OSGI MODULE DEVELOPMENT

DIGITAL EXPERIENCE MANAGER 7.2
TABLE OF CONTENT

1 INTRODUCTION ........................................................................................................................................6

2 WHAT IS OSGI? .........................................................................................................................................7

  2.1 A minimal OSGi bundle ......................................................................................................................7

3 WHY OSGI? ..............................................................................................................................................8

4 DIGITAL EXPERIENCE MANAGER AND OSGI ....................................................................................9

  4.1 What’s new? .........................................................................................................................................9

5 DIGITAL EXPERIENCE MANAGER OSGI ARCHITECTURE ...............................................................11

  5.1 Architecture overview .......................................................................................................................11
  5.2 Request flow ......................................................................................................................................12
  5.3 Class loader graph ............................................................................................................................13

6 BUILDING A NEW OSGI MODULE .......................................................................................................15

  6.1 Using the Digital Experience Manager Studio to create a new project... .......................................15
  6.2 Using a Maven Archetype ...............................................................................................................15
  6.3 From scratch .....................................................................................................................................16

7 OSGI PACKAGE DEPENDENCIES AND EXPORTS .........................................................................19

8 DEPLOYING AN OSGI MODULE IN DIGITAL EXPERIENCE MANAGER .......................................21

  8.1 Deploying using the Jahia Maven Plugin .........................................................................................21
  8.2 Un-deploying/re-deploying a module ............................................................................................22

9 SERVICE SHARING BETWEEN MODULES .......................................................................................23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Spring osgi:service and osgi:reference tags</td>
<td>23</td>
</tr>
<tr>
<td>9.2</td>
<td>OSGi:list usage</td>
<td>24</td>
</tr>
<tr>
<td>9.3</td>
<td>Export-Package instruction</td>
<td>24</td>
</tr>
<tr>
<td>9.4</td>
<td>Import-Package instruction</td>
<td>24</td>
</tr>
<tr>
<td>9.5</td>
<td>Other service declaration and referencing mechanisms</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>DEPLOY-FREE CODING</td>
<td>26</td>
</tr>
<tr>
<td>10.1</td>
<td>How to use it</td>
<td>26</td>
</tr>
<tr>
<td>10.2</td>
<td>How it works</td>
<td>26</td>
</tr>
<tr>
<td>10.3</td>
<td>Releasing modules</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>DIGITAL EXPERIENCE MANAGER OSGI TOOLING</td>
<td>29</td>
</tr>
<tr>
<td>11.1</td>
<td>Jahia Maven Plugin</td>
<td>29</td>
</tr>
<tr>
<td>11.1.1</td>
<td>jahia:dependencies goal</td>
<td>29</td>
</tr>
<tr>
<td>11.1.1.1</td>
<td>Package scanning</td>
<td>31</td>
</tr>
<tr>
<td>11.1.1.2</td>
<td>Content definition scanning</td>
<td>32</td>
</tr>
<tr>
<td>11.1.1.3</td>
<td>Troubleshooting dependencies</td>
<td>34</td>
</tr>
<tr>
<td>11.1.1.4</td>
<td>Deactivating/controlling content definition scanning</td>
<td>35</td>
</tr>
<tr>
<td>11.1.2</td>
<td>jahia:find-package-uses goal</td>
<td>35</td>
</tr>
<tr>
<td>11.1.3</td>
<td>jahia:find-packages goal</td>
<td>37</td>
</tr>
<tr>
<td>11.1.4</td>
<td>jahia:osgi-inspect goal</td>
<td>38</td>
</tr>
<tr>
<td>11.2</td>
<td>Felix Maven Bundle Plugin</td>
<td>38</td>
</tr>
<tr>
<td>11.2.1</td>
<td>More information about the plugin</td>
<td>38</td>
</tr>
<tr>
<td>11.3</td>
<td>The Manage Modules administration UI</td>
<td>39</td>
</tr>
<tr>
<td>11.4</td>
<td>The Apache Felix Web Console</td>
<td>40</td>
</tr>
<tr>
<td>11.5</td>
<td>KARAF console</td>
<td>40</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Command line configuration</td>
<td>41</td>
</tr>
<tr>
<td>11.5.2</td>
<td>Accessing the shell</td>
<td>41</td>
</tr>
<tr>
<td>11.5.3</td>
<td>JCR Commands</td>
<td>42</td>
</tr>
<tr>
<td>12</td>
<td>USING LIBRARIES IN AN OSGI MODULE</td>
<td>44</td>
</tr>
</tbody>
</table>
# OSGi Module Development

## OSGi Module Development

### 12.1 Finding OSGi bundles .......................................................... 44

### 12.2 Embedding non-OSGi libraries ........................................... 45

### 12.3 Transforming non-OSGi libraries into bundles ....................... 46

#### 12.3.1 Static transformation ..................................................... 47

#### 12.3.2 Dynamic transformation ............................................... 49

### 12.4 Deploy non-OSGi libraries at the framework level (or above) ....... 49

### 13 OSGi and Java Profiling .......................................................... 51

### 14 Converting an Existing Module into an OSGi Module ................. 52

#### 14.1 Digital Experience Manager's automatic module transformation .... 52

#### 14.1.1 Controlling automatically generated package imports and exports .... 52

#### 14.2 Source code changes ........................................................... 53

#### 14.2.1 Using the Jahia Maven Plugin ......................................... 53

#### 14.2.2 Manually upgrading the source code ................................ 54

##### 14.2.2.1 Changing the Maven POM ......................................... 54

##### 14.2.2.2 Moving directories .................................................... 55

#### 14.2.3 Rules ............................................................................. 55

#### 14.2.4 Hibernate ....................................................................... 56

#### 14.2.5 Workflow ....................................................................... 56

#### 14.2.6 Tag libraries ................................................................... 57

### 15 Troubleshooting ..................................................................... 58

#### 15.1 Common OSGi-related errors and their solutions ..................... 58

#### 15.1.1 Missing dependencies when deploying OSGi module ............ 58

#### 15.1.2 JSP compilation error due to missing MANIFEST dependencies ...... 61

##### 15.1.2.1 Possible solutions ..................................................... 61

#### 15.1.3 LinkageError .................................................................. 62

##### 15.1.3.1 Solution .................................................................... 62

#### 15.1.4 Large amount of imports generated by the Maven plugins ........ 62

##### 15.1.4.1 Solution .................................................................... 62
## 16  Digital Experience Manager OSGI Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>OSGi Framework startup</td>
<td>64</td>
</tr>
<tr>
<td>16.2</td>
<td>OSGi Servlet Bridge</td>
<td>64</td>
</tr>
<tr>
<td>16.3</td>
<td>Digital Experience Manager Module Extender</td>
<td>64</td>
</tr>
<tr>
<td>16.4</td>
<td>Bundle packaging</td>
<td>64</td>
</tr>
<tr>
<td>16.5</td>
<td>OSGi and Digital Experience Manager Module States</td>
<td>65</td>
</tr>
<tr>
<td>16.5.1</td>
<td>Bundle life cycle</td>
<td>65</td>
</tr>
</tbody>
</table>

## 17  Appendix 1 - Configuring a Module that Extends the System

## 18  Appendix 2 - Additional Resources

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1</td>
<td>OSGi Glossary</td>
<td>77</td>
</tr>
<tr>
<td>18.1.1</td>
<td>General OSGi terms</td>
<td>77</td>
</tr>
<tr>
<td>18.1.2</td>
<td>OSGi core implementations</td>
<td>77</td>
</tr>
<tr>
<td>18.1.3</td>
<td>OSGi compendium frameworks &amp; implementations</td>
<td>77</td>
</tr>
<tr>
<td>18.1.4</td>
<td>Other important frameworks</td>
<td>78</td>
</tr>
<tr>
<td>18.1.5</td>
<td>Other terms</td>
<td>79</td>
</tr>
<tr>
<td>18.2</td>
<td>Web references</td>
<td>79</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Starting with Digital Factory 7.0, new modules must now be written using OSGi bundles. As OSGi is a mature and powerful dynamic module system for Java, it becomes a lot easier to build full-fledged Digital Experience Manager modules that can interact with each other, while avoiding complex interdependencies that might make maintenance and deployment complex. At the same time, it makes it possible to leverage the already available OSGi bundles such as the Felix Web Console or the Felix Shell to quickly add functionality to a Digital Experience Manager installation.
WHAT IS OSGI?

OSGi (aka Open Services Gateway initiative) is a dynamic module system for Java. It is currently the most powerful and most mature dynamic module system available for Java applications. In OSGi, modules are actually called “bundles”. They are specialized JARs that include extra MANIFEST.MF entries that declare the dependencies between modules, as well as versioning and package information made available to other modules. In effect, most existing JAR can be converted to an OSGi bundle with little work (they are even automatic transformation tools available). In the OSGi runtime, only packages that are exported will be available and visible to other bundles, only if the using bundles also import them. So in effect there can be fine-grained control of accessible Java packages (as well as associated versions) between bundles.

2.1 A MINIMAL OSGI BUNDLE

An OSGi bundle is basically a classic Java JAR file with additional metadata information inside the META-INF/MANIFEST.MF file, such as:

- Bundle identifier (symbolic name)
- Bundle version
- Bundle package imports and exports
- (Optional) Bundle activator

Here is an example of a minimal OSGi bundle:

```
META-INF/MANIFEST.MF:
  Bundle-SymbolicName: org.jahia.modules.example
  Bundle-Version: 1.0
```
3 WHY OSGI?

Taken from the OSGi’s official website:

“From the developers point of view:

OSGi reduces complexity by providing a modular architecture for today’s large-scale distributed systems as well as small, embedded applications. Building systems from in-house and off-the-shelf modules significantly reduces complexity and thus development and maintenance expenses. The OSGi programming model realizes the promise of component-based systems.

From the business point of view:

The OSGi modular and dynamic model reduces operational costs and integrates multiple devices in a networked environment, tackling costly application development, maintenance and remote service management.

The key reason OSGi technology is so successful is that it provides a very mature component system that actually works in a surprising number of environments. The OSGi component system is actually used to build highly complex applications like IDEs (Eclipse), application servers (GlassFish, IBM Websphere, Oracle/BEA Weblogic, Jonas, JBoss), application frameworks (Spring, Guice), industrial automation, residential gateways, phones, and so much more.”

Despite an initial learning curve that requires learning how to setup OSGi modules properly (especially their dependencies), OSGi benefits quickly make themselves visible in even small projects.
4  DIGITAL EXPERIENCE MANAGER AND OSGI

In version 7.0, Digital Factory introduced the possibility to use OSGi to package and distribute Digital Factory modules. This means that you can now dynamically deploy and undeploy modules into a Digital Experience Manager installation, making it easier to manage modules during both development and production phases.

4.1  WHAT’S NEW?

In the following table we show the main differences in module development before and after OSGi introduction in Jahia’s Digital Experience Manager.

<table>
<thead>
<tr>
<th></th>
<th>Before OSGi</th>
<th>With OSGi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class or library deployment</td>
<td>Requires web app restart</td>
<td>No restart needed</td>
</tr>
<tr>
<td>Module is &quot;exploded&quot; on deployment</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Quick changes to source files don't require deployment</td>
<td>Only works in exploded directory, not module's source</td>
<td>Yes</td>
</tr>
<tr>
<td>External libraries are deployed into WEB-INF/lib and exposed to all other modules</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Undeployment cleans up everything immediately</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Modules depending on others cannot be deployed without their dependency</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Modules started/stopped after installation</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
For administrators:

- Modules can now be fully undeployed at run-time (including module libraries)
- No more writing inside web application /modules directory at deployment
- Deploy directly modules from public or private stores
- New tools for module administration & debugging
- Only declared resources are web accessible (closed by default)
- And more...

