RenderRequest req, RenderResponse resp) {
       try {
              resp.setContentType("text/html");
              PortletRequestDispatcher d =
                        getPortletContext().getRequestDispatcher(“/jsp/help.jsp”);
              d.include(req,resp);
       } catch(Exception e){
              // Manage exception
}
}
```

The *help.jsp* file is as follows:

```
<div>
The portlet enables the logged-in user to save the title of his or her favorite film.
</div>
```

Contrary to JSPs intended for servlets, this JSP generates only an HTML fragment. As a result, it does not contain the tags <html>, <head>, and <body>, among others.

3 THE *PORTLET.XML* FILE

This file serves to describe the portlets that will be deployed in Jahia in the form of a Web application.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app version="1.0">
  <portlet>
        <description>Hello portlet</description>
        <portlet-name>HellodPortlet</portlet-name>
        <portlet-class>org.jahia.portlet.HelloPortlet</portlet-class>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
             <portlet-mode>edit</portlet-mode>
            <portlet-mode>help</portlet-mode>
        </supports>
        <portlet-info>
            <title>HelloPortlet</title>
        </portlet-info>
        <portlet-preferences>
            <preference>
               <name>HelloPortlet</name>
               <value>HelloPortlet</value>
            </ preference >
        </portlet- preferences >
    </portlet>
```

```
</portlet-app>
```

4 What's Next?

See sections 3, 4, and 5 of "Chapter 2: Developing a Basic Portlet."

CHAPTER 4: PORTLETS, FRAMEWORKS AND BRIDGES

This chapter covers various J2EE frameworks that can be used to develop portlets.

1 INTRODUCTION

The majority of Web applications are developed using frameworks like Struts or JSF. Bridges were developed to integrate these technologies. Primarily, they enable converting an application (e.g., a servlet) into a portlet without modifying the Java code.

Most of bridges are developped by the open source community and may contains some bugs. Before using a specific bridge, make sur that it is stable (by checking mailing list and forums,...).

2 STRUTS FRAMEWORK

This bridge enables a Struts application to be converted into a JSR 168 portlet rather quickly. Simply create the *portlet.xml* file and a bridge configuration file called *struts-portlet.xml*. The *struts-portlet.xml* file enables the generated URL type, *actionURL* or *renderURL*, to be specified to the framework.

In addition, the JSPs have to be modified so that the URLs generated are PortletURLs.

For more information on using the Struts bridge, go to: http://portals.apache.org/bridges/multiproject/portals-bridges-struts/index.html

3 JSF FRAMEWORK

This bridge enables a JSF application to be converted into portlet. Simply create the *portlet.xml* file and add the *jsf-bridge* library. The `org.apache.portals.bridges.jsf.FacesPortlet` class is configured to implement the portlet instead of the `GenericPortlet` class doing this.

We recommend using the JSF framework to develop JSR 168 portlets because it was specifically designed with portlets in mind.

For more information on using the JSF bridge, go to:
http://portals.apache.org/bridges/multiproject/portals-bridges-jsf/index.html

4 OTHER BRIDGES: PHP, PERL, ETC.

The JSR 168 community is working hard on providing solutions for developers to integrate existing applications based on languages other than Java. Most proposed solutions are based on bridges. The main difficulty is the bridge configuration.

For more information on using these bridges, go to:
http://portals.apache.org/bridges/release-notes.html

CHAPTER 5: CONVERTING JAHIAWEBAPPS TO PORTLETS

This chapter covers various methods for converting JahiaWebApps to portlets.

