Deploying Arbitrary Resources

In This Topic

A plugin may contain Java files, images, JSPs, etc. Some of the resource types are covered by the component types that XperienCentral supports. For example, XperienCentral supports deploying JSPs for the website environment by the presentation component and deploying static files by all component types.

This part describes the deployment of any resource that is not covered by a component type. Examples are:

- Deploying a custom tag library
- Deploying filters
- Deploying a custom `web.xml`
- Deploying a custom library
- Adding an arbitrary resource to your plugin which must be accessible from the plugin

Deploying Custom JSP Tags

A tag library is a collection of JSP tags you can import and use in a JSP. JSP tags are in fact functions you can invoke from any JSP in the same way you define methods in a Java class and import that class into another.

To deploy a custom tag library onto a normal web application, follow these steps:

1. Define the tag library in the `<jsp-config>` section of the `web.xml`, for example:

```
<taglib>
        <taglib-uri>http://www.gx.nl/taglib/idg_functions</taglib-uri>
        <taglib-location>/WEB-INF/prj/tld/gx_functions.tld</taglib-location>
</taglib>
```

2. The `taglib-location` in the `web.xml` must point to the location of the .TLD file which defines the contents of that tag library. This .TLD may define a function that points to a method in a particular Java class on the Application server's classpath, for example:

```
<function>
        <name>getAllBooks</name>
        <function-class>nl.gx.product.BookTag</function-class>
        <function-signature>Book[] getAllBooks()</function-signature>
</function>
```

3. Compile the Java classes that implement the actual tags to a JAR and copy them to `/WEB-INF/lib` in order to ensure that they are on the application server's classpath.

Because the `web.xml` must be changed and this cannot be done at runtime, it is not possible to add a custom tag or tag library from a plugin, therefore, in order to add a custom tag or tag library, you must perform the steps above. You must restart the application server in order to apply the changes.

Back to Top

Deploying Custom JSP Tag Files

A custom JSP tag file is very similar to a custom JSP tag, the only difference is that the actual function is not implemented in Java but in JSP. The extension of the custom tag file is by convention `.tag`.

To deploy a custom tag file, perform the following steps:

1. Create a `/tags` subdirectory within the `/src/main/resources` directory of your plugin.
2. Create or copy the tag file to the newly created `/tags` directory.

When the plugin is deployed, the tag file is automatically available through the following path:

`/WEB-INF/tags/[…BUNDLE_ID…]`

Where the `[…BUNDLE_ID…]` is the actual bundle ID of the plugin, for example `nl.gx.webmanager.mypresentation`.

To use the tag file, you must add the following line to your plugin:

```
<%@ taglib tagdir="/WEB-INF/tags/[…BUNDLE_ID…]" prefix="myprefix" %>
```

You can then call the tag as you would normally do using the prefix and tag filename.

Back to Top

Adding and Accessing an Arbitrary Resource in a Plugin

This part describes how to distribute and access an arbitrary resource within a plugin. For example, suppose that you want to distribute along with a plugin a static .properties file containing configuration properties used by some Java class within the plugin.

⚠ This example describes static properties - If you want the properties to be configurable, you must use the `ConfigurationManagement` service instead.

To distribute the resource with the plugin you need to simply put it somewhere in `/src/main/resources`. It can be stored in any subfolder. To access this resource from Java, simply retrieve the resource from the classloader of that class by invoking `getResourceAsStream` on the class. For example, if we distributed a file called `config.properties` in `/src/main/resources/conf/books` we could access this resource from `Book.java` like this:

```
Class<Book>
clazz = Book.class;
InputStream propFile = clazz.getResourceAsStream("conf/books/config.properties" );
Properties props = new Properties();
props.load( propFile);
```

Back to Top

Deploying and Accessing Static Files

By static files we mean files that are typically static within the context of a web application like images, JavaScript files, HTML files and stylesheets. Such static files can have two purposes:

- To be used in the Workspace. The file is only used to render the plugin in the Workspace.
- To be used in the website environment. The file is part of the website presentation.

Deploying Static Files to the Edit Environment

To deploy static files to the Workspace, simple copy the resources to `/src/main/resources/static/backend`. When the plugin is deployed, the files will be copied to the directory of the static web application in a folder named `/wm/b/wcb/<Component bundle definition ID>`.

Since this directory is not accessible from the website environment, the file will not be accessible. To access static files from Java, the URL can be calculated by:

```
public String getIcon() {
        return "/wm/b/wcb/" + Activator.BUNDLE_DEFINITION_ID + "/icon.gif";
}
```

Alternatively, some API classes contain methods to retrieve the static backend directory:

```
CmsItemBase.getStaticResourceDir();
ElementComponentDefinition:getStaticBackendDir();
```

If the full path name to the file is needed, the pathname of the static web root directory can be retrieved from the Configuration management service:

```
confManagement.getParameter("website_settings.www_root_directory");
```

To point to this resource from a JSP, the easiest way is to set the path into the reference data of the controller. The controller always has access to the Activator of the component by which it is contained. For example:

```
public Map<String, Object> referenceData(HttpServletRequest req, Object command, Errors errors) throws
Exception {
        Map<String, Object> data = super.referenceData(req,command, errors); data.put("staticBackendDir","/wm/b
/wcb/" +
        Activator.BUNDLE_DEFINITION_ID + "/");
        return data;
}
```

All properties written to the reference data are available directly in the JSP, so to define the URL to an icon called `icon.gif`, the following code snippet would display that icon in a JSP:

```
<img src=”${staticBackendDir}icon.gif”/>
```

Deploying Static Files to the Website Environment

To deploy static files to the website environment, the presentation component methodology as described in Presentation Component should be used. For this reason resources should be placed in `/src/main/resources/presentationtype/static` - The files will be copied to `/static`, inside the web root directory of the static web application when the plugin is deployed.

