jahia

DIGITAL EXPERIENCE MANAGER – BEST PRACTICES

SUMMARY

1. INTRODUCTION

This document provides the best practices to keep in mind when developing Jahia modules on Digital Experience Manager. These tips will help developers minimize errors, optimize performance, and improve code reusability.

NB : these best practices also apply to Digital Factory 7.0.X.X

JAHIA TERMINOLOGY

1.1. TEMPLATE

A template is a specific type of Jahia module, which provides pages and content templates to a Jahia site. Templates can be used to do the following :

- Create specific areas to allow editors/contributors to add content
- Load resources needed by a page (CSS, Javascript)
- Define common reusable elements on a page (example : navigation menu)

During page rendering, the content (which are nodes under a page) and the elements defined on the templates, are merged to provide the page.

1.2. MODULE

A module is a package which contains resources (JSP, CSS, Javascript, Drools rules, Java code, …) and components. Modules are the primary way of adding new functionalities to Digital Experience manager. Modules are like Jahia “plugins” that provide a way to add new features and functionalities.

1.3. COMPONENT

A component (functional equivalent to “node type”) is a content which can be instantiate within a template or a site.

Components can be associated to view(s) in order to be displayed (example : news component, navigation menu component, rich text component). They can have properties or even complex data structure, and they can be associated to Jahia advanced mechanisms (filters, actions, rules, …)

2. DEVELOPMENT BEST PRACTICES

2.1. NAMING CONVENTIONS

2.1.1. Namespaces

The "Compact Namespace and Node type Definition" (CND) gives us a simple syntax for defining node types and declare namespaces regrouping them.

Declaration of a namespace and a node type example :

```
/*  An example node type definition */
// The namespace declaration
<ns = 'http://namespace.com/ns'>

// Node type name
[ns:nodeType]
```

As the name of a node type is unique, namespaces protect against name conflicts which could occurs if you have several modules installed on your DX. We suggest to reproduce Jahia's schema by having two namespaces by project :

- One related to primary node types (example : <jnt='http://www.jahia.org/jahia/nt/1.0'>)
- One related to mixin node types (example : <jmix='http://www.jahia.org/jahia/mix/1.0'>)

But you could also multiply namespaces for categorizing components or for avoiding conflicts between two developers team working on a same Jahia instance.

2.1.2. Modifications of node types definitions

Modifications of the definitions.cnd file have to be made with caution, if content have already been created with this node types definitions it could lead to content integrity issues.

Type of modification	Operation	Comment
Namespace	Creation	Will not create problem
Namespace	Deletion	Should never be done. Instead of a deletion, stop using the previous namespace
Namespace	Modification	Should never be done. Instead of a modification, create a new namespace and stop using the previous one
Node type	Creation	Will not create problem
Node type	Deletion	Should never be done before having deleted all the instantiated nodes of this type from the template/site Instances of this node could be found and deleted using Jahia Tools/JCR Console. It is also possible to script (groovy) this operation
Node type	Modification	Renaming a node type is similar to perform a deletion of the previous node type, and creation of a new one
Property of a node type	Creation	Will not create problem
Property of a node type	Deletion	Should never be done before having set the property to "null" on all the instantiated nodes, otherwise it will lead to publication issues. Alternative possibility is to declare this property "hidden", and cease using it.
Property of a node type	Modification	Should never be done if there is node instantiated with this property. If necessary, create a new property and refer to "Deletion" section above.

2.1.3. Modules

It is impossible to deploy two modules having the same name. To avoid this kind of conflict, we recommend to prefix all modules of a project by a common key referring to the project (example : "aelb-intranet-template, aelb-intranet-components").

2.1.4. Components

Regarding components droppable/usable by contributors, Jahia provides a set of categories for classifying them (example : "Basic content", "Advanced content", "Form content", ...).

You can use existing categories or add new ones for your project. To do so, create a mixin inheriting of "jmix:droppableContent" :

```
[namespace:nomDeLaCategorie] > jmix:droppableContent mixin
```

2.2. COMPONENTS CREATION

2.2.1. Properties manipulation

2.2.1.1. Displaying a property

When you want to display a property which is not mandatory, it is usually recommended to first check if this property has a value, for avoiding to open empty HTML elements :

```
     <c:set var="textProp" value="${currentNode.properties.text}"
/>
     <c:if test="${not empty textProp}">
          <div>${textProp.string}</div>
     </c:if>
```

2.2.1.2. Displaying properties from another node

When the view of a component should display properties from another node (example : list displaying children elements, display of a weakreference, …), the following code, displaying directly other node's properties, should be avoid :

```
<h2>${currentNode.properties['jcr:title'].string}</h2>
<h3>Company</h3>
${currentNode.properties.AnotherNode.node.properties['jcr:title'].st
ring}
```

A direct manipulation of another node property will lead to cache issue.