1 INTRODUCTION

Jahia 4 supports its own portlet format called *JahiaWebApp*. Now that the JSR 168 specification exists, we recommend converting your JahiaWebApps to portlets. This section covers standard methods for simplifying the migration:

- Using the *jahiaWebAppPortlet* bridge
- Using the different framework bridges
- Converting the Java code
- Using a jahia.xml file (not recommended: the portlet will not be a JSR 168 portlet)

2 *JAHIAWEBAPPPORTLET* BRIDGE

The *JahiaWebAppPortlet* bridge enables the conversion of a JahiaWebApp without modifying the Java code or JSPs. This involves the following steps:

- Adding the `portlet.xml` file
- Modifying the `web.xml` file

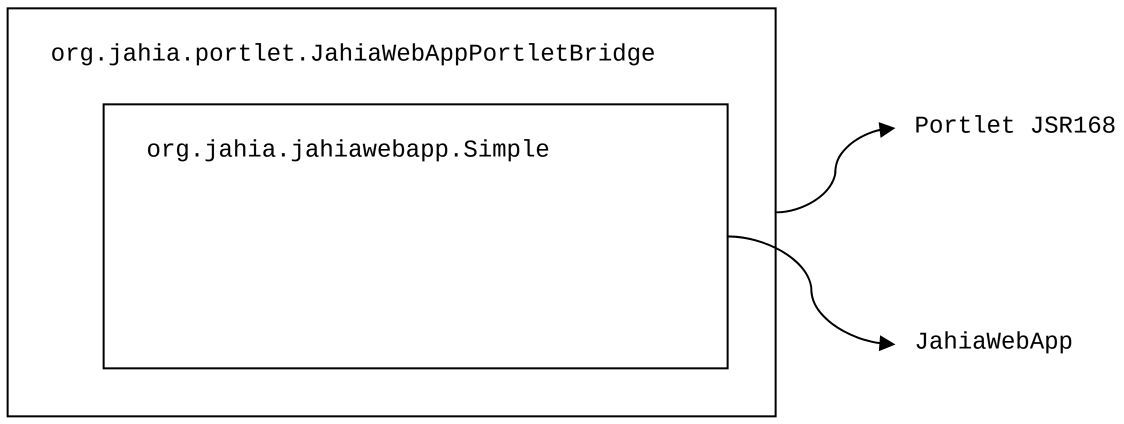
Warning

Do not forget to add the jahiaWebAppPortlet.jar library to WEB-INF/lib.

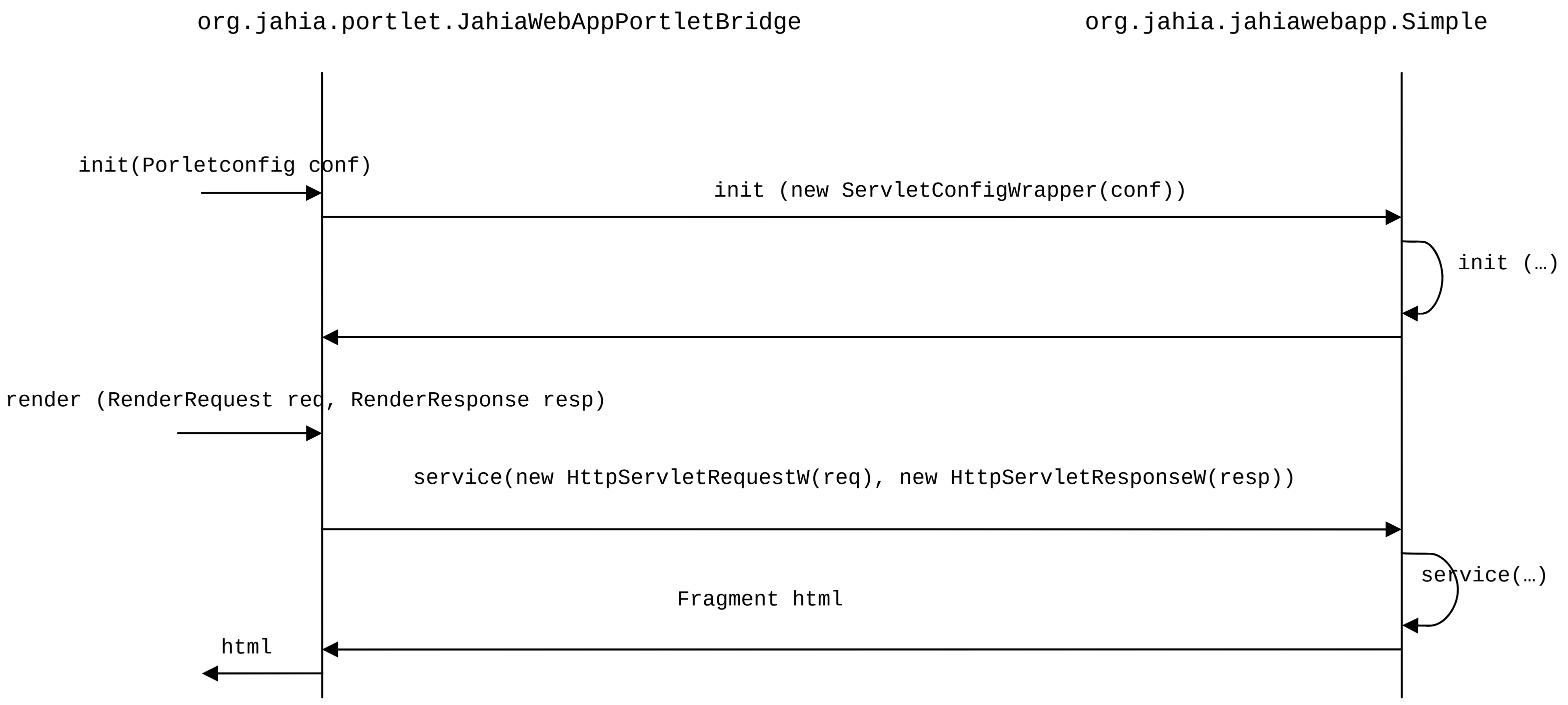
2.1 HOW DOES THE BRIDGE WORK?

