Chapter 1. Packaging service

Table of Contents

This service is used essentially in the context of PaaS. To deploy an application in the platform, this service take a descriptor of the application as parameter and generate an addon.

1.1. Cloud descriptors

These are xml files which describe :

- Environment (`environment-template.xml`): describes the components of the platform such as instances of JOnAS application servers, data sources, etc..
- Application (`cloud-application.xml`): Each application is described in a file of that type. An application can be composed of several modules, their locations are specified in this file.
- Mapping topology (`deployment.xml`): describes the relationship between applications and application servers

To parse these xml files, some APIs were developed in JPaaS-Util[1] project using JAXB.

1.1.1. Example of use for `cloud-application.xml`

First, create an instance of Cloud application descriptor. The constructor has three signatures :

- Empty constructor

```
/**
* Default constructor
*/
public CloudApplicationDesc() throws Exception;
```

- With the URL of the xml file

```
/**
* Constructor with xml url
* @param urlCloudApplication
* @throws Exception
*/
public CloudApplicationDesc(URL urlCloudApplication) throws Exception;
```

- With the content of the xml file

```
/**
* Constructor with xml content
* @param cloudApplication
* @throws Exception
*/
public CloudApplicationDesc(String cloudApplication) throws Exception;
```

[1] http://gitorious.ow2.org/ow2-jasmine/jpaas-util

1.1.1.1. Parse an xml file

To parse an xml file, use the constructor with the URL of xml content.

```
URL urlCloudApplication = new URL(CLOUD_APPLICATION_URL);
CloudApplicationDesc desc = new CloudApplicationDesc(urlCloudApplication);
```

Then, retrieve the root element of xml schema :

```
CloudApplicationType cloudApplication = desc.getCloudApplication();
```

After, for getting any element content in the xml file, just call the corresponding method on the root element. To get application name for example, do :

```
String applicationName = cloudApplication.getName();
```

The `cloud-application.xml` contains a list of deployables that form the application. These deployables are defined in two separated namespaces `embedded-xml` and `artefact`. To get the list of deployables from `cloud-application.xml` do:

```
// Gets deployables
DeployablesType deployables = cloudApplication.getDeployables();
List<Object> listDeployables = deployables.getDeployables();
```

Then, for each element in `listDeployables` test if it is an instance of `org.ow2.jonas.jpaas.util.clouddescriptors.cloudapplication.artefact.v1.generate` or `org.ow2.jonas.jpaas.util.clouddescriptors.cloudapplication.xml.v1.generated.Xml`

1.1.1.2. Construct an xml file

To generate a `cloud-application.xml` file, start by creating a new element corresponding to the root element of the xml:

```
CloudApplicationType cloudApplicationType = new CloudApplicationType();
```

Then, set the content of each xml element. For example, to set application name do :

```
cloudApplicationType.setName("my application");
```

To construct the list of deployables, retrieve the empty list of deployables created when `cloudApplicationType` was instantiated :

```
DeployablesType deployablesType = new DeployablesType();
List<Object> listDeployables = deployablesType.getDeployables();
```

And then, create deployables. Suppose that the application has one artefact deployable, do:

```
ArtefactDeployableType deployable = new ArtefactDeployableType();
deployable.setName("my deployable");
deployable.setId("my id");
deployable.setLocation(MY_DEPLOYABLE_URL);
...
```

Add this deployable to the list of deployables :

```
listDeployables.add(deployable);
```

Now, all xml elements were set. So, generate the xml. For that, create an empty cloud application descriptor :

```
CloudApplicationDesc desc = new CloudApplicationDesc();
```

Construct the JAXB element corresponding to the root element `cloudApplicationType`:

```
ObjectFactory objectFactory = new ObjectFactory();
JAXBElement<CloudApplicationType> cloudApplication =
 objectFactory.createCloudApplication(cloudApplicationType);
```

And call the method `generateCloudApplication()` :

```
String xml = desc.generateCloudApplication(cloudApplication);
```

This one returns the xml content.

1.2. Packaging

The Packaging service is responsible of package of applications in the platform. It takes a `cloud-application.xml` (required) and a tenant identifier (optional) as parameters. By default, the addon is placed in `$JONAS_BASE/work/addons`, set the `outputDir` parameter otherwise.