For developers:

- Undeploy and redeploy any module code changes without restarting Jahia server
- Deploy-free Coding for JSPs, static files loads updates directly from module source code!
- Embed your own versions of libraries if the ones bundled with Jahia don't fit your needs
- Expose new services or use services registered by other modules
- New tools to help with OSGi development
- And a lot more!
5 DIGITAL EXPERIENCE MANAGER OSGI ARCHITECTURE

In this section of the documentation we present high-level views of the architecture, flows and class loader hierarchies of Digital Experience Manager OSGi integration.

5.1 ARCHITECTURE OVERVIEW

Digital Experience Manager has integrated into version 7.0 an OSGi framework developed by the Apache Foundation called Felix. The goal of this integration was to offer the possibility for module developers to use OSGi to package their modules, therefore benefiting from the clean isolation that this technology offers. Over time, it is expected that more and more parts of the Digital Experience Manager core code will migrate into modules, therefore expanding the usage of OSGi bundles throughout the application. This process will take some time and will clearly happen over the course of several versions but should not affect module developers significantly as their bundles will already be packaged using OSGi bundles.
Inside the OSGi framework, all the modules are now deployed as bundles, but Digital Experience Manager also embeds a service that transforms a legacy WAR Digital Experience Manager module into an OSGi bundle at deployment time.

Since 7.2, all the OSGi bundles are managed by Karaf, an OSGi bundle container that is still working with Felix, but bring new functionalities as support of features or module management in cluster.

Other system bundles include the Felix Web Console and the Karaf Command Line shell, which are two low-level administration tools that are quite useful to access the OSGi internal runtime. The Digital Experience Manager administration is now implemented using OSGi modules and therefore now also lives inside the OSGi framework.

### 5.2 REQUEST FLOW

The request flow was modified with the introduction of the OSGi framework to address the following requirements:

- be able to serve resources directly from an OSGi bundle, compiling JSPs that are located inside bundles and serving images directly from the JAR
- integrating other OSGi Http services such as the Felix Web console

The graph below shows how the different software components are involved in request processing. Basically on the left side this is the legacy request flow processing before OSGi introduction and in the middle the Felix Http Bridge relays the requests through the OSGi Http service to integrate with bundles that register servlets inside the Http Service. The Digital Experience Manager OSGi extender is a bundle that listens to bundle events and is responsible for registering all the different resources of a bundle into the Http Service. On the right Digital Experience Manager also exposes another OSGi proxy servlet to expose services such as the Felix Web Console available at the Digital Experience Manager http://localhost:8080/tools URL.
5.3 CLASS LOADER GRAPH

In this section, we illustrate what the class-loading graph looks like in a typical Digital Experience Manager installation. By default, the server is integrated with the Apache Tomcat servlet container, and the graph below includes the Tomcat class loaders.
As we can see the Jahia Webapp is the main application class loader, but below we see that the graph no longer looks like a tree and OSGI class loaders may access different class loaders depending on their declared dependencies.

If you’re interested in learning more about the Tomcat class loader, you can find the related documentation here: http://tomcat.apache.org/tomcat-7.0-doc/class-loader-howto.html
6 BUILDING A NEW OSGI MODULE

You have multiple options to build a new OSGi module, and we will quickly present them here by starting with the easiest all the way to the most complex. Depending on your needs and skills, all of these might be interesting at some point in your projects.

6.1 USING THE DIGITAL EXPERIENCE MANAGER STUDIO TO CREATE A NEW PROJECT

The easiest way to create a new OSGi module is simply to create a new module using the Digital Experience Manager Studio that allows you to create and modify module or template projects directly from the Digital Experience Manager server development environment. It will by default use the new OSGi packaging and all you will have to do is simply customize it for your needs.

We will not give the details here on how to create a new project using the Studio, as this is already explained in detail in the Digital Experience Manager templating guide, available on the Jahia.com website.

6.2 USING A MAVEN ARCHETYPE

We provide a Maven archetype to get started quickly with a new project. The Maven Archetype is also used internally by the Digital Experience Manager Studio to initialize a new project. The steps below guide you through the process of creating a new project.

1. Create a new project using a Maven Archetype:

   mvn archetype:generate
   -DarchetypeCatalog=https://devtools.jahia.com/nexus/content/repositories/jahia-releases

2. Select the following archetype:

   3: https://devtools.jahia.com/nexus/content/repositories/jahia-releases ->
   org.jahia.archetypes:jahia-module-archetype (Archetype for creating a new module project to be run on a Digital Experience Manager server)

3. Enter project metadata and confirm

4. Change into project directory and build using:
mvn clean install

You can then open the Maven project in your favorite IDE and start building your Digital Experience Manager OSGi module.

### 6.3 FROM SCRATCH

First and foremost, if you're ok using the Digital Experience Manager Modules parent project, it will automatically configure both the Felix Maven Bundle Plugin and Jahia Maven Plugin to use defaults that make sense for most projects. To do so simply set as a parent to your Maven project:

```xml
<parent>
  <artifactId>jahia-modules</artifactId>
  <groupId>org.jahia.modules</groupId>
  <version>7.0.0.0</version>
  <relativePath/>
</parent>
```

If you prefer not to use Digital Experience Manager module parent project, you will have to setup the plugins yourself, as explained here.

To build an OSGi project, it is recommended to use the Felix Maven Bundle Plugin to help with the basic packaging. This is fairly easy to setup. First change the project's packaging to bundle:

```xml
<packaging>bundle</packaging>
```

Then configure and add the plugin to the project:

```xml
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-Name>${project.name}</Bundle-Name>
      <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
      <Bundle-Category>jahia-module</Bundle-Category>
      <Implementation-Title>${project.name}</Implementation-Title>
      <Implementation-Version>${project.version}</Implementation-Version>
      <Implementation-Vendor>${project.organization.name}</Implementation-Vendor>
    </instructions>
  </configuration>
</plugin>
```
This is the default minimal configuration for building a Jahia OSGi module bundle.

However, there are a few things that the Felix Bundle Maven plugin cannot do, it cannot scan in non-Java resources for package uses such as:

- JSPs
- Taglibs
- Groovy files
- Spring descriptors
- Content definitions
- Content import files

Fortunately, we provide a new goal in the Jahia Maven Plugin that will integrate with the Felix Bundle plugin that will scan all the standard Jahia module resources for you and build the required import package statements in the Felix Bundle plugin configuration.

Here is an example of setting up the Jahia Maven Plugin to scan for dependencies:

```xml
<plugin>
  <groupId>org.jahia.server</groupId>
  <artifactId>jahia-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>prepare-package-dependencies</id>
      <goals>
        <goal>dependencies</goal>
      </goals>
      <configuration>
        <contentDefinitionCapabilitiesActivated>true</contentDefinitionCapabilitiesActivated>
      </configuration>
      <phase>prepare-package</phase>
    </execution>
  </executions>
</plugin>
```

The plugin will generate the value for the property jahia.plugin.projectPackageImport that was already inserted in the Felix Maven Bundle Plugin configuration we had previously setup.

Now the project is ready for building, which you can simply do using:

`mvn clean install`
7 OSGI PACKAGE DEPENDENCIES AND EXPORTS

Before going any further, it is very important to understand what OSGi package dependencies are, how they work and how to use them properly.

The OSGi framework will only let you access a Java package located in another OSGi bundle if:

- It is exported by an OSGi bundle
- It is imported by your own bundle

OSGi imports and exports must be declared in the META-INF/MANIFEST.MF file. If an import or export is missing, or if versions of packages don't match, no access to the package will be allowed by the bundle's class loader.

Management of the package imports and exports is the main learning curve involved in learning to use OSGi. Fortunately, Digital Experience Manager offers tooling such as the Jahia Maven...
Plugin that helps generate dependencies for common module projects. There are also other OSGi plugins available on the Internet but usually the Felix Maven Bundle Plugin and the Jahia Maven plugin should be sufficient for most projects.
8 DEPLOYING AN OSGI MODULE IN DIGITAL EXPERIENCE MANAGER

Deploying an OSGi module is relatively straightforward. Once you have the generated JAR file (in a Maven project it is generated in the target subdirectory), you can simply deploy it to the digital-factory-data/modules directory of your Digital Experience Manager installation. This can be done whether the server is running or not, it will be picked up and deployed by the server. Every resource in the module (provided it deployed without any errors) will be immediately accessible, including any libraries that have been embedded in the package. If you have declared services in your bundle, they will also be available upon deployment.

8.1 DEPLOYING USING THE JAHIA MAVEN PLUGIN

During development, you will probably want to deploy and redeploy your module often, so using the jahia:deploy goal makes this a lot easier to do often. You just need to setup a profile with the location and type of your target server, such as in the following example:

```xml
<profile>
  <id>jahia-7.0-tomcat</id>
  <properties>
    <jahia.deploy.targetServerType>tomcat</jahia.deploy.targetServerType>
    <jahia.deploy.targetServerVersion>7</jahia.deploy.targetServerVersion>
    <jahia.deploy.war.dirName>ROOT</jahia.deploy.war.dirName>
    <jahia.deploy.war.contextPath>/</jahia.deploy.war.contextPath>
    <jahia.deploy.war.servletPath>/cms</jahia.deploy.war.servletPath>
    <jahia.debug.address>socket:hostname=localhost,port=8000</jahia.debug.address>
  </properties>
</profile>
```

You can then simply compile and deploy your module using the following command line:

```bash
mvn clean install jahia:deploy -P jahia-7.0-tomcat
```
8.2 UN-DEPLOYING/RE-DEPLOYING A MODULE

If you are using the Jahia Maven Plugin you can simply redeploy a module from the project using the jahia:deploy goal again. Upon deployment in the digital-factory-data/modules directory, Digital Experience Manager will automatically undeploy the old version and deploy the new one.

Alternatively, you can also use the administration’s manage modules UI to undeploy the module or remove the jar file from digital-factory-data/modules.
9  SERVICE SHARING BETWEEN MODULES

OSGi bundles may declare or implement OSGi services that will then be registered in a global framework service registry. Through the registry, other bundles may access the services to interact with them. This simply but powerful mechanism makes it possible to decouple bundles while still allowing strong interactions between them.

We will illustrate how to do this by using the example of how our external provider bundles are setup.

9.1  SPRING OSGI:SERVICE AND OSGI:REFERENCE TAGS

First we have a Digital Experience Manager module that provides an interface and implements it. We will use the Spring OSGi XML tags to register the service with OSGi’s service registry. This can be simply done by using the XML as in this example:

```xml
<bean id="ProviderInitializerService" class="org.jahia.modules.external.id.ExternalProviderInitializerServiceImpl">
  <property name="hibernateSessionFactory" ref="moduleSessionFactory"/>
  <property name="cacheProvider" ref="ehCacheProvider"/>
  <property name="extensionProvider" ref="DefaultJCRStoreProvider"/>
  <property name="overridableItemsForLocks">
    <list>
      <value>jmix:lockable.j:locktoken</value>
      <value>jmix:lockable.j:lockTypes</value>
      <value>mix:lockable.jcr:lockIsDeep</value>
      <value>mix:lockable.jcr:lockOwner</value>
    </list>
  </property>
</bean>

<osgi:service id="ExternalProviderInitializerService" ref="ProviderInitializerService" interface="org.jahia.modules.external.ExternalProviderInitializerService"/>
```

Note that for more complex service registrations, it might be a good idea to have two separate modules, one for the interfaces, and another for the implementation. This way you can deploy each separately, for example if the interfaces are stable but the implementation is still ongoing.