The *JahiaWebAppPortlet* bridge consists of a set of wrappers that wrap the main portlet objects such as `RenderRequest`, `RenderResponse`, `PortletContext`, and `PortletSession` in `HttpServletRequest`, `HttpServletResponse`, `ServletContext`, and `HttpSession` objects.

The main class is `org.jahia.portlet.JahiaWebAppPortletBridge` of the type `javax.portlet.Portlet`, and it enables wrapping a JahiaWebApp in a JSR 168 portlet.


org.jahia.portlet.JahiaWebAppPortletBridge
org.jahia.jahiawebapp.Simple
Portlet JSR168
JahiaWebApp

The bridge works according to the following diagram:


org.jahia.portlet.JahiaWebAppPortletBridge
org.jahia.jahiawebapp.Simple
init(Porletconfig conf)
init (new ServletConfigWrapper(conf))
init (…)
render (RenderRequest req, RenderResponse resp)
service(new HttpServletRequestW(req), new HttpServletResponseW(resp))
service(…)
Fragment html
html

At initialization, the container calls the `init(…)` method of the `org.jahia.portlet.JahiaWebAppPortletBridge` portlet with `PortletConfig` as the parameter. The bridge wraps the `PorletConfig` object in a `ServletConfig` object and calls the JahiaWebApp `init(…)` method.

Each time the container calls the `render(…)` method, the bridge wraps the `RenderRequest` and `RenderResponse` objects in `HttpServletRequest` and `HttpServletResponse` objects, then calls the JahiaWebApp service method.

The bridge generates only actionUrl type URLs. The sendRedirect(…) method is simulated using JavaScript.

2.2 How is it used?

2.2.1 Migration example

The following sections cover a JahiaWebApp example whose *web.xml* file is the following:

```
<web-app>
    <servlet>
       <servlet-name>
          AdressBook
       </servlet-name>
       <servlet-class>
         org.jahia.webapps.addressbook.AdressBook
       </servlet-class>
       <load-on-startup>1</load-on-startup>
       <init-param>
           <param-name>name</param-name>
          <param-value>value</param-value>
       </init-param>
    </servlet>
    <servlet>
       <servlet-name>
          InitServlet
       </servlet-name>
       <servlet-class>
         org.jahia.webapps.InitServlet
       </servlet-class>
             <load-on-startup>1</load-on-startup>
    </servlet>
</web-app>
```

2.2.2 ADDING THE *PORTLET.XML* FILE

First, starting from the *web.xml* file, determine the servlet (`entryPoint`) that represents the JahiaWebApp. In our example, `entryPoint` corresponds to the servlet called `AdressBook`. Once `entryPoint` has been identified, the rest comes down to creating the `portlet.xml` file according to these rules:

- `<servlet-name>` → `<portlet-name>`
- `<servlet-class>` → the `jahiaWebAppClass` parameter value
- `<init-param><param-name><param-value>` → `<init-param><name>< value>`

The value of `<portlet-class>` is `org.jahia.portlet.JahiaWebAppPortletBridge`.