In this case, the view should delegate the rendering of these properties directly to the other node. If the other node has no view, we should add one. Then in our first view, instead of displaying directly these properties, we ask the other node to display itself :

```
<h2>${currentNode.properties['jcr:title'].string}</h2>
<h3>Company</h3>
```

```
<template:module node="${currentNode.properties.AnotherNode.node}"
view="hidden.nameOfTheView" />
```

Where the view "hidden.nameOfTheView" is defined like this :

```
${currentNode.properties['jcr:title'].string}
```

Another possibility is to continue displaying directly the other node's property, but you then need to add a cache dependency toward this node. In many cases this solution is more costly than delegating the rendering. If the other node is modified, you will have to render entirely the view of your component, instead of just the part related to the other node.

2.2.1.3. Title property

Several editorial components need a property title. Instead of defining a new string property, the best practice is to inherit the mixin "mix:title", which directly provide a property named "jcr:title" :

```
[nt:editorialComponent] > jnt:content, mix:title
- image (weakreference, picker[type='image'])
```

This way, a reference will be created between your component's title and the system-name of its instantiation. The system-name will then have more meaning, while exploring the repository (or creating weakreferences), giving the editor a better experience than the default system-names.

If you wish to display the title coming from "mix:title" mixin, you could do it this way :

```
<h2>${currentNode.properties['jcr:title'].string}</h2>
```

2.2.1.4. Display an image child node

When a component has a weakreference property of type image, instead of directly displaying the image, it is possible to use the native Jahia view (imageReference.jsp) this way :

```
<c:if test="${not empty imageProperty}">
<template:module node="${imageProperty.node}" editable="false"/>
</c:if>
```

It will :

- Be faster for the developer
- Handle automatically “alt” attribute of the image with the description field of this node (if editor/contributor filled it)
- Automatically add width and height attributes, using the size of the image, for saving space for the image in the page (which will avoid resizing of the page during page loading)
- Handle correctly cache for the image, without having to add a cache dependency

Drawback :

- If new attributes have to be handled (CSS class for instance), you will not be able to use the native view. In this case, you have to handle manually the display, or you can add a new view to **imageReference**

2.2.2. Handling URL

When handling URL (example : displaying an internal link toward another node or a resource), it is important to use the taglib **c:url**. This mechanism allows Jahia to perform rewriting rules for preview/live mode, and it also enable Vanity URLs.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url value="/images/image.png” context="${url.currentModule}"/>
```

2.2.3. Externalization of properties

Every module natively provides a file “.properties”, you can find in src/main/resources/resources, and initialy named “nameOfTheModule.properties”.

These properties files are used to save key/value parameters.

2.2.3.1. Key unicity

When resolving a value contained in a properties file, DX search for the key in all the properties files contained in deployed modules. It is important to ensure unicity for your keys.

To do so, you can prefix your key by the name of the component using it :

```
nt_nameOfTheComponent.label.key = Bla bla
```

2.2.3.2. Property/component name

By default, the name of your component and properties is not really “friendly” for editors/contributors. We recommend providing a label for your properties, to be displayed in the back-office while creating/modifying content.

In order to do so, you have to follow a naming convention for your labels :

- For component, the key is : *namespace_nameOfTheComponent*
- For a property, the key is : *namespace_nameOfTheComponent.nameOfTheProperty*

Example, for the following component :

```
[nt:editorialComponent] > jnt:content, mix:title
 - image (weakreference, picker[type='image'])
```

Here are the keys to define for having friendlier labels :

```
nt_ editorialComponent = Editorial component
nt_ editorialComponent.image = Visual content
```

2.2.3.3. Labels in views (JSP)

Hardcoded labels should be avoided in JSP for several reasons (no possible internationalization, no reutilisation, …). In this case, properties files could be used to provide labels.

To do so, first you need to declare a new entry in your properties file, example :

```
nt_editorialComponent.label.author = Author
```

Then, inside your JSP, you can retrieve the value behind the key, using the taglib fmt :

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
…
<fmt:message key=”nt_editorialComponent.label.author” />
```

2.2.3.4. Internationalization

You can add as many .properties files as you wish/need, as you can have a .properties file for every needed locale.