Once the service has been registered, in our second module, we can use a Spring OSGi service reference XML tag to access the registered service in the OSGi service registry:

```xml
<bean id="ProviderInitializerService" class="org.jahia.modules.external.id.ExternalProviderInitializerServiceImpl">
  <property name="hibernateSessionFactory" ref="moduleSessionFactory"/>
  <property name="cacheProvider" ref="ehCacheProvider"/>
  <property name="extensionProvider" ref="DefaultJCRStoreProvider"/>
  <property name="overridableItemsForLocks">
    <list>
      <value>jmix:lockable.j:locktoken</value>
      <value>jmix:lockable.j:lockTypes</value>
      <value>mix:lockable.jcr:lockIsDeep</value>
      <value>mix:lockable.jcr:lockOwner</value>
    </list>
  </property>
</bean>

<osgi:service id="ExternalProviderInitializerService" ref="ProviderInitializerService" interface="org.jahia.modules.external.ExternalProviderInitializerService"/>
```
But the work is not yet complete. An important step must still be completed: package export and import.

### 9.2 OSGI:LIST USAGE

You can define a list of services implementing a specified interface. Note that if the current bundle provides both the interface and an implementation, the implementation reference must be set as optional.

```xml
<osgi:list id="ModuleGlobalObjectList" interface="org.jahia.services.content.rules.ModuleGlobalObject" availability="optional">
    <osgi:listener ref="org.jahia.modules.default.ModuleRegistry" bind-method="osgiBind" unbind-method="osgiUnbind" />
</osgi:list>
```

### 9.3 EXPORT-PACKAGE INSTRUCTION

To make the service interface available to other bundles, we must export the package in which it is located. This is achieved by adding the following line to the Felix Maven Bundle Plugin configuration:

```xml
<Export-Package>org.jahia.modules.external</Export-Package>
```

### 9.4 IMPORT-PACKAGE INSTRUCTION

Now the last piece of the puzzle is the Import-Package instruction. In most cases this doesn't need to be manually configured as the Felix Maven Bundle Plugin will scan the module's code for any dependencies and pick up the service reference. Also, the Jahia Maven Plugin will also pick up the reference inside the Spring XML file.
9.5 OTHER SERVICE DECLARATION AND REFERENCING MECHANISMS

In OSGi there are a lot of different ways of registering and referencing services. In the above example, we have illustrated the most common one for Digital Experience Manager modules, but we will quickly list other alternatives:

- OSGi BluePrint: this is most recent service framework in OSGi. It is based on the Spring OSGi implementation but is now part of the standard. This is directly available in Digital Experience Manager and is usable out of the box.

- OSGi Declarative Services: this service framework is also part of the OSGi specification, but it is not part of the Digital Experience Manager out of the box configuration so you will need to deploy an implementation bundle if you want to use it. Declarative services are one of the most mature services frameworks in the OSGi and relatively simple, so it might be an interesting alternative to some.

- OSGi ServiceTracker: it is possible to manually register services using code, but OSGi provides a ServiceTracker class that makes it easier to track services as they appear and disappear at runtime due to OSGi highly dynamic nature. As this much lower level it is available in all OSGi frameworks and usable out of the box in Digital Experience Manager. It is however much more difficult to setup so it not recommended unless you have a good reason to use it (such as building your own extender for example).

- Manual service registration: this is always possible but really not recommended, unless you really know what you are doing. As services may appear or disappear any time, your implementation must handle this properly, and this can be quite complicated to handle.
10 DEPLOY-FREE CODING

Since the introduction of OSGi modules, Digital Experience Manager no longer “explodes” modules upon deployment. Before version 7, Digital Experience Manager would uncompress the contents of the WAR module file into a sub-directory of the /modules directory. This could lead to multiple problems notably with environments where writing on disk in the web application directory is not recommended (or not allowed), because it is not a good Java EE practice. Also this might tempt developers to modify exploded files directly during development in order to benefit from very quick develop-test-modify cycles, but they would then have the problem of having to sync back (manually) the changes to the module’s source code.

Digital Experience Manager 7 solves this by introducing Deploy-free coding. This new feature makes it possible, upon compilation and deployment of an OSGI module, to work directly from the source code without needing to redeploy.

10.1 HOW TO USE IT

1. Create your module’s source code project
2. Compile and deploy it to your Digital Experience Manager server that must be on the same file system as your project (initial deployment “links” source code with Jahia server). Note that the file system could be a network shared file system this is of course allowed.
3. Modify static resources directly in your source code; Jahia will pick up the changes directly from the source, no deployment needed.

10.2 HOW IT WORKS

When building the project, the Jahia Maven Plugin will add a special MANIFEST.MF header that points to the source code location with an absolute path. This is configured in the Felix Maven Bundle plugin by the following line:

```xml
<Jahia-Source-Folders>${project.basedir}</Jahia-Source-Folders>
```

Upon generation of the MANIFEST.MF inside the JAR, it will look something like this:
When a request comes in to a /modules/assets URL, Digital Experience Manager will use the Jahia-Source-Folders manifest header (if present) to look for the source code of the project. If it is present, it will try to access the requested resource directly from the source folder instead of using the contents of the bundle.

There are some limitations in this mechanism, not all files can be directly used, as some require compilation or internal caches may interfere with the proper detection of file modifications. Here is a list of resource types that are known to work with Deploy-free coding:

- JSP
- HTML
- Images
- CSS
- Javascript
- Velocity
- Other static file types such as documents, text files, etc....

And here is a list of resources types that don’t work:

- Groovy files (due to the internal Groovy engine class cache)
- Java classes
- Java Libraries

Of course, for the file types that don’t work with Deploy-free coding, it is still possible to hot-deploy them using OSGi bundle redeployment. So, for those file types the deployment life cycle is a little longer but still much faster than in previous versions of Digital Experience Manager.

### 10.3 RELEASING MODULES

When releasing modules, it is recommended that you remove the Jahia-Source-Folders configuration attribute from the Felix Maven Bundle plugin configuration. This is just to avoid
any potential lookups in case the source folders also exist on the server. It will also prevent unnecessary file lookups.
11 DIGITAL EXPERIENCE MANAGER OSGI TOOLING

11.1 JAHIA MAVEN PLUGIN

The Jahia Maven plugin has a few new goals to help you with your OSGi module projects. Here is a quick overview:

- **jahia:dependencies**: this goal will parse your project and its dependencies to generate a realistic list of imports for your module. It does this by scanning a lot of different resources to produce a list of package imports as well as content definition export and dependencies.

- **jahia:find-package-uses**: will use the BND tool from Peter Kriens internally to figure out where the package import came from, specifically which class coming from which Maven dependency that is generating this import. You could then use a tool such as JD-GUI (http://java.decompiler.free.fr/?q=jdgui) to open the JAR and decompile the class that is referencing the package to understand its uses, and if it is mandatory or not. One thing you could do once you understand the package use is marking it as an optional OSGi resolution.

- **jahia:find-packages**: will scan all the projects dependencies, including optional and provided ones, to find a package. So, you can use this if you suspect the package must be provided by a JAR in the project’s dependencies but have trouble finding it (for example because it is optional).

- **jahia:osgi-inspect**: will output a nicely formatted and easy to read view of the MANIFEST.MF headers and optionally also the packages contained inside a JAR (it doesn’t even have to be an OSGi bundle).

- **jahia:help**: to get inline information

11.1.1 **jahia:dependencies goal**

The jahia:dependencies goal helps you build the following OSGi MANIFEST headers:
- Import-Package (list of packages required by the OSGi bundle)
- Export-Package (list of packages exported by the OSGi bundle)
- Provide-Capability (list of capabilities provided by the OSGi bundle)
- Require-Capability (list of capabilities required by the OSGi bundle)

It is capable of scanning a lot of different resource types, to complement the class scanning that the Felix Maven Bundle Plugin already does, notably:

- .jsp files (both page import and taglib dependencies)
- .tld (tag library definition) files
- .cnd (content node definition) files
- .drl (Drools Rule) files
- .jbpm.xml (JBPM Workflow definition) files
- Spring context files
- Jackrabbit XML import files

The following graph explains how the jahia:dependencies goal integrates with the Felix Bundle Maven plugin to generate extended package imports.

Build steps:
1. The Jahia Maven plugin jahia:dependencies goal scans the project source code to detect all package references inside resources that are not supported by the Felix Bundle Maven Plugin class scanning
2. It generates a list of packages that is then made available to the Maven runtime in the following variable: ${jahia.plugin.projectPackageImport}
3. The Felix Bundle Maven plugin is configured to generate an OSGi bundle using its own configuration and also references the variable generated by the Jahia Maven plugin goal to generate the final OSGi manifest headers.

11.1.1.1 Package scanning

Most of the work the plugin does is scan different resource types to see which packages are used. For example, in the case of a JSP file it will scan the directives at the beginning of the file to see if any page import or taglibs are used. In the case of a page import it will simply retrieve the list of packages, but in the case of a tag library it will retrieve the corresponding TLD file, and scan its content to find the packages used inside the TLD file. This makes it easy for integrators to make sure they are importing the proper packages even when they use tag libraries, the plugin does all the work behind the scenes to make sure that the proper imports are generated. Also, it will also check for inconsistencies such as a missing dependency if a tag library is used in a JSP but missing from the project's Maven dependencies.

Once all the package scanning is completed, the plugin sets the following Maven project property: ${jahia.plugin.projectPackageImport}. This property contains a list of all the packages found in the project, and can then be used as import to the Import-Package instruction of the Maven Bundle plugin as in the example below:

```xml
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-Name>${project.name}</Bundle-Name>
      <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
      <Bundle-Category>jahia-module</Bundle-Category>
      <Implementation-Title>${project.name}</Implementation-Title>
    </instructions>
```

11.1.1.2 Content definition scanning

Much in the same way that we scan a project for package dependencies, the jahia:dependencies goal always scans all the resources for content node type definitions and references. Specifically, it scans the following file types:

- *.cnd: content definition files. Here it actually parses the files to extract all the new node type definitions as well as all the node types used either as super types, child node types, or even property types that can use a node type in their values.
• JCR import files (*.xml): the scanner uses the following XPath queries to retrieve content node type references: //@jcr:primaryType and //@jcr:mixinTypes
• The result of the scanning is stored in Maven project properties:
• ${jahia.plugin.providedNodeTypes}: a list of all content node type definitions defined in the project, formatted in OSGi Provide-Capability format.
• ${jahia.plugin.requiredNodeTypes}: a list of all required content node type definitions found in the project, formatted in OSGi Require-Capability format.

Here is an example of what these properties look like when generated for a Digital Experience Manager module:

| Provide-Capability: com.jahia.services.content; |
|--------------------------|--------------------------|
| nodetypes:List<String>="jmix:retrievableContent,jnt:contentRetrieval"  |

| Require-Capability: com.jahia.services.content; |
|--------------------------|--------------------------|
| filter:="(nodetypes=jmix:basicContent)",com.jahia.services.content;  |
| filter:="(nodetypes=jmix:editorialContent)",com.jahia.services.content;  |
| filter:="(nodetypes=jmix:list)",com.jahia.services.content;  |
| filter:="(nodetypes=jmix:queryContent)",com.jahia.services.content;  |
| filter:="(nodetypes=jmix:renderableList)",com.jahia.services.content;  |
| filter:="(nodetypes=jnt:content)",com.jahia.services.content;  |
| filter:="(nodetypes=jnt:page)",com.jahia.services.content;  |
| filter:="(nodetypes=mix:title)" |

In the above example we see that the project defined the new node type definitions jmix:retrievableContent and jnt:contentRetrieval and that it needs content node definitions such as jmix:basicContent in order to work correctly. If these requirements are not fulfilled the module will not be able to start.