Warning

Servlets observe the following:

```
 <init-param>
        <param-name>name</param-name>
        <param-value>value</param-value>
</init-param>
```

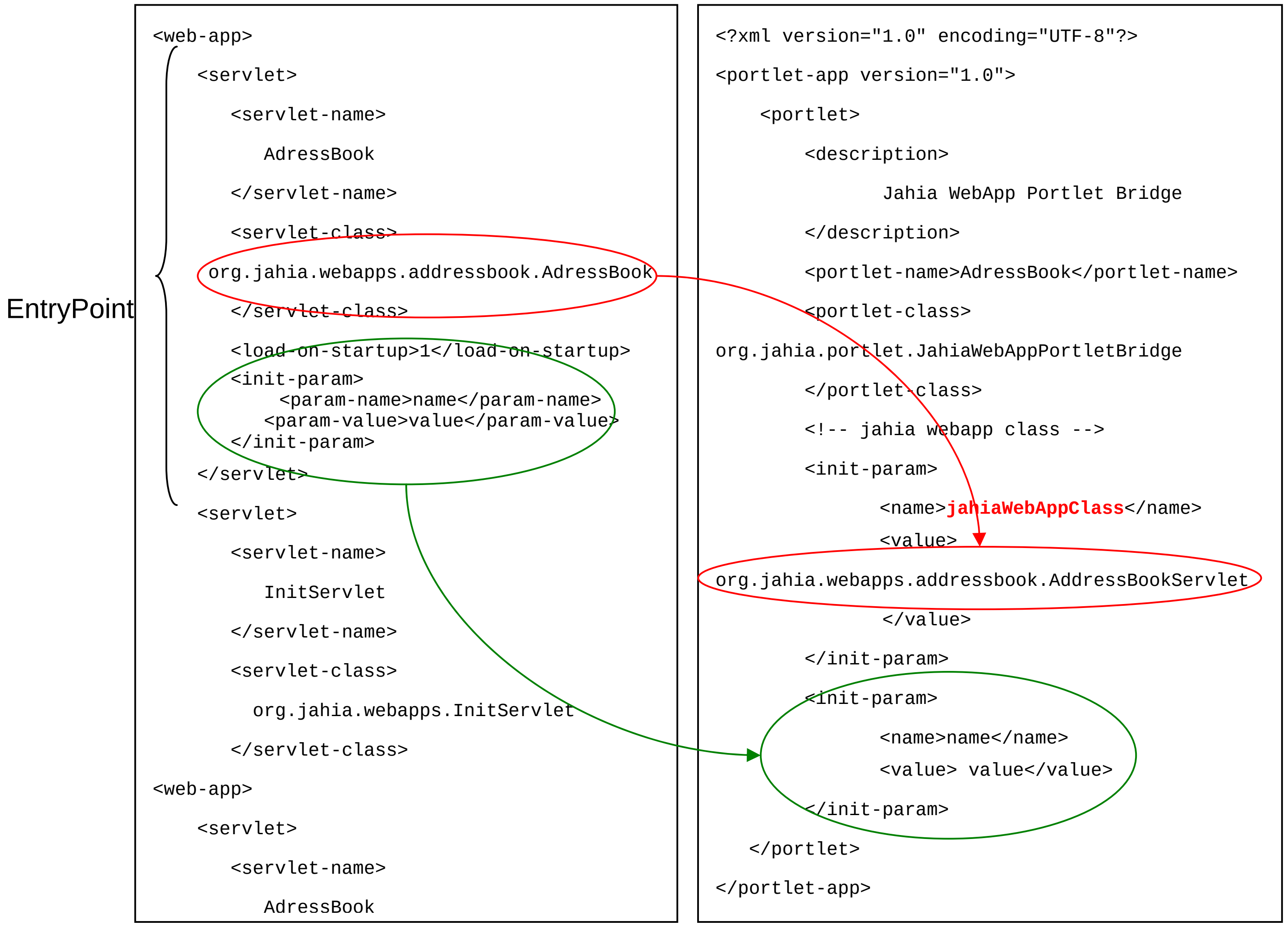
Whereas portlets observe the following:

```
 <init-param>
        <name>name</name>
        <value>value</value>
</init-param>
```

