WIDGETS TECHNICAL DOCUMENTATION

PORTAL FACTORY 2.0

SUMMARY

1 INTRODUCTION

Jahia Digital Experience Manager is a platform for managing a variety of digital initiatives in a productive and secure fashion. Initiatives such as building websites, mobile sites, intranets and portals, all of which can interact with visitors in order to deliver the best user experience possible.

This guide presents some technical features of Portal Factory, a new, advanced portal generation and management interface, available as an Add-on for Jahia Digital Experience Manager.

Portal Factory allows you to create Portal models for your site and give your users access to one or more Portals based on these models. They can also under circumstances customize these portals within the limits you define.

In this documentation, you will learn how to create custom Portal Widgets, how to use Portals in an existing Jahia Digital Experience Manager, and how to interact with the Portal JavaScript API. Explanations remain simple and factual, so this guide can help developers and web masters to get acquainted rapidly to the tools they will use.

For general administration and usage of Portal Models and User Portals, please refer to the Portal Factory User Guide, available on the Jahia Web Site.

2 CUSTOM PORTAL WIDGETS

You can create your own custom widgets and use them in any Portal. To do so, your widgets' content needs to respect a precise structure.

2.1 DEFINITIONS

A widget need to have its own definition and the mixin that specifies it as a portal widget.

```
[jnt:documentBrowserWidget] > jnt:content, jmix:portalWidget
- rootPath (string) = "/sites" indexed=no autocreated
```

In blue the mixin “jmix:portalWidget” specifies that this component is a portal widget, and that its use is allowed in portals.

In orange, the property option “autocreated” is useful for the widget, because when a widget is added to a portal page, the edit view of the widget is displayed but at this state the node is created and it is not sure which properties are initialized. The user is able to customize the properties, but if the property is not found in the views of your widget it is possible to get exceptions. That’s why we choose to put this option so this property is auto created at creation of the node.

You can also catch the exception when a property is not yet created in your views, but the approach is trickier to handle. Same caution applies with mandatory properties.

There is another mixin that can be put on your widget node type:

```
[jnt:scriptGadgetWidget] > jnt:content, jmix:portalWidget, jmix:portalGadget
- j:script (string, richtext) = "" indexed=no autocreated
```

In blue, the mixin “jmix:portalGadget” is used to specify that this specific widget component needs to be used without AJAX. By default, all portal widgets are loaded using AJAX, but some widgets use dynamic JavaScript includes or need to be loaded at the same time as the page. In order to achieve that, you can add this mixin to prevent the AJAX behavior on this type of widget.

For now, that's all you need to know about the widgets node type definition.

If you wish to delve deeper into writing Portal Widgets, you can take a look at all the default widgets brought by the module “Portal factory - widgets” https://github.com/Jahia/portal-factory_widgets and that you can see in the Portal Widget Zap.

2.2 VIEWS

Widget components have specific portal views; in the current portal implementation, the views used in the widgets (in the User Portal) are the following:

helloWorldWidget.portal.view.jsp: the default view for a widget

helloWorldWidget.portal.edit.jsp: the edit view for a widget (optional)

helloWorldWidget.portal.full.jsp: the full view for a widget (optional)

The following view is used in the core Portal Model view:

helloWorldWidget.portal.model.jsp : the view with the specific model form, that allows to edit portal widget model properties. (properties used at the user portal instantiation, that are not propagated to the user instance widget.)

There is no limit on the views you can use for a widget, since the JavaScript portal API allows you to load whichever view you want. You can also create your own views and just call the portal function `widget.load("view")`. In the current implementation, we chose to prefix all the portal-specific views with “portal” but this is not mandatory.

2.3 SKINS

Skins are the different layouts used for a widget. You can create your own skins as well. The only thing you need to do to create a new skin is to create a view for the portalWidget node type that looks like “portalWidget.box.<skin name>.jsp”

For example:

portalWidget.box.advanced_purple.jsp

portalWidget.box.advanced_blue.jsp

portalWidget.box.advanced_green.jsp

This naming convention uses Jahia's default wrapper system.

In the current portal implementation, the skins implement some features like:

- minimize/maximize the widget
- load the edit view when possible
- load the full view.

But you can create your own custom skins that fits your needs.

3 USING PORTALS IN YOUR SITE (PORTAL TEMPLATES)

To be able to use Portals in an existing Jahia Digital Experience Manager Web Site,

- Portal Factory Modules must be activated for this site
- Specific portal templates including portal content must be created through the Studio and added to your site.

3.1 ACTIVATE THE PORTAL MODULES FOR YOUR SITE

In order for Portals to work on your site, you first need to install and activate the Jahia Portal Factory modules through the Studio. Make sure the modules are installed in the following order, as dependencies could prevent them from working correctly otherwise.