We can therefore integrate the capability generation with the Maven Bundle plugin by expanding on the example provided previously in the package scanning example, as seen here:

```xml
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-Name>${project.name}</Bundle-Name>
      <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
      <Bundle-Category>jahia-module</Bundle-Category>
      <Implementation-Title>${project.name}</Implementation-Title>
    </instructions>
  </configuration>
</plugin>
```
11.1.1.3 Troubleshooting dependencies

Sometimes understanding why a dependency was generated can be tedious, so fortunately launching the jahia:dependencies goal in debug mode will generate a lot of interesting logging information on where it has extracted a dependency. You can activate the debug mode simply by using the -X option on the command line as in the following example:

mvn -X jahia:dependencies
11.1.1.4 Deactivating/controlling content definition scanning

If for some reason you need to control the content definition scanning, there are three possibilities:

- artifactExcludes property: deactivate it for specific files by using the artifactExcludes configuration option

  ```xml
  <artifactExcludes>
  <exclude>org.jahia.modules:*</exclude>
  <exclude>org.jahia.templates:*</exclude>
  <exclude>org.jahia.test:*</exclude>
  <exclude>*.jahia.modules</exclude>
  </artifactExcludes>
  ```

- scanDirectories property: using the scanDirectory you can specify a list of directories to scan. The default value is:

  ```java
  scanDirectories.add(project.getBasedir() + "\src\main\resources");
  scanDirectories.add(project.getBasedir() + "\src\main\import");
  scanDirectories.add(project.getBasedir() + "\src\main\webapp");
  ```

- excludeFromDirectoryScan : you can specify which files to exclude from the directory scan

  ```xml
  <excludeFromDirectoryScan>
  <exclude>imports/import.xml</exclude>
  <exclude>imports/importIndexOptionNodes.xml</exclude>
  </excludeFromDirectoryScan>
  ```

11.1.2 jahia:find-package-uses goal

Our Jahia Maven plugin has new tools to help with resolving dependency problems when generating bundles using the Maven bundle plugin. Often this plugin will generate imports for packages that you might not know where they are coming from. It is usually due to some third-party library that has a dependency on a package either by importing it, or by using it in a Class.forName() reflection API call (BND actually finds these).

Here is an example with our Digital Experience Manager test module project (jahia/test/jahia-test-module). We noticed on deployment that it was generating two strange imports: kaffe.util and weblogic. We can use the new find-packages-uses goal to figure out where they are used:
mvn jahia:find-package-uses -DpackageNames=weblogic,kaffe.util

This will generate the following result:

```
[INFO]-----------------------------------------------------------------------------------
[INFO] SEARCH RESULTS
[INFO]-----------------------------------------------------------------------------------
[INFO] Found package weblogic used in class
[INFO] Found package kaffe.util used in class org.apache.tools.ant.util.JavaEnvUtils
  from trail org.jahia.test:jahia-test-module:bundle:7.0.0.0-SNAPSHOT --> org.apache.ant:ant:jar:1.8.2
```

We can then go into the JARs for these two dependencies and decompile the classes using JD-GUI to understand the usage. We can then mark the dependencies as optional in the Maven Bundle Plugin configuration as in the following example:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-Name>${project.name}</Bundle-Name>
      <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
      <Bundle-Category>jahia-module</Bundle-Category>
      <Implementation-Title>${project.name}</Implementation-Title>
      <Implementation-Version>${project.version}</Implementation-Version>
      <Implementation-Vendor>${project.organization.name}</Implementation-Vendor>
      <Implementation-URL>${project.organization.url}</Implementation-URL>
      <Specification-Title>${project.name}</Specification-Title>
      <Specification-Version>${project.version}</Specification-Version>
      <Specification-Vendor>${project.organization.name}</Specification-Vendor>
    </instructions>
  </configuration>
</plugin>
```
We could use the findPackages goal if we suspected that a dependency included some specific packages, as in the following example:

```
mvn jahia:find-packages -DpackageNames=weblogic,kaffe.util
```

This generates the following result:

```
[INFO]--------------------------------------------------------------------------------
[INFO] SEARCH RESULTS
[INFO] --------------------------------------------------------------------------------
[WARNING] Couldn't find weblogic anywhere !
[WARNING] Couldn't find kaffe.util anywhere !
```

This confirms we were right in marking the dependencies as optional, as we don’t have them in our project dependencies but they might be provided at runtime by another OSGi bundles, but they are also not required at runtime.
11.1.4 jahia:osgi-inspect goal

This new goal makes easy to dump the headers of an OSGi bundle JAR file. It will also work with normal JARs though, so it can be useful to check if the headers were properly generated. By default, it will look for the project’s generated artifact (must have been previously generated). Here is an example that will print out the contents of the project’s artifact JAR:

```
mvn jahia:osgi-inspect
```

If you prefer to pass a parameter to specify which JARs should be inspected, you can simply use the jarBundles parameter as in the following example:

```
mvn jahia:osgi-inspect -DjarBundles=target/project-1.0-SNAPSHOT.jar,target/project-1.0-SNAPSHOT-sources.jar
```

This will print out the headers for both JAR files.

11.2 FELIX MAVEN BUNDLE PLUGIN

(Note: the following section is a reproduction of the online Felix Maven Bundle plugin documentation, available in full here: http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html)

This plugin for Maven 2 is based on the BND tool from Peter Kriens. The way BND works is by treating your project as a big collection of classes (e.g., project code, dependencies, and the class path). The way you create a bundle with BND is to tell it the content of the bundle's JAR file as a subset of the available classes. This plugin wraps BND to make it work specifically with the Maven 2 project structure and to provide it with reasonable default behavior for Maven 2 projects.

11.2.1 More information about the plugin

You can find more documentation and information about the plugin at the official site:


As well as the Maven standard plugin documentation available here:

11.3 THE MANAGE MODULES ADMINISTRATION UI

The “Manage module” server administration UI in Digital Experience Manager is a user-friendly yet powerful user interface to manage the modules deployment, state and view more details about its dependencies and other meta-information. We won’t go into many details here as this UI is covered in other Digital Experience Manager documents such as the administrator’s guide or the templating development guide.
11.4 THE APACHE FELIX WEB CONSOLE

The Apache Felix Web Console is an OSGi tool that is integrated into Digital Experience Manager. To access it, open a browser at the URL: http://localhost:8080/tools, enter the requested login information and then click on “OSGi console”.

The Web Console is a powerful tool to see the internals of the Digital Experience Manager OSGi framework. As this is an integrated external tool, to learn more please go to the Apache Felix Web Console project website: http://felix.apache.org/site/apache-felix-web-console.html

11.5 KARAF CONSOLE

Integrated directly into Digital Experience Manager 7.2, enabled only for localhost by default for security reasons, the Karaf command line shell is incredibly useful to diagnose and query the OSGi framework in the cases where the Web Console might not yet be available (during Digital Experience Manager startup for example), or simply when users prefer a command line interface.
11.5.1 Command line configuration

By default, the shell is only available for localhost (127.0.0.1) on port 8101. To change this default settings, while the Digital Experience Manager server is not running, open up the Digital Experience Manager configuration file located at WEB-INF/etc/config/jahia.properties, uncomment the following line and set its value to something like this:

```
# The following setting is used to change the port which the
# Apache Karaf OSGi command line shell will listen to for SSH
# connections. Set it to a negative value to disable this feature
# entirely.
kafar.remoteShell.port = 8101
# The bind address for the SSH shell. '127.0.0.1' means the SSH shell
# will only allow local connections to be established. You may want to
# define here an dedicated IP address, the console will bind to, or
# a '0.0.0.0' which will mean it will be bound to all available
# network interfaces.
kafar.remoteShell.host = 127.0.0.1
```

You can find more information about Karaf configuration here:

https://karaf.apache.org/manual/latest/#_sshd_server

Configuration files are available here:

digital-factory-data/karaf/etc

11.5.2 Accessing the shell

The Karaf shell is accessible via SSH:

```
ssh -p 8101 jahia@localhost
```

To connect, use the same credentials as in the tools (jahia/password by default). The first connection will ask you to allow the host as a known secured host, then you should be greeted with a screen that looks something like this:

```
~ $ ssh -p 8101 jahia@localhost
The authenticity of host 'localhost':8101 ([127.0.0.1]:8101) can't be established.
RSA key fingerprint is SHA256:AhtzuXQ90I517qhXZoTtLS0xw2S47aRhqYvwBZxh0tA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost':8101 (RSA) to the list of known hosts.
Password authentication
```
You can then type in command such as:

jahia:modules

This will generate an output that looks something like this:

<table>
<thead>
<tr>
<th>Id</th>
<th>State</th>
<th>Symbolic-Name</th>
<th>Version</th>
<th>Depends on</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>139</td>
<td>Installed</td>
<td>calendar</td>
<td>2.0.3</td>
<td>[default]</td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>Installed</td>
<td>bootstrap-components</td>
<td>3.0.2</td>
<td>[default, bootstrap, search,tabularList]</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>Started</td>
<td>dx-base-demo-core</td>
<td>1.0.1-SNAPSHOT</td>
<td>[default, event, rating, topstories]</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Started</td>
<td>translateworkflow</td>
<td>2.0.4</td>
<td>[default]</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>Started</td>
<td>twitter</td>
<td>2.0.2</td>
<td>[default]</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Started</td>
<td>document-thumbnails</td>
<td>2.0.6-SNAPSHOT</td>
<td>[document-management-api, default]</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>Started</td>
<td>sitemap</td>
<td>2.0.4</td>
<td>[default]</td>
<td></td>
</tr>
</tbody>
</table>

The jahia:modules command gives precise state information on the deployed Digital Experience Manager modules.

### 11.5.3 JCR Commands

We also provide from the console several commands that let you browse the JCR from the console.

The prefix is jcr

**Available commands are:**

- `jcr:cd` : to change the current path
- `jcr:l` : to list the nodes under the current path
- `jcr:prop-delete` : to delete a property
- `jcr:prop-get` : to read a property
- jcr:prop-set : to set a property
- jcr:query : to perform a query
- jcr:workspace : to read current or switch workspace

Example:

jahia@dx()> jcr:cd users/root/
/users/root
jahia@dx()> jcr:l

<table>
<thead>
<tr>
<th>Name</th>
<th>UUID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>files</td>
<td>5732847d-41e0-4eff-a339-631fcd61916f</td>
<td>jnt:folder</td>
</tr>
<tr>
<td>portlets</td>
<td>a09c11e2-8f8a-4176-bce2-3bf066ee80f1</td>
<td>jnt:portletFolder</td>
</tr>
<tr>
<td>contents</td>
<td>3a4c2fab-c365-4a40-b909-9cdf753cda69</td>
<td>jnt:contentFolder</td>
</tr>
<tr>
<td>preferences</td>
<td>cdf802b0-53cf-48a1-98a4-8954331f46e9</td>
<td>jnt:preferences</td>
</tr>
<tr>
<td>j:acl</td>
<td>c48d8201-cbf2-4e44-8952-3f16d1266d0b</td>
<td>jnt:acl</td>
</tr>
<tr>
<td>j:addresses</td>
<td>89a98897-8ab0-47db-8c57-b9c1f6e5cb43</td>
<td>jnt:addresses</td>
</tr>
<tr>
<td>j:phones</td>
<td>03643752-ff35-4178-abc3-24a8f0ed842e</td>
<td>jnt:phoneNumbers</td>
</tr>
<tr>
<td>passwordHistory</td>
<td>40231a5b-b9e1-4f3f-955e-d204e81ac1ba</td>
<td>jnt:passwordHistory</td>
</tr>
<tr>
<td>workflowTasks</td>
<td>e7975eed-8def-4850-ba46-470cba7a5080</td>
<td>jnt:tasks</td>
</tr>
</tbody>
</table>
12 USING LIBRARIES IN AN OSGI MODULE

When building a Digital Experience Manager OSGi module, very quickly the need to integrate third party libraries comes. To integrate these into your project there are different possibilities, which we will now present:

- Find an existing OSGi bundle you can simply deploy and use as a dependency
- Embed them inside your bundle (either as a JAR inside a JAR or by inlining them). Not optimal in terms of re-use but quite nice in terms of isolation and easy to do. Be careful with Embed-Transitive though!
- Transform a non-OSGi library into an OSGi compliant bundles (multiple options, from dynamic transforming to new projects)
- Deploy them at the framework level, exposing them to the whole OSGi runtime. This is not recommended unless you have a very specific reason to do so and know what you are doing. In any case, you should always specify the version of all the packages you are exporting.

We will now go into more details of each option, in order from most to less recommended.

12.1 FINDING OSGI BUNDLES

Finding ready-made OSGi bundles can be a bit of a challenge for the OSGi newcomer. So we will quickly give a few tips on where to find good bundles.

The Apache software foundation has made quite a large effort to update most of its projects to now offer either separate OSGi bundle releases, or simply to add OSGi metadata to existing release JARs. So, for example the Apache Commons project JARs are almost all OSGi bundles.

The Apache ServiceMix project also maintains “ports” of existing common libraries as OSGi bundles, and of course encourages other to contribute to the already quite large collection of ported libraries. You can find the projects and the releases here:
http://servicemix.apache.org/developers/source/bundles-source.html

SpringSource also has a bundle repository here (http://ebr.springsource.com/repository/app/), but be careful before using a JAR from that repository as it is no longer actively maintained and
even worse some bundles contain metadata errors! So, if you have a choice between an Apache ServiceMix bundle and a SpringSource EBR bundle, always use the ServiceMix version.

Another possibility is to talk to library authors and either help them or convince them to release OSGi bundles of their projects (the effort is usually minimal, but see section 12.3 on how to meet them half-way).

### 12.2 EMBEDDING NON-OSGI LIBRARIES

A quick way to integrate non-OSGi libraries with your module is to directly embed them inside the module’s JAR. This technique has tradeoffs though.

**Advantages:**

- Good isolation
- Easy to do
- Complete control over deployment

**Disadvantages:**

- Can lead to duplicate deployment of libraries if multiple modules embed the same library
- Not the preferred OSGi way of deploying code and dependencies

Despite the disadvantages, especially in projects migrating from older versions of Digital Experience Manager, embedding the libraries is usually the best way to migrate to OSGi. Removing the embedded library and building more modular modules could then be done in a second phase once everything is up and running in the first phase.

Embedding JARs is done through the usage of the Felix Maven Bundle Plugin configuration. Here is the syntax of the main configuration property:

```xml
<Embed-Dependency>dependencies</Embed-Dependency>
```

where:

- `dependencies ::= clause (',' clause)*`
- `clause ::= MATCH (',' attr '=' MATCH | ';inline=' inline)`
attr ::= 'groupId' | 'artifactId' | 'version' | 'scope' | 'type' | 'classifier' | 'optional'
inline ::= 'true' | 'false' | PATH ( '|' PATH ) *
MATCH ::= <globbed regular expression>
PATH ::= <Ant-style path expression>

The plugin uses the `<Embed-Dependency>` instruction to transform the project dependencies into `<Include-Resource>` and `<Bundle-ClassPath>` clauses, which are then appended to the current set of instructions and passed onto BND. If you want the embedded dependencies to be at the start or middle of `<Include-Resource>` or `<Bundle-ClassPath>` then you can use `{maven-dependencies}`, which will automatically expand to the relevant clauses.

The MATCH section accepts alternatives, separated by |, and can be negated by using ! at the beginning of the MATCH. Use * to represent zero or more unknown characters and ? to represent zero or one character. You can also use standard Java regex constructs. There is no need to escape the . character inside MATCH. The first MATCH in a clause will filter against the artifactId.

Examples:

```xml
<!-- embed all compile and runtime scope dependencies -->
<Embed-Dependency>*</scope=compile|runtime</Embed-Dependency>

<!-- embed any dependencies with artifactId junit and scope runtime -->
<Embed-Dependency>junit;scope=runtime</Embed-Dependency>

<!-- inline all non-pom dependencies, except those with scope runtime -->
<Embed-Dependency>*</scope=!runtime;type=!pom;inline=true</Embed-Dependency>

<!-- embed all compile and runtime scope dependencies, except those with artifactIds in the given list -->
<Embed-Dependency>*</scope=compile|runtime;inline=false;artifactId=!cli|lang|runtime|tidy|jsch</Embed-Dependency>

<!-- inline contents of selected folders from all dependencies -->
<Embed-Dependency>*</inline=images/**|icons/**</Embed-Dependency>
```

### 12.3 TRANSFORMING NON-OSGI LIBRARIES INTO BUNDLES

If a library you want to use is not available as an OSGi bundle, it is possible to transform it into an OSGi bundle. This can be done in two ways:
- Statically using a wrapper project that will generate a new JAR containing the library with proper OSGi metadata
- Dynamically by using an OSGi bundle that will perform the wrapping at runtime

We will now go into more detail of the two possibilities, as well as explain the advantages and disadvantages of both methods.

12.3.1 Static transformation

The static transformation of OSGi bundles is simply done by creating a new project that will wrap the existing library, usually using the Felix Maven Bundle plugin’s embedding feature.

Here is an example pom.xml for wrapping the mysql-connector JAR, taken from the SpringSource OSGi example (https://svn.code.sf.net/p/springframework/svn/osgi-repo/trunk/pom.xml). Note that this example is originally two Maven projects, but they were merged here for the sake of simplicity:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project
   xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

   <parent>
      <artifactId>osgi-repo</artifactId>
      <groupId>org.springframework.osgi</groupId>
      <version>1.0-SNAPSHOT</version>
   </parent>

   <modelVersion>4.0.0</modelVersion>
   <groupId>org.springframework.osgi</groupId>
   <artifactId>mysql-connector-java.osgi</artifactId>
   <packaging>bundle</packaging>
   <version>3.1.14-SNAPSHOT</version>
   <name>mysql-connector-java (OSGi version)</name>

   <properties>
      <unpack.version>3.1.14</unpack.version>
      <export.packages>
         com.mysql*;version=${unpack.version},
         org.gjt*;version=${unpack.version}
      </export.packages>
      <import.packages>
         com.mchange*;resolution=optional,
         org.apache.log4j;resolution=optional,
         org.jboss*;resolution=optional,
      </import.packages>

```
javax.naming*;resolution:=optional,
javax.net*;resolution:=optional,

</import.packages>
</properties>

<dependencies>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>3.1.14</version>
<scope>provided</scope>
</dependency>
</dependencies>

<build>

<plugins>
<!-- OSGi Felix bundle plugin -->
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<version>1.2.0</version>
<configuration>
<unpackBundle>${unpack.bundle}</unpackBundle>
<obrRepository>NONE</obrRepository>

</configuration>
</plugin>

<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-deploy-plugin</artifactId>
<version>2.4</version>
</plugin>
</plugins>

</build>
</project>
Advantages:

- Full control over the embedding process
- Proper repackaging of non-OSGi

Disadvantages:

- Requires more initial work to repackage library

12.3.2 Dynamic transformation

The dynamic transformation is a possibility offered by some OSGi bundles such as the Pax URL project. It is described at the following URL:

https://ops4j1.jira.com/wiki/display/paxurl/Wrap+Protocol

This technology makes it possibly to simply deploy a non-OSGi JAR using a specific URL format and the Pax URL project will then internally execute BND to process the JAR and generate on-the-fly an OSGi bundle and deploy it.

Advantages:

- Easier to quickly wrap a library
- Pax URL wrap protocol offers some controls over the wrapping process

Disadvantages:

- There might be some cases where the configuration becomes so large that using URLs is no longer possible (or viable)
- Some libraries might do strange things that might not be possible to wrap dynamically

12.4 DEPLOY NON-OSGI LIBRARIES AT THE FRAMEWORK LEVEL (OR ABOVE)

The last solution for including non-OSGi libraries is to deploy them at the framework level, which means they will be exposed to all the OSGi bundles in the runtime. This solution is quite extreme and should usually be chosen only as a last resort, as it can have many consequences.
What is extremely important if this solution is selected is to make sure that the library is exposed with very precise version numbers, so that bundles still have the possibility to deploy other versions of the same library if they choose to.

For more information on how to perform this, see the example at the end of this document: “Configuring a module that extends the system”.
13 OSGI AND JAVA PROFILING

As the OSGi framework consists of many class loaders interacting with each other, the usage of a profiling tool, especially one that uses instrumentation such as YourKit, often requires some custom framework settings.

Here is an example on how to configure YourKit for profiling Digital Experience Manager. In the WEB-INF/etc/config/felix-framework.properties, make sure you have the YourKit packages declared in the bootdelegation property:

```properties
```

If using another profiler, you might have to also add the profiler packages to the boot delegation property. By default, Digital Experience Manager comes pre-configured for profiling with YourKit.

You can find an interesting blog entry that gives example for different profilers here:

http://blog.knowhowlab.org/2010/03/osgi-tips-osgi-profiling-yourkit.html
14 CONVERTING AN EXISTING MODULE INTO AN OSGI MODULE

14.1 DIGITAL EXPERIENCE MANAGER'S AUTOMATIC MODULE TRANSFORMATION

To ease the migration from a non-OSGi Digital Experience Manager module to a fully OSGi module, Digital Experience Manager is capable of dynamically transforming a Digital Experience Manager WAR module into an OSGi module. It will attempt to resolve dependencies by scanning the contents of the module to determine which other modules or libraries are needed, and in general, on simple modules, this should work seamlessly. However, if the modules are doing more advanced integrations with Digital Experience Manager, the automatic transformation might not be able to pick up the most advanced features and converting the module to an OSGi version manually might be required (for example if custom rule consequences were implemented, those will require some Java code change as the API has changed in Digital Experience Manager 7). Also, using a native OSGi bundle packaging will open the possibility to expose services and integrate with other OSGi frameworks that might be useful to module developers, so in general it is recommended to migrate manually at some point.

All new modules must however be developed as OSGi bundles, as the transformation system is mostly a migration tool, and might be deprecated in future versions of Digital Experience Manager.

14.1.1 Controlling automatically generated package imports and exports

In the file felix-framework.properties, the following settings allow you to control the generation of package imports and export for Digital Experience Manager's module transformation engine:

- org.jahia.bundles.url.jahiawar.importedPackages: is a list of package imports that will be added to the automatically generated list of package imports. The format of the list is the following:

  MODULE_ROOT_FOLDER=PACKAGE_NAME1,
MODULE_ROOT_FOLDER=PACKAGE_NAME2
Where MODULE_ROOT_FOLDER is either the module's root folder or "*" to indicate that
the package import applies to all modules.