The basic *portlet.xml* file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app version="1.0">
    <portlet>
        <description>Jahia WebApp Portlet Bridge</description>
        <portlet-name>…</portlet-name>
        <portlet-class>org.jahia.portlet.JahiaWebAppPortletBridge</portlet-class>
        <!-- jahia webapp class -->
        <init-param>
            <name>jahiaWebAppClass</name>
            <value>…</value>
        </init-param>
       <init-param>
              <name>name</name>
              <value>value</value>
      </init-param>
    </portlet>
</portlet-app>
```

For the preceding example, the *portlet.xml* file is as follows:



```
<web-app>
    <servlet>
       <servlet-name>
          AdressBook
       </servlet-name>
       <servlet-class>
     org.jahia.webapps.addressbook.AdressBook
       </servlet-class>
       <load-on-startup>1</load-on-startup>
       <init-param>
           <param-name>name</param-name>
          <param-value>value</param-value>
       </init-param>
    </servlet>
    <servlet>
       <servlet-name>
          InitServlet
       </servlet-name>
       <servlet-class>
         org.jahia.webapps.InitServlet
       </servlet-class>
<web-app>
    <servlet>
       <servlet-name>
          AdressBook
```

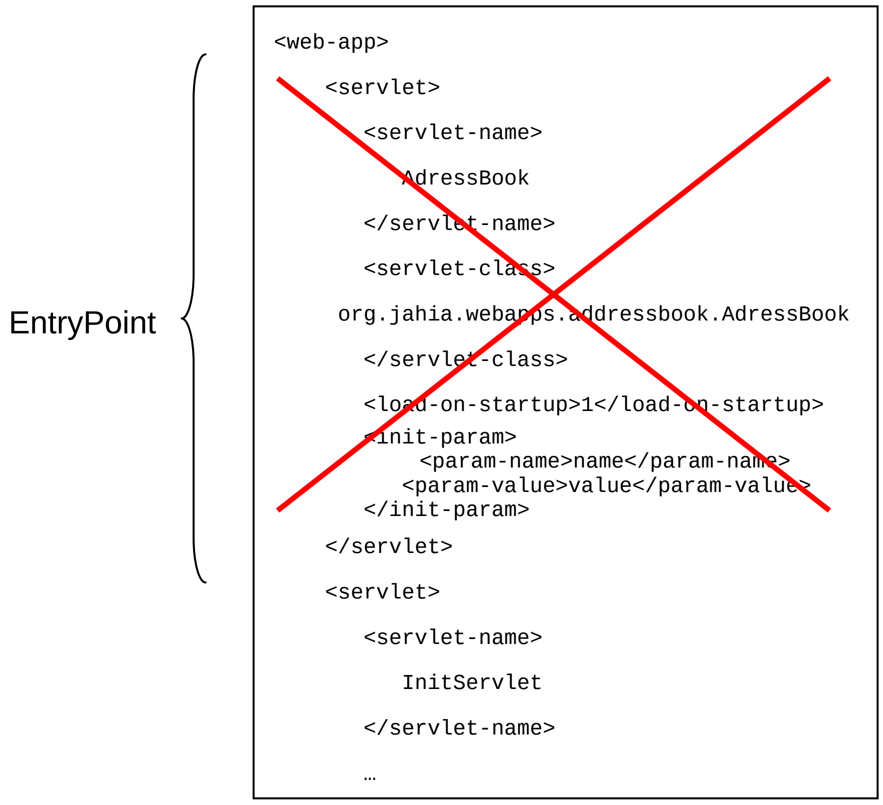
```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app version="1.0">
    <portlet>
        <description>
               Jahia WebApp Portlet Bridge
        </description>
        <portlet-name>AdressBook</portlet-name>
        <portlet-class>
org.jahia.portlet.JahiaWebAppPortletBridge
        </portlet-class>
        <!-- jahia webapp class -->
        <init-param>
              <name>jahiaWebAppClass</name>
              <value>
org.jahia.webapps.addressbook.AddressBookServlet
              </value>
        </init-param>
        <init-param>
              <name>name</name>
              <value> value</value>
        </init-param>
    </portlet>
</portlet-app>
```

The *portlet.xml* file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app version="1.0">
    <portlet>
        <description>Jahia WebApp Portlet Bridge</description>
        <portlet-name>AdressBook</portlet-name>
        <portlet-class>org.jahia.portlet.JahiaWebAppPortletBridge</portlet-class>
        <!-- jahia webapp class -->
        <init-param>
            <name>jahiaWebAppClass</name>
            <value>org.jahia.webapps.addressbook.AddressBookServlet</value>
        </init-param>
         <init-param>
             <name>name</name>
      <value> value</value>
        </init-param>
   </portlet>
</portlet-app>
```