1.2.1. Computing generated addons number

Each application deployable can include these requirements :

```
<artefact:requirements>
  <artefact:requirement>(collocated-to=id1)</artefact:requirement>
  <artefact:requirement>(not-collocated-to=id2)</artefact:requirement>
</artefact:requirements>
```

`collocated-to` means "must be hosted in the same container". Similarly, `not-collocated-to` means "must be not hosted in the same container".

Therefore, it is possible that as a result of packaging of one application, more than one addon will be generated (the effect of `not-collocated-to` property). In order to optimize the number of addons generated, an algorithm was implemented.

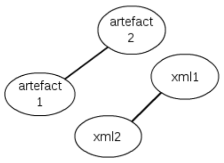
This problem is resolved by using a non-oriented graph. Each deployable is represented by a vertex. There is an edge between two vertices if these represented deployables are collocated.

Consider this `cloud-application.xml`:

```
<cloud-application>
  ...
  <deployables>
    <artefact:deployable name="ear" id="artefact1">
      <artefact:location></artefact:location>
      <artefact:requirements>
        <artefact:requirement>(collocated-to=artefact2)</artefact:requirement>
        <artefact:requirement>(not-collocated-to=xml2)</artefact:requirement>
      </artefact:requirements>
    </artefact:deployable>

  <artefact:deployable name="war" id="artefact1">
    <artefact:location></artefact:location>
    <artefact:requirements>
      <artefact:requirement>(not-collocated-to=xml1)</artefact:requirement>
    </artefact:requirements>
  </artefact:deployable>

  <embedded-xml:deployable name="xml1" id="xml1">
```

```
    <embedded-xml:requirements>
      <embedded-xml:requirement>(collocated-to=xml2)</embedded-xml:requirement>
    </embedded-xml:requirements>
  </embedded-xml:deployable>

  <embedded-xml:deployable name="xml2" id="xml2">
    <embedded-xml:requirements>
      <embedded-xml:requirement>(collocated-to=xml1)</embedded-xml:requirement>
      <embedded-xml:requirement>(not-collocated-to=artefact1)</embedded-xml:requirement>
    </embedded-xml:requirements>
  </embedded-xml:deployable>
  </deployables>
</cloud-application>
```

We deduce the following graph :



Then, we take each connected components, in this case `{artefact1, artefact2}` and `{xml1, xml2}`, and we try to merge them one by one. Two connected components are merged if any element in the first has a `not-collocated-to` requirement with an element in the second. This is not the case in the example above because `artefact1` which is in the first connected component is `not-collocate-to` `xml2` which is in the second component.

After merging, the final connected components represent the content of generated addons. In the example, the first addon will contain `artefact1` and `artefact2` and the second addon will contain `xml1` and `xml2`.

1.2.2. Addon's repository provisioning

After computing the content of each addon to generate, a deployment plan is generated and deployables will be stored in the maven `repository/` directory inside the addon :

- The groupid of all deployables is `org.ow2.jonas.jpaas`
- The version of deployables is the application version set in `cloud-application.com`.
- The artifact of each deployable is its name.
- The type for `artefact` deployables is the extension of the file. For `xml` deployable, the type is `xml`.

When the addon is being deployed, the `repository` inside the addon is added to `repositoryManager`.

1.3. Usage

Packaging service can be used by two way:

1.3.1. As a JOnAS service

Start a addon profile JOnAS and deploy the addon of packaging service. See `org.ow2.jonas.packaging.IPackagingManager`.

1.3.2. Command line

It is possible to generate application addon using a maven plugin.

```

<groupId>org.ow2.jonas.tools.maven</groupId>
```

```
<artifactId>maven-packaging-plugin</artifactId>
<phase>generate-resources</phase>
<goal>generate-application-addon</goal>
```

This plugin require the URL of `cloud-application.xml` and the URL of `outputDirectory`. In fact, this plugin is executed without a running JOnAS. TenantId is an optional parameter. Basic maven plugin usage:

```
mvn org.ow2.jonas.tools.maven:maven-packaging-plugin:5.3.0-M7-SNAPSHOT:generate-application-
addon
-DurlCloudApplication=cloud-application.xml -DoutPutDirectory=. -Dtenant-id=T1
```

To use easily this plugin, a shell script is written and simplify the command line :

```
Usage: gen-addon -app urlCloudApplication -out urlOutputDir
Options:
  -tid   tenant identifier value
  -env   url of environment-template.xml
  -map   url of deployment.xml (mapping topology)
```