- org.jahia.bundles.url.jahiawar.excludedImportPackages: uses the same format of entries
  but controls the exclusion of automatically generated package imports
- org.jahia.bundles.url.jahiawar.excludedExcludePackages: uses the same format of entries
  but controls the exclusion of automatically generated package exports

14.2 SOURCE CODE CHANGES

14.2.1 Using the Jahia Maven Plugin
The Jahia Maven Plugin has a new goal to help with the conversion to OSGi: jahia:convert-to-
 osgi. This goal is designed to be used on a WAR packaging Digital Experience Manager module
and will perform some of the project transformation described here below, notably:

- change the module’s packaging
- move the directories
- remove the no longer needed WEB-INF/web.xml file
- parse the project source code and list the required modifications for updating the project
  (there is even a command line parameter to attempt to perform the source code
  modifications automatically)

Using this goal does not mean that you won't have to change anything in your project once it
has been executed. Please review the following sections to see what changes you might have to
perform manually to fully migrate your module to an OSGi package.

Here is the procedure to use the jahia:convert-to-osgi goal:

1. If you're using the jahia-modules artifact as a parent project, you need to change your
   Maven POM from:

   ```xml
   <parent>
     <artifactId>jahia-modules</artifactId>
     <groupId>org.jahia.modules</groupId>
   ```
2. You can now launch the goal using:

```shell
mvn jahia:convert-to-osgi
```

The goal will update your project but also scan the source code for any potential issues that must be resolved before it’s usable on Digital Experience Manager 7.

### 14.2.2 Manually upgrading the source code

#### 14.2.2.1 Changing the Maven POM

The modifications to your Maven project are quite straightforward. Basically you need to:

1. If you’re using the jahia-modules parent project, you need to update the version from 6.6.0.0 to 7.0.0.0
2. Remove any old repository such as:

```xml
<repositories>
  <repository>
    <id>jahiaRepository</id>
    <name>Jahia's Maven Repository</name>
    <url>http://maven.jahia.org/maven2</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>jahiaRepository</id>
    <name>Jahia's Maven Repository</name>
    <url>http://maven.jahia.org/maven2</url>
  </pluginRepository>
</pluginRepositories>
```

and replace them with the following:
<repositories>
  <repository>
    <id>jahia-public</id>
    <name>Jahia Public Repository</name>
    <url>https://devtools.jahia.com/nexus/content/groups/public</url>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

3. Change the <packaging> from war to bundle
4. Remove the Maven WAR plugin configuration
5. Setup the Jahia Maven Plugin with either a default configuration if you inheriting from Jahia’s modules parent project (this is the default behavior of the jahia:convert-to-osgi goal) or with your own custom configuration (see section “6.3 From scratch” for more information on how to do this)
6. Setup the Felix Maven Bundle plugin to re-use the output of the Jahia Maven plugin and adjust any parameter (again if you used the jahia:convert-to-osgi goal this will be inherited from the parent project).

If you need to setup custom configurations, please refer to section 6.3 From scratch for more information on the configurations and the settings.

14.2.2.2 Moving directories

As there is no more web application packaging, the contents of the src/main/webapp directory must be moved into the src/main/resources directory. This is the only main structural change that is required on the module’s structure apart from the Maven POM modifications.

14.2.3 Rules

Most rules should be directly compatible. If they aren’t, the jahia:convert-to-osgi goal will list the eventual problems and you should then perform the indicated modifications.
14.2.4 Hibernate

If your module requires a Hibernate session and was previously using the Digital Experience Manager default session Factory, it will have to be modified to use a new “moduleSessionFactory”. Here is an example from the Digital Experience Manager remote publication module. Before OSGi migration it looked like this:

```xml
<bean id="RemotePublicationLogService"
   class="org.jahia.modules.remotepublish.LogService">
   <property name="sessionFactoryBean" ref="sessionFactory"/>
   <property name="journalEntriesReadChunkSize" value="${jahia.remotePublicationService.journalEntriesReadChunkSize:100}"/>
   <property name="resetObjectOutputStreamAfterEachBytes" value="${jahia.remotePublicationService.resetObjectOutputStreamAfterEachBytes:20971520}"/>
</bean>
```

After the modification, the file looks like this:

```xml
<bean id="RemotePublicationLogService"
   class="org.jahia.modules.remotepublish.LogService">
   <property name="sessionFactoryBean" ref="moduleSessionFactory"/>
   <property name="journalEntriesReadChunkSize" value="${jahia.remotePublicationService.journalEntriesReadChunkSize:100}"/>
   <property name="resetObjectOutputStreamAfterEachBytes" value="${jahia.remotePublicationService.resetObjectOutputStreamAfterEachBytes:20971520}"/>
</bean>
```

As you can see this change is relatively trivial and easy to do.

14.2.5 Workflow

As in Digital Experience Manager 7, the workflow back-end was upgraded from jBPM 4 to jBPM 6, the workflow definition files must be modified to use the BPMN 2.0 specification. Tools such as the Eclipse jBPM plugin (https://www.jboss.org/jbpm/components/eclipse-plugin.html) or the jBPM Designer (https://www.jboss.org/jbpm/components/designer) can be used to graphically rebuild the workflow definition files.

Workflow custom tasks will also need minor modifications since the API has changed, but usually these changes can be done quite quickly.
14.2.6 Tag libraries

If your module used or included tag libraries, you will have to modify their usage the following way:

- If it used an external tag library, check if there is an OSGi bundle version of the tag library. If there is you’re in luck you can then simply add a dependency to it and deploy it as a separate bundle. If not, you’ll have to use one of the solutions proposed in “12 Using libraries in an OSGi module” to use the tag library.

- If you were providing your own library of tags and they were packaged inside a module that also offers other services (views, etc.), it is highly recommended to build a separate module (or simply an OSGi bundle, that way you could even re-use it in other environments) just for the tag library. Then in your modules you can easily add the tag library as a dependency. It is not recommended to keep the “one module for everything” philosophy for modules including tag libraries (although it should work).
15 TROUBLESHOOTING

This section aims to offer a few troubleshooting tips, mostly based on Jahia’s own experience of migrating our projects to OSGi.

15.1 COMMON OSGI-RELATED ERRORS AND THEIR SOLUTIONS

15.1.1 Missing dependencies when deploying OSGi module

Missing dependencies are one of the most common problems a newcomer to OSGi will face, especially when migrating an existing non-OSGi module. Here's an example of a log entry that illustrates the problem of a missing dependency:

```
org.osgi.framework.BundleException: Unresolved constraint in bundle filesync [118]: Unable to resolve 118.0: missing requirement [118.0] osgi.wiring.package; (osgi.wiring.package=com.ibm.uvm.tools)
at org.apache.felix.framework.Felix.resolveBundleRevision(Felix.java:3974)
at org.apache.felix.framework.Felix.startBundle(Felix.java:2037)
at org.apache.felix.framework.BundleImpl.start(BundleImpl.java:955)
at org.apache.felix.framework.BundleImpl.start(BundleImpl.java:942)
at org.jahia.bundles.extender.jahiamodules.Activator$BundleStarter.startAllBundles(Activator.java:870)
at org.jahia.bundles.extender.jahiamodules.Activator$BundleStarter.frameworkEvent(Activator.java:851)
at org.apache.felix.framework.util.EventDispatcher.fireEventImmediately(EventDispatcher.java:785)
at org.apache.felix.framework.util.EventDispatcher.run(EventDispatcher.java:1088)
at org.apache.felix.framework.util.EventDispatcher.access$000(EventDispatcher.java:54)
at org.apache.felix.framework.util.EventDispatcher$1.run(EventDispatcher.java:101)
at java.lang.Thread.run(Thread.java:744)
```

If a module doesn’t start because of missing dependencies, the easiest way to diagnose this is to use the OSGi Web Console and look at the bundle dependencies detail view (by expanding the bundle details by clicking on the little arrow at the left of the bundle’s name). It should look something like this:
As you can see in the above screenshot, the missing dependencies are highlighted in red. This means that the highlighted packages could not be resolved from another other bundle exporting packages or from the system (framework) bundle.

One quick and dirty way to solve this would be to simply mark all the unresolved packages as optional. This way the module will start, but it might not work, as some of these dependencies might actually be required for proper operation of the deployed module. You could use this as an integration phase, but proper package-use analysis is required to make sure no functional regressions will occur when resolving dependencies.

Adding the package dependencies as optional would require changing the Import-Package instruction of the Felix Maven bundle plugin to something like this:

```xml
<Import-Package>
  com.ibm.uvm.tools;resolution:=optional,
  com.jayway.jsonpath;resolution:=optional,
  com.jcraft.jsch;resolution:=optional,
  com.sun.jdmk.comm;resolution:=optional,
  com.sun.tools.javac;resolution:=optional,
  javax.jmdns;resolution:=optional,
  javax.jms;resolution:=optional,
  joptsimple;resolution:=optional,
  junit.framework;resolution:=optional,
  kaffe.rmi.rmic;resolution:=optional,
  org.apache.avalon.framework.logger;resolution:=optional,
  org.apache.bsf;resolution:=optional,
  org.apache.env;resolution:=optional,
  ...
$({jahia.plugin.projectPackageImport}),
*/Import-Package>
```
To help understand where a package is used, you can use the Jahia Maven find-package-uses goal as in the following example:

```shell
mvn jahia:find-package-uses -DpackageNames=com.ibm.uvm.tools
```

You will get a result similar to this:

```
[INFO] SEARCH RESULTS SUMMARY
[INFO] Package com.ibm.uvm.tools used in classes:
 [INFO] org.apache.log4j.spi.LocationInfo (/Users/loom/.m2/repository/log4j/log4j/1.2.17/log4j-1.2.17.jar)
```

We can therefore understand that the com.ibm.uvm.tools package is used inside a Log4J class. If we really wanted to dig deep, we could look at the source code for the Log4J project since we now have the proper information for the library dependency as well as the version of the code.

We have found the source file in the public SVN repository here:


```java
87 // Check if we are running in IBM's visual age.
88 static boolean inVisualAge = false;
89 static {
90     try {
91         inVisualAge = Class.forName("com.ibm.uvm.tools.DebugSupport") != null;
92         LogLog.debug("Detected IBM VisualAge environment.");
93     } catch(Throwable e) {
94         // nothing to do
95     }
```

As we can see this package is accessed to check the existence of a class, so it is probably safe to mark it as optional in our dependencies. This concludes the dependency analysis for the first
package, and it should be repeated for each unresolved dependency. If a dependency is resolved to be required, we should then add a bundle that provides the dependency (see section 12 “Using libraries in an OSGi module” for more information on how to achieve this).

15.1.2 JSP compilation error due to missing MANIFEST dependencies

"PWC6197: An error occurred at line: 222 in the jsp file: /jnt_petition/html/petition.full.jsp
PWC6199: Generated servlet error:
ServicesRegistry cannot be resolved

org.apache.jasper.JasperException: PWC6033: Unable to compile class for JSP
PWC6199: Generated servlet error:
Only a type can be imported. org.jahia.registries.ServicesRegistry resolves to a package
PWC6197: An error occurred at line: 222 in the jsp file: /jnt_petition/html/petition.full.jsp
PWC6199: Generated servlet error:
ServicesRegistry cannot be resolved"

This may happen if no entry in the MANIFEST.MF was added for the org.jahia.registries package. The cause may either be:

- The Jahia Maven Plugin was not configured with the jahia:dependencies goal
- The module is a legacy WAR format and automatic transformation is missing the dependency

15.1.2.1 Possible solutions

- Modify the project to become an OSGi module if possible (meaning that it will only work on Digital Experience Manager 7.0 and above)
- Add the package to the global list of imports for all the transformed modules. This list can be added the WEB-INF/etc/config/felix-framework.properties. The default value comes from the applicationContext-felix.xml file.
15.1.3 LinkageError

This happens when the same class is loaded twice from different bundles. For example, two bundles working together that both use different versions of SLF4j and one embeds it.