2.2.3 UPDATING THE *WEB.XML* FILE

In the *web.xml* file, delete the servlet corresponding to the JahiaWebApp.

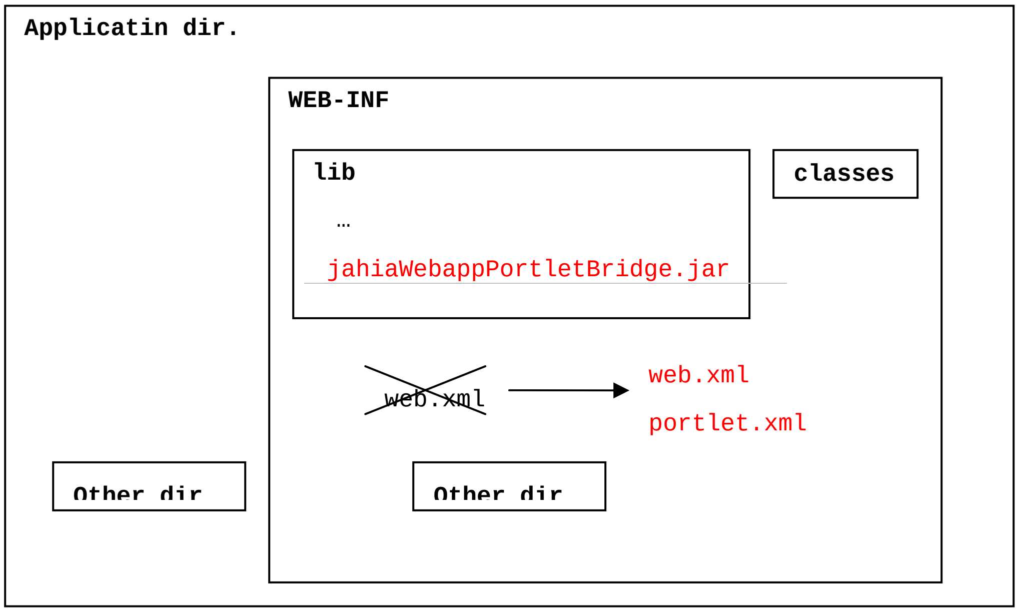


```
<web-app>
    <servlet>
       <servlet-name>
          AdressBook
       </servlet-name>
       <servlet-class>
     org.jahia.webapps.addressbook.AdressBook
       </servlet-class>
       <load-on-startup>1</load-on-startup>
       <init-param>
           <param-name>name</param-name>
          <param-value>value</param-value>
       </init-param>
    </servlet>
    <servlet>
       <servlet-name>
          InitServlet
       </servlet-name>
       …
```

In our example, the *web.xml* file becomes the following:

```
<web-app>
    <servlet>
       <servlet-name>
          InitServlet
       </servlet-name>
       <servlet-class>
         org.jahia.webapps.InitServlet
       </servlet-class>
             <load-on-startup>1</load-on-startup>
    </servlet>
</web-app>
```

2.2.4 DIRECTORY STRUCTURE



3 USING STANDARD FRAMEWORKS

Some JahiaWebApps are developed using a framework like Struts. It is possible to convert this type of JahiaWebApp using standard bridges (see the chapter on bridges). A special feature of most of these bridges is that their parameters can be adjusted using xml files.