Create a new .properties file, in the same folder, for every handled locale, by naming it this way :

- nameOfTheModule.properties : default file used as a fallback or when the user has no locale specified
- nameOfTheModule_**de**.properties : file used for German users
- nameOfTheModule_**en**.properties : file used for English users
- nameOfTheModule_**fr**.properties : file used for French users
- …

2.2.4. Cache

2.2.4.1. General rules

When a cache issue is raised on a component or a view, developers tend to deactivate cache on this component. This could be legitimate in a very few cases, but normally **the cache should never be totally deactivate**. Only a few cases could lead to deactivation :

- Displaying data from an external source (Webapp, RSS, external API, …) with a strong constraint on having no out-dated data
- Displaying data on which you cannot add a cache dependency, with a strong constraint on having no out-dated data

Even in such cases, we strongly recommend not to deactivate cache, but instead, having a lower time to live cache configuration (a few minutes for instance).

If you have no other choice, and you have to deactivate the cache for a component, you may consider having to deactivate it on only a small part of the JSP. To do so, you have to provide two views for your component :

- A view displaying only the not cacheable data : this view will have no cache
- A view displaying the rest of the component, and performing a template:include of the first view

Example : given a component having a title, a description and displaying data coming from an external Webapp, we will create two views :

- viewA : performs the call to the Webapp, and displays its data
- viewB : display title and description, and include the viewA

View B :

```
<c:set var="title" value="${currentNode.properties['jcr:title']}"/>
<c:set var="description" value="${currentNode.properties['description']}"/>

<c:if test="${not empty title}">
   <h2>${title.string}</h2>
</c:if>

<c:if test="${not empty description}">
   <span class="description">${description.string}</span>
</c:if>

<%-- Including the view fetching and displaying external data --%>
<template:include view="viewA"/>
```

2.2.4.2. Testing in live mode

In preview and edit/contribute modes, cache is not being handled. One common mistake is to test your components only in preview mode, avoiding cache issues.

It is imperative to **test the rendering of your components directly in live mode**.

2.2.5. Permissions

When using "root" user, permissions are never being checked. If the rendering of one your views is based on a permission check, you must test it with a non root user.

2.3. LOGGERS

DX uses log4j logging framework. It is recommended to use logger inside your Java classes (and eventually your Drools rules and Groovy scripts), logging at least informations which could be used for debugging.

By example, if you expose a REST action requiring input POST parameters, it could be interesting to log at least :

- In debug level, input values
- If a mandatory parameter is missing, in warning level, a message specifying the missing parameter

2.3.1. Using a logger in a Java class

How-to define a logger in your Java class :

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
…
private static final Logger LOG =
LoggerFactory.getLogger(NomDeVotreClasseJava.class);
```

Using the logger :

```
// Will display a log if log level is set to info or superior
LOG.info("This is a info level log");

// Will display a log if log level is set to warning or superior
LOG.warn("This is a warning level log");

// Will display a log if log level is set to debug or superior
LOG.debug("This is a debug level log");

// Will display a log if log level is set to error or superior
LOG.error("This is an error level log");
```

2.3.2. Using a logger in a Drools rule

When defining a new Drools rule in a .drl file, it is possible to directly use Java for creating and using a logger.

A logger set to info level is provided by default, a Drools consequence is provided by DX for this purpose :

```
[consequence][]Log {message}= logger.info({message});
```

Using this logger could then be done this way :

```
rule "Import file"
  salience 50
  when
      A new node is created
      The node has a parent
           - it has the extension type jnt:importDropBox
  then
      Import the node
      Log "Import file fired for node " + node.getPath()
end
```

2.3.3. Using a logger in a Groovy script

It is possible to execute Groovy scripts from different ways :

- Directly in the Groovy Console : /tools/jcrConsole.jsp
- By putting the script directly in the folder /digital-factory-data/patches/groovy . By doing so, scripts could be executed at the startup of DX, when the JCR is ready but before allowing users to perform modifications on it, or directly when the script is dropped in the folder

When executing groovy scripts, a logger is automatically instantiated and can be used this way :

```
…
log.info("Will be logged when executing my groovy script")
…
```

2.3.4. Modify log levels for debugging

For debug purposes, sometimes it is interesting to change log levels for a class or a package. Instead of modifying the log4j.xml (/WEB-INF/etc/config/log4j.xml) it is possible to override the configuration directly from the tools administration.

This overriding is temporary, the configuration from the log4j.xml file will be reapply at the next DX startup.

To modify loggers configuration from the tools, go on this page : /tools/log4jAdmin.jsp

2.4. CODE MUTUALIZATION

As much as possible it is important to mutualize code, for various reasons (time saving, improved maintainability, ...).

This mutualization could and must be done on several levels.

2.4.1. Mutualization between modules

When components are being used by different projects (for instance, two different sites of a customer, such as an intranet and an extranet), we recommend creating a new **"transversal" module** defining these components.

It is especially true when exposing Actions, Java API or taglibs.

This way, two sites could use this module, without having to have the other project's components.

NB : if this need is identified lately in a project, moving components from a module to another one could be difficult and costly, especially if these components have already been instantiated on a site

2.4.1.1. Java resources

If you have Java classes/packages being mutualized, two manipulations have to be performed for allowing theses classes to be used in another module :

- An **export package** of these classes in the pom.xml of the transversal module
- In the other module, add a dependency toward the transversal module in its pom.xml

Example of an export package in the pom.xml of the module exposing Java classes :