In order to make this work, the plugin must define a presentation component, therefore in order to develop a plugin containing an element component also containing such static files, a presentation component must be defined in the activator of that plugin also.

To access the static files from Java, the method `website.getStaticFilesUrl` can be invoked which returns the web root of the static web application. Note that `/static` must be appended to the filename since the method returns only the web root.

To access the static files from a JSP, the same method can easily be used since the website is available in the `presentationcontext` which is available in any JSP. For example:

```
<c:set var="staticFilesUrl"
value="${presentationcontext.website.staticFilesUrl}" />
<img src="${staticFilesUrl}/static/project/icon.gif"/>
```

There is an important difference between static files or JSPs for the website environment distributed with the plugin and those distributed with a separate presentation component. Static files and JSPs distributed with a plugin that contain only resources for that plugin are intended for demonstration purposes only. They provide a default layout which ensures that the plugin has a presentation on any website onto which it is deployed.

Because each website uses a different presentation, a plugin should never contain the actual presentation itself. A website usually has one active presentation component which contains all JSPs and static files used to render the complete website environment which overwrites all presentations and static files in a plugin (if it contains these resources).

To ensure that the presentation plugin overwrites all presentations and static files from the individual plugins, it must be deployed after deploying all other plugins.

Back to Top

Deploying Custom Libraries

Using Java libraries which are available in the Maven repository is quite simple. Defining the artifact dependency in the `pom.xml` of the plugin will automatically bundle the JAR with the plugin and class loading issues will be handled by the framework automatically upon using the library at runtime.

It becomes more complicated when a Java library is used that is not contained by the Maven repository. The best approach in this case is to simply upload the JAR file to the (local or remote) Maven repository.

The following steps must be performed in order to use a custom Java library in a plugin:

1. Execute a Maven command to upload the JAR file and to create a new artifact for it in the Maven repository. This command is explained below.
2. Define the artifact dependency in the `pom.xml`. An example is given below.
3. Compile the plugin, open its generated `MANIFEST.MF`, and if the library should also be available to other plugins, copy the value of `Import-Package` from the Manifest into a new attribute `importPackage` in the `build/plugins/plugin/configuration/osgiManifest` section of the `pom.xml`.
4. Copy the value of `Import-Package` from the manifest into a new attribute `exportPackage` in the `build/plugins/plugin/configuration/osgiManifest` section of the `pom.xml`.
5. If the library should only be available to this plugin, copy the value of `Import-Package` from the Manifest into a new attribute `importPackage` in the `build/plugins/plugin/ configuration/osgiManifest` section of the `pom.xml`. For this situation, the `Import-Package` value contains too many packages: remove all packages that are contained by the Java library.

In both situations, follow the rule: “Import what you export”.

The Maven command to upload a JAR file and create a new artifact for it in the Maven repository has the following syntax:

```
mvn install:install-file -Dfile=[filename] -DgroupId=[groupId] -DartifactId=[artifactId] -Dpackaging=jar -
Dversion=[version] -s [path to settings.xml]
```

where:

- `filename` is the name of the JAR file to upload
- `groupId` is the group ID that the new Maven artifact should have
- `articfactId` is the artifact ID that the new Maven artifact should have
- `version` is the version number that the new Maven artifact should have

As an example we describe how to use an external library called `konakart.jar` which we will not share with other plugins:

1. Run the Maven command to upload the JAR file to the Maven repository:

   ```
   mvn install:install-file -Dfile=konakart.jar -DgroupId=com.konakart -DartifactId=konakart -Dpackaging=jar -
   Dversion=1.0.0 -s settings.xml
   ```

   The library is now available in the Maven repository as an artifact with group Id `com.konakart` and the artifact Id `konakart`.

2. Define a dependency for this new artifact with scope provided in the `pom.xml`:

```
<dependency>
  <groupId>com.konakart</groupId>
  <artifactId>konakart</artifactId>
  <scope>provided</scope>
  <version>1.0.0</version>
</dependency>
```

3. Build the plugin.
4. Open the `MANIFEST.MF` file inside the generated JAR file and copy the value of the `Import-Package` property to the clipboard.
5. Open the `pom.xml` and create a new attribute `importPackage` in the `build/plugins/plugin/configuration/osgiManifest` section.
6. Paste the value from the clipboard into the `importPackage` attribute. Remove all packages exported by the `konakart` library (this can be looked up by opening the `konakart.jar`).