Tip

This method is recommended for JahiaWebApps developed using a framework.

4 CONVERTING JAVA CODE AND JSPS

Finally, it is possible to convert the JahiaWebApp into an *actual* portlet. There are no standard methods for this. Nevertheless, some mapping rules must be defined:

- `HttpSession` → `PortletSession`
- `ServletContext` → `PortletContext`

The class that implements the servlet interface must implement the portlet interface.

The `service(…)` method is split between the `processAction(…)` and `render(…)` methods.

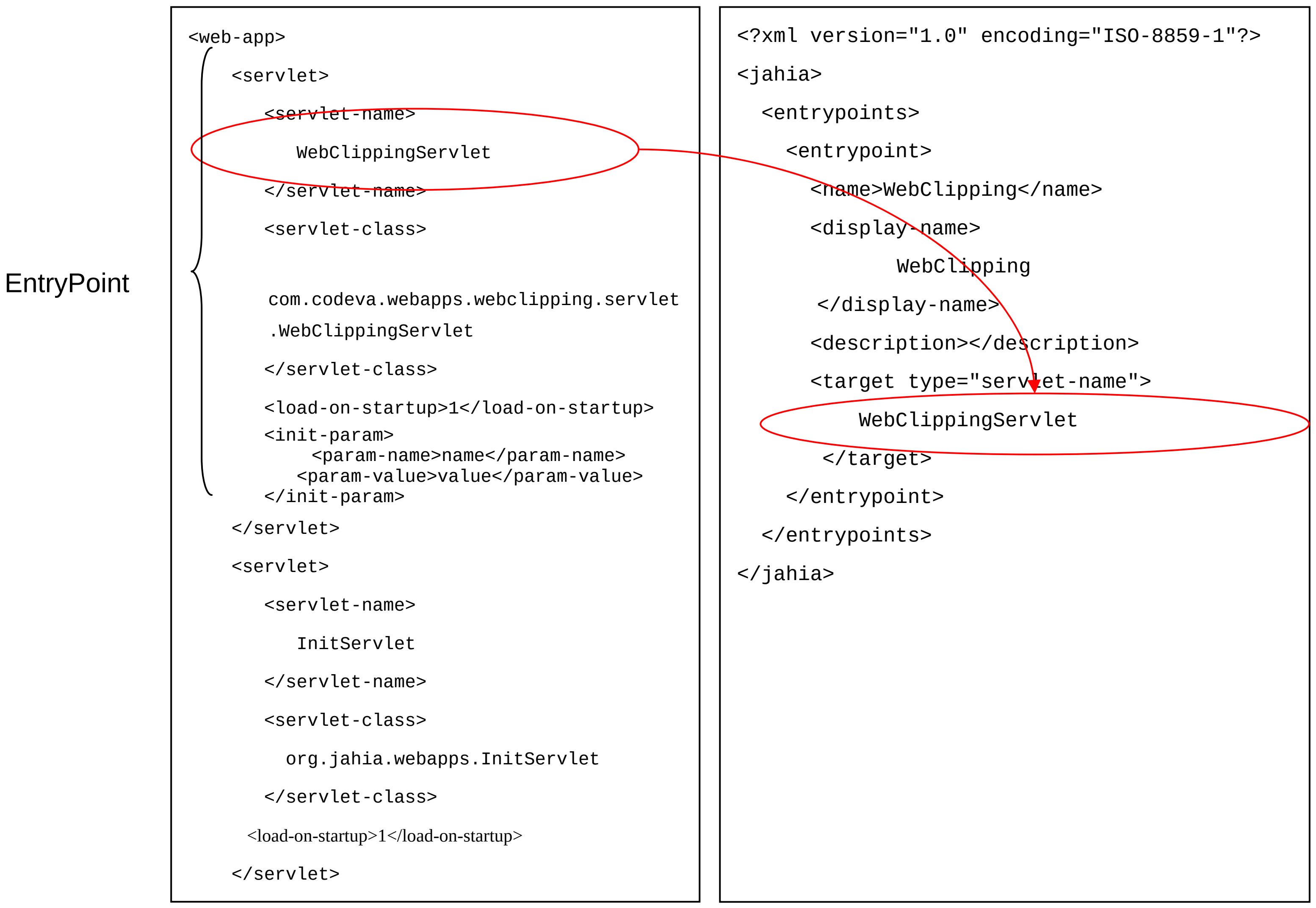
In JSPs, the portlet taglib argument must be imported and the `request` and `response` objects must be replaced by `renderRequest` and `renderResponse`.