15.1.3.1 Solution

Externalize the classes into separate bundles and use import package statements with precise versions to make sure you reduce the possibilities for conflicts.

15.1.4 Large amount of imports generated by the Maven plugins

If you have a large amount of imports being generated by the Maven plugins, this might not always be a good thing, since having many dependencies will make your module difficult to deploy.

15.1.4.1 Solution

There are two possibilities:

- Manually edit the Import-Package instructions for the Maven Bundle plugin
- Embed dependencies that are pulling too many transitive dependencies

In the first solution, you will have complete control over the Import-Package instructions so you shouldn’t have any problems. The main downside of this solution is that it is a bit tedious to first setup since it mostly a trial and error loop of building, deploying over and over until the bundle properly starts. This can take some time initially but the dependencies should rapidly stabilize and then they will usually not need to be modified until another dependency is introduced or modified. Also, using this technique you can also properly specify package dependency version ranges, which is also a good thing for the flexibility of an OSGi bundle deployment.

The second solution usually takes less time to setup, but it will also require some trial and error before it will work properly, although it should be much shorter than the first solution. However, the generated imports might not be minimal and you might still be importing packages that are referenced for example in dead code. Also, it is not a good OSGi practice to embed too many
dependencies since it will make the bundles much larger and potentially cause conflicts if the dependencies are ever exported.
16 DIGITAL EXPERIENCE MANAGER OSGI IMPLEMENTATION

16.1 OSGI FRAMEWORK STARTUP

The OSGi framework is started once DX spring context is instantiated. In turn, FrameworkService initializes Karaf instance which registers all the bundles that will initially be started and made available to all deployed bundles. Of particular note, FrameworkService loads its configuration from the felix-framework.properties file which contains important settings such as which packages are made available to all bundles.

16.2 OSGI SERVLET BRIDGE

The OSGi servlet bridge consists of two parts, a proxy servlet that is mapped as a servlet in the WEB-INF/web.xml file of the Digital Experience Manager web application and then the actual bridge that will make it possible to call “regular” servlet inside the OSGi framework.

16.3 DIGITAL EXPERIENCE MANAGER MODULE EXTENDER

The core of the module integration is in the jahia/bundles/jahiamodule-extender project. Here the Activator class is the starting point, that registers the bundle listeners to listen for the deployment and undeployment of bundles and does all the registration work inside Digital Experience Manager. This is where most of the OSGi “magic” is happening, including the registration and dispatching to JSPs.

16.4 BUNDLE PACKAGING

We can have both system bundles that will be handled by Karaf (such as Karaf extensions) and DX bundles (common DX modules).

We are relaying on Karaf for system bundle. All the system bundles are in the Karaf folder, by default located in digital-factory-data/karaf. From here you can use all the Karaf framework to deploy any bundle / feature / kar file. See more information here:

https://karaf.apache.org/manual/latest
DX has a also watched directory located in digital-factory-data/modules that contains all dx installed bundles

16.5 OSGI AND DIGITAL EXPERIENCE MANAGER MODULE STATES

16.5.1 Bundle life cycle

With the installation of a bundle in the OSGi runtime the bundle is persisted in a local bundle cache. The OSGi runtime then tries to resolve all dependencies of the bundle.

If all required dependencies are resolved, the bundle is in the RESOLVED status otherwise it is in the INSTALLED status.

If several bundles exist which would satisfy the dependency, then the bundle with the highest version is used. If the versions are the same, then the bundle with the lowest install ID will be used (the bundle gets a unique identifier assigned by the framework during the installation). If the bundle is started, its status is STARTING. Afterwards it gets the ACTIVE status.
This life cycle is depicted in the following graphic:

Digital Experience Manager extends the default OSGi lifecycle states to add intermediary states that detail the state in which a module is. You can find the description of these states in the following table.

<table>
<thead>
<tr>
<th>State name</th>
<th>OSGi</th>
<th>DXM</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNINSTALLED</td>
<td>x</td>
<td></td>
<td>Bundle is not installed</td>
</tr>
<tr>
<td>UNRESOLVED</td>
<td>x</td>
<td></td>
<td>Bundle is installed, but dependencies haven't been resolved</td>
</tr>
<tr>
<td>RESOLVED</td>
<td>x</td>
<td></td>
<td>Bundle is installed and all dependencies have been resolved</td>
</tr>
<tr>
<td>Status</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAITING_TO_BE_PARSED</td>
<td>Bundle is started but it depends on other Jahia modules for content definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARSED</td>
<td>Bundle is started and all its content definitions have been parsed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTALLED</td>
<td>Bundle was installed, but not started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATED</td>
<td>Bundle was updated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOPPED</td>
<td>Bundle was stopped</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOPPING</td>
<td>Bundle is stopping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STARTING</td>
<td>Bundle is starting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAITING_TO_BE_STARTED</td>
<td>Bundle is waiting to be started, waiting for another module to be started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAITING_TO_BE_IMPORTED</td>
<td>Bundle is waiting to import its content, waiting for a dependency to import its content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERROR_DURING_START</td>
<td>An error occurred during Jahia module start</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STARTED</td>
<td>Bundle is fully started and available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERROR_WITH_DEFINITIONS</td>
<td>Bundle cannot start due to definitions error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPRING_STARTING</td>
<td>Bundle is started but is waiting for spring context to be available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td></td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SPRING_NOT_STARTED</td>
<td>x</td>
<td>Bundle is started but Spring context cannot be instantiated</td>
<td></td>
</tr>
<tr>
<td>INCOMPATIBLE_VERSION</td>
<td>x</td>
<td>Bundle cannot start due to version errors</td>
<td></td>
</tr>
</tbody>
</table>
17 APPENDIX 1 - CONFIGURING A MODULE THAT EXTENDS THE SYSTEM

Some modules might use frameworks provided by the Digital Experience Manager system and need to extend them.

For example, your module might use some sub projects from the Spring framework. Spring framework is highly configurable and can handle that some jars are present or not in the system. In your module, you need to embed some jars and add some inside Digital Experience Manager WEB-INF/lib

Here are the dependencies we want to add to our module:

```xml
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-core</artifactId>
    <version>${spring-social.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-web</artifactId>
    <version>${spring-social.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-facebook</artifactId>
    <version>${spring-social-facebook.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-twitter</artifactId>
    <version>${spring-social-twitter.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
    <version>${spring-security-crypto.version}</version>
</dependency>

To embed some jars in your project you need to add a line like this in your pom.xml

```xml
<Embed-Dependency>*;groupId=org.springframework.social|org.springframework.security; scope=compile; type=!pom; inline=false</Embed-Dependency>
```
This will restrain the scope of the embedded jars to the one we are declaring as we do not want to embed all of org.springframework.social transitive dependencies (most of them are related to spring framework which is already part of the system).

This example introduce some JSON frameworks (org.codehaus.jackson) that is used both by the social module and by springframework web this means we have a transitive dependency that needs to be added to our system as spring framework web is a system one.

To find our dependencies we can use the plugin dependency from Maven:

```
mvn -o dependency:tree
```

This return something like:

```
[INFO] --- maven-dependency-plugin:2.6:tree (default-cli) @ socialNetworkConnector --
[INFO] org.jahia.modules.socialNetworkConnector:socialNetworkConnector:bundle:2.0- -SNAPSHOT
[INFO] |  +-- org.springframework:spring-tx:jar:3.0.5.RELEASE:compile
[INFO] |  |    +-- aopalliance:aopalliance:jar:1.0:compile
[INFO] |  |    |
[INFO] |  +-- org.springframework:spring-aop:jar:3.0.5.RELEASE:compile
[INFO] |  +-- org.springframework:spring-orm:jar:3.0.5.RELEASE:compile
[INFO] |  +-- org.springframework:spring-beans:jar:3.0.5.RELEASE:compile
[INFO] |  +-- org.springframework:spring-core:jar:3.0.5.RELEASE:compile
[INFO] |  |    |
[INFO] |  +-- org.springframework:spring-web:jar:3.0.5.RELEASE:compile
[INFO] |  +-- org.springframework:spring-webmvc:jar:3.0.5.RELEASE:compile
[INFO] |  |    +-- org.springframework:spring-context:jar:3.0.5.RELEASE:compile
[INFO] |  |    |
[INFO] |  +-- org.springframework:spring-context-support:jar:3.0.5.RELEASE:compile
[INFO] |  |
[INFO] |  |    |
[INFO] |  |
[INFO] |  |
[INFO] |  +-- org.jahia.server:jahia-impl:j:7.0.0.0-SNAPSHOT:provided...
[INFO] |  |
[INFO] |  +-- org.springframework.webflow:spring-js:jar:2.3.2.RELEASE:provided
[INFO] |  |    +-- org.springframework.webflow:spring-js-resources:jar:2.3.2.RELEASE:provided
[INFO] |  |    |
[INFO] |  |
[INFO] |  |
```

As all other packages are transitive dependencies from org.jahia.server

Now we see as a transitive that we depend on org.codehaus.jackson:

Let's see if we find other dependencies towards org.codehaus.jackson on the list, the command can be written as is:

```
mvn -o dependency:tree -Dincludes=org.codehaus.jackson,org.springframework,org.jahia
```

result:

```
[INFO] --- maven-dependency-plugin:2.6:tree (default-cli) @ socialNetworkConnector

[INFO] org.jahia.modules.socialNetworkConnector:socialNetworkConnector:bundle:2.0-SNAPSHOT
[INFO] | \ org.jahia.server:jahia-impl:jar:7.0.0.0-SNAPSHOT:provided
[INFO] | \ org.codehaus.jackson:jackson-mapper-asl:jar:1.9.9:compile
[INFO] | \ org.springframework:spring-social-facebook:jar:1.0.3.RELEASE:compile
[INFO] | \ org.springframework:spring-social-twitter:jar:1.0.5.RELEASE:compile
[INFO] | \ org.springframework.security:spring-security-crypto:jar:3.1.3.RELEASE:compile
[INFO] | \ org.springframework:spring-security:jar:3.1.3.RELEASE:compile
[INFO] | \ org.springframework:spring-aop:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-orm:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-beans:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-core:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-asm:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-webmvc:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-web:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context-support:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-jdbc:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-expression:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-jdbc:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-orm:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-aop:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-beans:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-core:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-asm:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-webmvc:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-web:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context-support:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-expression:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-jdbc:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-orm:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-aop:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-beans:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-core:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-asm:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-webmvc:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-web:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context-support:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-context:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-expression:jar:3.0.5.RELEASE:compile
[INFO] | \ org.springframework:spring-jdbc:jar:3.0.5.RELEASE:compile

This means that Digital Experience Manager system has also some dependencies towards org.codehaus.jackson, so we need to add org.codehaus.jackson in WEB-INF/lib to make it available for Digital Experience Manager itself.
Now those classes will also be used inside our module so we need to tell our OSGI container (Felix) that those classes are system classes and need to be shared with the bundles.