1. Portal factory - core: https://github.com/Jahia/portal-factory_coreThis is the core module for the portal, it contains the JavaScript Portal API and the portal taglib, all the filters, actions, scripts, and views necessary to the portal to work correctly.
2. Portal factory - user portal: https://github.com/Jahia/portal-factory_user-portal These are the components related to the portal for end-users. These components rely on the portal JavaScript API and the portal taglib. The user portal contains:
    a. Portal toolbar: This toolbar displays the tabs and the actions available to create and edit portals
    b. User's portal list: A simple component displaying all the portals accessible by the current user.
    c. Portal widget zap: A component displaying all the drag&droppable widgets in the portal.
3. Portal factory - widgets: https://github.com/Jahia/portal-factory_widgets This is the portal widgets sample. (twitter widget, rss feed widget, script widget, document browser widget, etc …).

3.2 DEFINE PORTAL-ENABLED TEMPLATES

To be able to activate portals in your site, you need to have specific templates that drives the portal tabs. Those templates must contain contain portal area(s). Portal areas are special areas designed to allow the drag and drap of widgets within.

Adding these templates is required to be able to create a portal model. Portal models are explained in the Portal Factory User Guide on the Jahia website.

To learn how to create a portal template, you can explore the source code of some existing Portal templates in the module called 'Portal factory - default templates' : https://github.com/Jahia/portal-factory_default-templates This is the default template set that can be used directly to start creating portals in a blank site.

For more information about creating your own templates, a video “Portal documentation - how to template portals” will be available on the Jahia Web Site.

4 PORTAL JAVASCRIPT API

A portalContext Java Bean is available in all portal components; it is added automatically on portal pages and widget views. Java Bean is serialized in Javascript and also available in the portal JavaScript API.

4.1 PORTAL JAVASCRIPT API

The portal JavaScript API is available in any page of your portal. A Jahia render filter adds automatically the necessary JavaScript to the page, so you do not have anything to do manually to use it.

The portal API is made of 3 objects:

- **Portal**: the portal itself. Some actions are possible directly on this object like “init drag&drop”, “add new widget”, “get widget types”, …
- **Area**: a column of the portal. Nothing special here, just that you can access the jQuery object of this area.
- **Widget**: a portal widget. You can access this object through the “Portal” object. The actions possible on this object are: load a specific view for the current widget, update the widget, delete it, …

In the following sections we will see some examples.

4.1.1 How to retrieve the widget API object

If you do not know the widget html ID, you can use a sub html element to retrieve the parent widget object:

```
var widget = portal.getCurrentWidget("HTML SUBCHILD ID");
```

If you know the html ID of the widget, you can use this function:

```
var widget = portal.getWidget("WIDGET HTML ID");
```

4.1.2 How to load the edit view of the widget manually

Use the function `load`. This function takes the view string name as first parameter. You can add an optional callback as second parameter.

```
widget.load("portal.edit")
```

Note: you can use the same function to load any widget view.

4.1.3 How to delete a widget

To delete a widget, call the `deleteWidget` function from the portal object:

```
portal.deleteWidget(widget);
```

4.1.4 How to retrieve the list of widget types allowed in the current portal

You can call the `portalWidgetTypes` function from the portal object. In order to be able to retrieve this information, you need to have Write permissions on the Portal tab :

```
portal.portalWidgetTypes
```

The array of portal widget types displays the following format:

```
[{
name: "jnt:bookmarksWidget",
displayableName: "My bookmarks widget",
views:
[
{
path: "/modules/portal-
    factory_widgets/jnt_bookmarksWidget/html/bookmarksWidget.portal.edit.jsp",
key: "portal.edit"
},
{
path: "/modules/portal-
    factory_widgets/jnt_bookmarksWidget/html/bookmarksWidget.portal.view.jsp",
key: "portal.view"
}
]
}, ...]
```

4.1.5 How to add a new widget

The simplest way to add a new widget is the following; the widget will be added to the first column of the current tab:

```
portal.addNewWidget("Widget node type", "Widget desired name");
```

but if you wish to connect a draggable to the portal sortable areas, more code is needed :

Add the following data to your draggable element:

```
<div data-widget_nodetype="WIDGET NODE TYPE" data-widget_view="WIDGET VIEW"
    class="widget_external_drop">draggable new widget</div>
```

This is the minimal code needed on the draggable element. The data attributes name are constants in the portal API. The missing part is the connector code to the draggable areas:

```
element.draggable({
  connectToSortable: Jahia.Portal.default.sortable_options.connectWith,
  helper: "clone"
});
```

Where 'element' is the previous “div”. You need to set this minimal option to the draggable function. The connectToSortable is mandatory, you need to specify the sortable areas selector, for simply we have store the sortable class selector in a constant of the portal API. The helper clone is also mandatory, because a clone of the div will be created and the portal API will replace it with the widget and the view you previously specify in the data of your div.

To have a closer look at the code, you can take a look at how it is done in the component “Portal Widgets Zap” in the module portal-factory_user-portal.