5 ADDING THE *JAHIA.XML* FILE

Jahia 5 always supports JahiaWebApps. But, you need one file to specify the application entrypoint in order to migrate a Jahia 4 JahiaWebApp to Jahia 5: *jahia.xml*.

5.1 CREATING THE *JAHIA.XML* FILE

jahia.xml tells Jahia what servlet is the application entrypoint.



```
<web-app>
    <servlet>
        <servlet-name>
            WebClippingServlet
        </servlet-name>
        <servlet-class>

         com.codeva.webapps.webclipping.servlet
         .WebClippingServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
        <init-param>
             <param-name>name</param-name>
           <param-value>value</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>
            InitServlet
        </servlet-name>
        <servlet-class>
          org.jahia.webapps.InitServlet
        </servlet-class>
      <load-on-startup>1</load-on-startup>
    </servlet>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jahia>
  <entrypoints>
    <entrypoint>
      <name>WebClipping</name>
      <display-name>
             WebClipping
       </display-name>
      <description></description>
      <target type="servlet-name">
          WebClippingServlet
       </target>
    </entrypoint>
  </entrypoints>
</jahia>
```

5.2 ADDING THE *JAHIA.XML* FILE

After the *jahia.xml* is created, move it to the WEB-INF directory of the portlet.

Warning *This portlet is not a JSR 168. Therefore, this method is <u>not recommended</u>. Future versions de Jahia will not support JahiaWebApps.*

CHAPTER 6: USEFUL LINKS

The JSR 168 community continues to grow. This chapter lists various useful sites, discussion lists, and information sources for portlet development.

1 OPEN SOURCE PORTLETS

1.1 JAHIA PORTLET REPOSITORY

The Jahia team has developed a set of useful portlets, like a calendar server and a forum, which are open source. These portlets are available for download at:

http://www.jahia.net/jahia/page571.html

1.2 PORTLETREPOSITORY

PortletRepository is a workspace for open source developers who contribute to the development of various JSR 168 projects. This workspace contains useful JSR 168 portlets, Web 2.0 technologies, and more.

For more information go to:

https://portlet-repository.dev.java.net/public/Portlets.html

1.3 GEMS

The Gems project makes a set of useful JSR 168 portlets such as AdressBook, FileUpload, and GoogleSearch available to developers.

For more information go to:

https://gems.dev.java.net/

1.4 PA-LAB

The Pa-Lab project provides useful JSR 168 portlets such as administrator, blog, and Yahoo! Japan search portlets.

For more information go to:

http://sourceforge.jp/projects/pal/

2 BROADCAST LISTS

Developers use broadcast lists to share information and help each others. Here are a few interesting lists about JSR 168 portlets:

- Yahoo: http://tech.groups.yahoo.com/group/portlets/
- Bridge portlet: http://portals.apache.org/bridges/mail-lists.html

3 USEFUL LIBRARIES

3.1 PORTLET TOOLBOXES

Here is a list of libraries that provides interesting tools for portlet development:
http://www.doc.ic.ac.uk/~mo197/portlets/index.php

3.2 BLOG

Here is a blog about JSR 168 portlets: http://www.portlets-jsr168.blogspot.com/

CONCLUSION

The intent of this document is to serve as a guide for basic JSR 168 portlet development in JAHIA. The examples presented are simple in order to facilitate quick understanding of basic portlet concepts. We encourage getting familiar with the JSR 168 specification before developing more elaborate applications.

LIST OF ILLUSTRATIONS