```
<build>
   <plugins>
   …
        <plugin>
           <groupId>org.apache.felix</groupId>
               <artifactId>maven-bundle-plugin</artifactId>
               <extensions>true</extensions>
               <configuration>
                   <instructions>
                       <Jahia-Module-Type>module</Jahia-Module-Type>
                    <Jahia-Depends>default</Jahia-Depends>
                    <Export-Package>org.jahia....tag</Export-Package>
                  </instructions>
               </configuration>
        </plugin>
   …
   </plugins>
</build>
```

Example of dependency in the pom.xml of the other module :

```
<dependencies>
   …
   <dependency>
        <groupId>groupIDOfTransversalModule</groupId>
        <artifactId>artifactIDOfTransversalModule</artifactId>
        <version>2.5.0</version>
        <scope>provided</scope>
   </dependency>
   …
</dependencies>
…
<build>
   <plugins>
   …
        <plugin>
           <groupId>org.apache.felix</groupId>
               <artifactId>maven-bundle-plugin</artifactId>
               <extensions>true</extensions>
               <configuration>
```

```
                    <instructions>
                        <Jahia-Module-Type>module</Jahia-Module-Type>
                     <Jahia-Depends>default,
            artifactIDOfTransversalModule </Jahia-Depends>
                    </instructions>
                </configuration>
        </plugin>
    …
    </plugins>
</build>
```

2.4.2. Mutualization between JSP

When you have common code across several JSP, it is recommended to sort this common code in a transverse JSP. Regarding the context, it could be done differently.

2.4.2.1. Common code across views of a component

When you have common code across several views of a same component, the easiest way to proceed is to create a new view hidden.

Then you just have to perform a template:include where the code is needed, eventually sending to the view some parameters :

Example of a view using an include :

```
…
<%-- Including the hidden view --%>
<template:include view="hidden.nameOfMyView">
 <%-- We could send parameters to the view --%>
 <template:param name="parameter1" value="${variable}"/>
</template:include>
…
```

Where the view “hidden.nameOfMyView” will display the code to be mutualized. If needed, you can get parameters this way :

```
${currentResource.moduleParams.parameter1}
```

2.4.2.2. Common code across several components

When several components have common code, you could mutualize this code by creating a mixin having a view with this code.

After this, you have to modify your components, adding them inheritance with this mixin. Then you can include the view directly.

Definitions example :

```
[mix:commonBehavior] mixin

[nt:componentA] > jnt:content, mix: commonBehavior
- field1 (string, richtext) i18n mandatory
- field2 (weakreference, picker[type='image'])
- …

[nt:componentB] > jnt:content, mix: commonBehavior
- field1 (string) i18n mandatory
```

Include of the view example :

```
<template:include view=”nameOfTheViewOfCommonBehaviorMixin” />
```

3. DEPLOYMENT BEST PRACTICES

Since Digital-Factory 7, OSGi is being used for packaging Jahia modules. This leads to version handling and the possibility to perform hot deployment.

OSGi modules are simples JAR files you have to deploy directly in the folder **digital-factory-data/modules**. You can also deploy these JAR using the administration panel of DX.

When deploying a new version of a module, the behavior will be different if you are in development or production mode :

- Development mode : the previous version of the module will be stopped, and the new one will be automatically started
- Production mode : the new version will stay in stop status, you will have to manually activate it

When in a cluster architecture, deployment process will be different if you want to perform a hot or a cold deployment.

3.1. COLD DEPLOYMENT IN A CLUSTER

Copy the JAR(s) in the folder *digital-factory-data/modules*.

When restarting the cluster, you **must restart processing server first**. Before restarting other nodes of the cluster, you must wait for the processing server's initialization to be over.

3.2. HOT DEPLOYMENT IN A CLUSTER

Copy the JAR(s) in the folder *digital-factory-data/modules* on **all the non processing-server nodes**. Once the deployment is over on those nodes, do the same on the **processing server**.