4.1.6 How to retrieve the tabs of the portal

For better performance, the JavaScript AJAX calls to query the portal tabs has been removed, to avoid an overload of HTTP calls when many users are using portals at the same time.

So we add a JCR query manually in the Portal Toolbar component and iterate on each portal tab node. The upside to this approach is that the toolbar is cached per user. Users may see different portal tabs according to accessibility options on portal tabs.

In the Portal Toolbar component, we add information to the portal JavaScript object to store the result of the query.

```
<c:set var="portalNode"
   value="${renderContext.mainResource.node.parent}"/>
<script type="text/javascript">
   portal.portalTabs = [];
   var portalTab = {};
   <c:forEach items="${portalNode.nodes}" var="portalTab">
       <c:if test="${jcr:isNodeType(portalTab, 'jnt:portalTab')}">
           <template:addCacheDependency path="${portalTab.path}"/>
           <c:set var="isCurrent" value="${portalTab.identifier eq
   renderContext.mainResource.node.identifier}"/>
               portalTab = {
                   path:"${portalTab.path}",
                   displayableName: "${portalTab.displayableName}",
                   accessibility: "${not empty
   portalTab.properties['j:accessibility'] ?
   portalTab.properties['j:accessibility'].string : 'me'}",
                   skinKey:
   "${portalTab.properties['j:widgetSkin'].string}",
                   templateKey:
   "${portalTab.properties['j:templateName'].string}",
                   current: ${isCurrent},
                   url: "<c:url
   value="${url.baseLive}${portalTab.path}.html"/>"
               }
               portal.portalTabs.push(portalTab);
           <c:if test="${isCurrent}">
               portal.portalCurrentTab = portalTab;
           </c:if>
       </c:if>
   </c:forEach>
</script>
```

4.1.7 How to load a specific widget in a specific portal template

Like the full view for the widgets, you may need to load a widget in a specific template. To do that you can use the `loadInCurrentTab` function:

```
portal.loadInCurrentTab("widget node identifier", "widget view", "widget state",
   "widget need to be loaded alone", "portal template");
```

This function allows to load a widget in the current tab; you can use the parameters:

- widget node identifier: the jcr node id of the widget (you can get it on the widget API object)

- widget view: the widget view to use for the widget (you can get the list of the widget views using portal functions)

- widget state: it's a flag for identify the state of the widget, you can specify whatever you want, for the full view I use the "full" state, so I know if the widget is actually display full or not, to be able to get back to normal.

- widget need to be loaded alone: is a boolean that tell the portal to only load this widget when is set to true, or to also load the others widgets available in the current page.

- portal template: finally, the portal tab template you want to use to display the widget.

You can take a look at the JavaScript portalWidgetWrapper.js in the module "portal-factory_widgets". This is the controller for all the widgets in the current implementation. The function is used to load widgets in full state.

4.1.8 How to instantiate a user portal from the Portal model

Just call the function:

```
portal.initPortalFromModel();
```

You can use a callback as parameter.

4.2 PORTAL TAGLIB

Portal Factory comes with a JSP taglib available to help you get some portal related information and objects in your JSPs.

In order to use it, you need to add the dependency to "portal-fatory_core" module in your own module and add the following import in your JSP headers:

```
<%@ taglib prefix="portal" uri="http://www.jahia.org/tags/portalLib" %>
```

A useful function is:

userPortalsBySite: returns all visible portals for the current user for the given site

4.3 PORTALCONTEXT

The portalContext is a Java Bean available in all JSP and render scripts used for portal components. To access this portalContext in your render scripts for your content, you need to add explicitly the mixin : `jmix:portalContextWatcher` to your content type.

By default, it is accessible in widget views. The portal toolbar is a sample component that uses portalContext, to display or hide specific information.

You can take a look at the Java Bean `org.jahia.modules.portal.service.bean.PortalContext.java`

```
private String path;
private String identifier;
private String tabPath;
private String tabIdentifier;
private String modelPath;
private String modelIdentifier;
private String fullTemplate;
private String baseUrl;
private List<PortalKeyNameObject> portalTabSkins;
private List<PortalKeyNameObject> portalTabTemplates;
private SortedSet<PortalWidgetType> portalWidgetTypes;
private boolean isLock;
private boolean isModel;
private boolean isCustomizable;
private boolean isEditable;
private boolean isEnabled;
private boolean isDebug;
private int siteId;
```

All this information is available from the portalContext, and since it is serialized and used in the JavaScript API, it's also available in JavaScript client code.

4.4 MORE EXAMPLES

For more code samples, you can dig into the portal JavaScript API from the Portal Factory sources. There, you will find some JS documentation on functions. The file is located in the module "portal-factory_core" and is named: jahia-portal.js

You can also take a look at the different portal components like the Portal toolbar, Portal Widgets Zap, to be found in the module "portal-factory_user-portal".

And finally you can explore the code of all the sample widgets that are in the module “portal-factory_widgets”. Most of them are AngularJS applications and use the portal JavaScript API to perform actions in the portal.