To do so we need to modify WEB-INF/etc/config/felix-framework.properties and add all the packages we want to make available to bundles in that file.

```java
org.codehaus.jackson;version="1.9.9",
org.codehaus.jackson.annotate;version="1.9.9",
org.codehaus.jackson.format;version="1.9.9",
org.codehaus.jackson.impl;version="1.9.9",
org.codehaus.jackson.io;version="1.9.9",
org.codehaus.jackson.jackson.map;version="1.9.9",
org.codehaus.jackson.map.annotate;version="1.9.9",
org.codehaus.jackson.map.deser;version="1.9.9",
org.codehaus.jackson.map.deser.impl;version="1.9.9",
org.codehaus.jackson.map.deser.std;version="1.9.9",
org.codehaus.jackson.map.exc;version="1.9.9",
org.codehaus.jackson.map.ext;version="1.9.9",
org.codehaus.jackson.map.introspect;version="1.9.9",
org.codehaus.jackson.map.jsontype;version="1.9.9",
org.codehaus.jackson.map.jsontype.impl;version="1.9.9",
org.codehaus.jackson.map.module;version="1.9.9",
org.codehaus.jackson.map.ser;version="1.9.9",
org.codehaus.jackson.map.ser.impl;version="1.9.9",
org.codehaus.jackson.map.ser.std;version="1.9.9",
org.codehaus.jackson.map.type;version="1.9.9",
org.codehaus.jackson.map.util;version="1.9.9",
org.codehaus.jackson.node;version="1.9.9",
org.codehaus.jackson.schema;version="1.9.9",
org.codehaus.jackson.sym;version="1.9.9",
org.codehaus.jackson.type;version="1.9.9",
org.codehaus.jackson.util;version="1.9.9",
```

Right now, there is no easy way to list those packages with version number so you need to generate that list by hand (jar tf on jar file will list all files).

Now you need to specify the import needed by your module to work. Digital Experience Manager will try to detect those packages as much as possible, to do so include that in your Import-Package directive in your pom.xml file:

```xml
<Import-Package>
    ${jahia.plugin.importPackage}
</Import-Package>
```

Now you can deploy your module and start your jahia (needed to take your new libs and properties into account)
If your module does not start you might be missing some imports, add them to your configuration.

Following our example, you end with something like that,

```
<Import-Package>
    ${jahia.modules.importPackage},
    javax.crypto,
    javax.crypto.spec,
    javax.servlet.http,
    net.sf.cglib.core,
    net.sf.cglib.proxy,
    org.apache.http,
    org.apache.http.client,
    org.apache.http.client.methods,
    org.apache.http.entity,
    org.apache.http.impl.conn.tsccm,
    org.apache.http.impl.client,
    org.apache.http.impl,
    org.apache.http.params,
    org.apache.http.util,
    org.apache.log4j,
    org.codehaus.jackson,
    org.codehaus.jackson.annotate,
    org.codehaus.jackson.format,
    org.codehaus.jackson.impl,
    org.codehaus.jackson.io,
    org.codehaus.jackson.sym,
    org.codehaus.jackson.type,
    org.codehaus.jackson.util,
    org.codehaus.jackson.map,
    org.codehaus.jackson.map.module,
    org.jahia.bin,
    org.springframework.util,
    org.springframework.web.util,
    org.springframework.web.client,
    org.springframework.http,
    org.springframework.http.client,
    org.springframework.core,
    org.springframework.http.converter,
    org.springframework.http.converter.json,
    org.slf4j,
    org.slf4j.impl,
    org.slf4j.spi,
    org.json
</Import-Package>
```

To find packages in a jar file you can use this line of command by providing the path to the jars you want to analyze:
mvn -o jahia:osgi-inspect -DjarBundles=/home/rincevent/.m2/repository/org/codehaus/jackson/jackson-core-asl/1.9.9/jackson-core-asl-1.9.9.jar,/home/rincevent/.m2/repository/org/codehaus/jackson/jackson-mapper-asl/1.9.9/jackson-mapper-asl-1.9.9.jar -DdumpHeaderOnly=false

This will dump the headers from the manifest file, option 'dumpHeaderOnly' allows analyzing the jar and listing the packages found in it if this is not an OSGI bundle.

Here an excerpt of the output:

```plaintext
[INFO] /home/rincevent/.m2/repository/org/codehaus/jackson/jackson-core-asl/1.9.9/jackson-core-asl-1.9.9.jar header dump:
Bnd-LastModified: 1343496987929
Built-By: tsaloranta
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0.txt
Bundle-ManifestVersion: 2
Bundle-Name: Jackson JSON processor
Bundle-RequiredExecutionEnvironment: J2SE-1.5 JavaSE-1.6
Bundle-SymbolicName: jackson-core-asl
Bundle-Vendor: http://fasterxml.com
Bundle-Version: 1.9.9
Created-By: 1.6.0_33 (Apple Inc.)
Export-Package: org.codehaus.jackson.format; version=1.9.9;
    uses:=org.codehaus.jackson.io,org.codehaus.jackson.annotate
    org.codehaus.jackson.io; version=1.9.9;
    uses:=org.codehaus.jackson.util
    org.codehaus.jackson.util; version=1.9.9;
Implementation-Title: Jackson JSON processor
Implementation-Vendor: http://fasterxml.com
Implementation-Version: 1.9.9
Import-Package: org.codehaus.jackson; version=1.9.9
    org.codehaus.jackson.annotate; version=1.9.9
    org.codehaus.jackson.format; version=1.9.9
    org.codehaus.jackson.impl; version=1.9.9
    org.codehaus.jackson.io; version=1.9.9
    org.codehaus.jackson.sym; version=1.9.9
    org.codehaus.jackson.type; version=1.9.9
Manifest-Version: 1.0
Specification-Title: JSON - JavaScript Object Notation
Specification-Version: 1.0
Tool: Bnd-unknown version
```
[INFO] List of package from jar file:
/home/rincevent/.m2/repository/org/codehaus/jackson/jackson-core-asl/1.9.9/jackson-core-asl-1.9.9.jar
[INFO] org.codehaus.jackson;version=1.9.9
[INFO] org.codehaus.jackson.annotate;version=1.9.9
[INFO] org.codehaus.jackson.format;version=1.9.9
[INFO] org.codehaus.jackson.impl;version=1.9.9
[INFO] org.codehaus.jackson.io;version=1.9.9
[INFO] org.codehaus.jackson.sym;version=1.9.9
[INFO] org.codehaus.jackson.type;version=1.9.9
[INFO] org.codehaus.jackson.util;version=1.9.9

[INFO] /home/rincevent/.m2/repository/org/codehaus/jackson/jackson-mapper-asl/1.9.9/jackson-mapper-asl-1.9.9.jar header dump:
Bnd-LastModified: 1343496988957
Built-By: tsaloranta
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0.txt
Bundle-ManifestVersion: 2
Bundle-Name: Data mapper for Jackson JSON processor
Bundle-RequiredExecutionEnvironment: J2SE-1.5
  JavaSE-1.6
Bundle-SymbolicName: jackson-mapper-asl
Bundle-Vendor: http://fasterxml.com
Bundle-Version: 1.9.9
Created-By: 1.6.0_33 (Apple Inc.)
DynamicImport-Package: org.joda.time
  org.joda.time.format
  org.w3c.dom.ls
  org.w3c.dom.bootstrap
Export-Package: org.codehaus.jackson.schema; version=1.9.9;
  uses:=org.codehaus.jackson.node,org.codehaus.jac...
With those information you can modify your file WEB-INF/etc/config/felix-framework.properties to add those packages.
18 APPENDIX 2- ADDITIONAL RESOURCES

18.1 OSGI GLOSSARY

18.1.1 General OSGi terms

OSGi Core specification – This is the core specification of how class loaders work with bundles, how bundles are specified, how services are registered and how their life cycle works.

OSGi Compendium specification - the compendium defines specific services such as the declarative services specification, the configuration admin service, remote services spec, etc...

OSGi Enterprise specification - introduces enterprise specific notions such as persistence support, transactions, the OSGi Blueprint specification based on Spring Dynamic modules to enable powerful dependency injection, etc.

18.1.2 OSGi core implementations

Apache Felix - the OSGi implementation at Apache. This project is the “minimal” OSGi implementation, and most advanced features are now moved to other project such as Apache Karaf or Apache Aries. Felix is embedded in Apache Sling, Karaf, ServiceMix, Adobe’s CRX, and many other servers.

Eclipse Equinox - the Eclipse implementation of the OSGi framework. This is historically the first implementation of the OSGi specification and therefore usually the most complete. Apache Felix is usually lagging a bit in terms of features behind the Equinox implementation but this is not necessarily a bad thing since it focuses on being minimal. Equinox is embedded in the Eclipse IDE as well as the WebSphere Application server.

18.1.3 OSGi compendium frameworks & implementations

OSGi Declarative Services - initial dependency injection framework based on XML descriptors and implemented in Apache Felix under the name SCR. Has plugins to use annotations. This is historically the oldest dependency framework and therefore the most mature in terms of implementations, but is also quite limited and tedious to work with.
OSGi Blueprint - dependency injection framework for OSGi based on the initial work done by Spring Dynamic Modules. The reference implementation of the Blueprint specification is actually the Spring DM implementation, which was now donated to Eclipse under the name Eclipse Gemini Blueprint project.

Apache Aries - also includes another implementation of the OSGi Blueprint specification, and is implementing the Enterprise OSGi specification more globally (such as persistence or transactions), using some services from other projects such as Apache ServiceMix or Apache Geronimo.

18.1.4 Other important frameworks

Apache Felix iPOJO - an alternative to OSGi Declarative Service or OSGi Blueprint, this is another dependency framework that was not standardized but that is compatible with multiple OSGi framework implementations. It is also quite well documented. Here is a comparison between the three dependency framework implementations:

http://felix.apache.org/site/ipojo-faq.html#iPOJOFaq-HowdoesiPOJOcomparetoDeclarativeServicesorBlueprint%253F

http://felix.apache.org/site/ipojo-faq.html - iPOJOFaq-HowdoesiPOJOcomparetoDeclarativeServicesorBlueprint%253F

Apache Karaf - designed to be a lightweight OSGi platform that includes useful features out of the box for developers and production. This is designed to be used as a generic OSGi server, whereas Apache Felix is just the basic code. For example, Apache Karaf can deploy “packs of bundles” called “features” making it easier to deploy a real-world application than deploying all bundles manually on Apache Felix. Karaf is also designed to run as a service on an operating system and provides shell scripts and documentation on how to do this. Also, the command line on Karaf is much more powerful than the Apache Felix one. So when would you prefer using Felix instead of Karaf? Well mostly when you want to embed an OSGi framework and want precise control over what you include or not. For example, in our Jahia Config Tool project we chose to use Felix directly since we were embedding it as a standalone Java application. This doesn't mean we can’t deploy Apache Karaf bundles inside Felix, quite the opposite in fact, but requires more developer setup while Karaf includes a lot of functionality out of the box. If you
are looking to develop a web application, it would make more sense to use Apache Karaf, you’ll be much more productive.

Apache ServiceMix - a flexible, open-source integration container that unifies the features and functionality of Apache ActiveMQ, Camel, CXF, ODE, Karaf into a powerful runtime platform you can use to build your own integrations solutions. It provides a complete, enterprise ready ESB exclusively powered by OSGi.

18.1.5 Other terms

MANIFEST-FIRST development - OSGi development with the Eclipse IDE is done a little differently than when using Maven.

18.2 WEB REFERENCES

- Mine of Information: 10 minutes introduction to OSGi, http://moi.vonos.net/java/osgi-intro/
- What is OSGi, http://www.osgi.org/Technology/WhatIsOSGi
- How to get started with OSGi, http://www.osgi.org/Technology/HowOSGi
- BND tool, http://www.aqute.biz/Bnd/Bnd
- Apache Felix OSGi implementation, https://felix.apache.org

Best practices for developing and working with OSGi applications,