Chapter 1. Creating a New JOnAS Service

Table of Contents

1.1. Target audience and rationale

This chapter is intended for advanced JOnAS users who require that some "external" services run along with the JOnAS server. A service is something that may be initialized, started, and stopped. JOnAS itself already defines a set of services, some of which are cornerstones of the JONAS Server. The JOnAS pre-defined services are listed in Configuring JOnAS services [configuration_guide.html#config.services] .

Java EE application developers may need to access other services, for example another Web container or a Versant container, for their components. Thus, it is important that such services be able to run along with the application server. To achieve this, it is possible to define them as JOnAS services.

This chapter describes how to define a new JOnAS service and how to specify which service should be started with the JOnAS server.

1.2. Introducing a new service

The customary way to define a new JOnAS service is to encapsulate it in a class whose interface is known by JOnAS. More precisely, such a class provides a way to start and stop the service. Then, the `jonas.properties` file must be modified to make JOnAS aware of this service.

1.2.1. Defining the service interface and implementation

A JOnAS service is represented by a class that implements its service interface and extends the class `org.ow2.jonas.lib.service.AbsServiceImpl`, and thus must implement at least the following methods:

- `public void doStart() throws ServiceException;`
- `public void doStop() throws ServiceException;`

These methods will be called by JOnAS for starting and stopping the service.

The service interface should look like the following:

```
package com.company.jonas.myservice;

/**
 * Defines the "business" interface of the service (no life-cycle methods).
 */
public interface IMyService {
    void doSomething();
}
```

The service class should look like the following:

```
package com.company.jonas.myservice.internal;

import com.company.jonas.myservice.IMyService;

import org.ow2.jonas.lib.service.AbsServiceImpl;
import org.ow2.jonas.service.ServiceException;

.....

public class DefaultMyService extends AbsServiceImpl implements IMyService {

    /** A configurable property. */
    private String property;

    @Override
    protected void doStart() throws ServiceException {
        // Do something on start-up
        // All the configuration and service dependencies has been injected at this time
    }

    @Override
    protected void doStop() throws ServiceException {
        // Do something on shutdown
        // Service dependencies are still available
    }

    public void doSomething() {
        // Business code ...
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

1.2.2. Building the OSGi bundle

A JOnAS service must be packaged in an OSGi bundle in order to be deployed on the JOnAS OSGi platform. It implies to create a standard packaging structure. An OSGi bundle is like a classic JAR file plus a specific MANIFEST file. To ease the service creation, configuration and management of dynamic dependencies to other JOnAS services, it is recommanded to use iPOJO [http://felix.apache.org/site/apache-felix-ipojo.html] to build your services. This guide will present the iPOJO solution.

The JAR structure of the bundle must contain the following parts:

com/company/jonas/myservice	contains the service interface(s)
com/company/jonas/myservice/internal	contains the service implementation classe(s)
META-INF/MANIFEST.MF	OSGi manifest file
metadata.xml	iPOJO metadata file

The OSGi MANIFEST should contain the following attributes:

```
Import-Package: org.ow2.jonas.lib.service, org.ow2.jonas.service, ...
Export-Package: com.company.jonas.myservice
Private-Package: com.company.jonas.myservice.internal
Bundle-Version: 5.1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyService
```

```
Bundle-SymbolicName: com.company.jonas.myservice
```

If the project is built with Maven [http://maven.apache.org/], it is possible to generate this file during the project compilation thanks to the maven-bundle-plugin [http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html]. This plugin is based on the BND [http://www.aqute.biz/Code/Bnd] tool which can also be used separately.

1.2.2.1. Building with maven

Even if it's not mandatory to use maven to build OSGi bundles, the JOnAS team use it extensively because of the large number of integrated OSGi™ build tools.

1.2.2.1.1. Maven project structure

The maven project that will contains the new service's code is structured just like any well known maven project:

```
<myservice>/
  src/
    main/
      java/
        a/b/<myservice>/
          ...
      resources/
        META-INF/
          <myservice>.bnd (optional)
        metadata.xml
    test/
      ...
  pom.xml
```

- `src/main/java/` contains all the Java source code that have been written for the new service (interface + implementation + any other new classes)
- `src/main/resources/` contains the iPOJO component description file named metadata.xml, plus an optional Bnd descriptor that will be used to describe the bundle's content (for advanced usages)
- `src/test/` contains all the unit tests that may have been written to unit test the service implementation (optional altough recommanded)
- `pom.xml` is the maven project's module descriptor

1.2.2.1.2. Maven project descriptor (pom.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/xsd/
maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.company.jonas</groupId>
  <artifactId>myservice</artifactId>
  <version>1.0.0</version>
  <packaging>bundle</packaging>

  <name>JOnAS Service</name>

  <properties>
    <jonas.version>5.1.2</jonas.version>
    <ipojo.version>1.4.0</ipojo.version>
  </properties>

  <dependencies>
    <!-- Imports the AbsService and required interfaces in our module -->
    <dependency>
      <groupId>org.ow2.jonas</groupId>
      <artifactId>jonas-commons</artifactId>
```

```
      <version>${jonas.version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Creates the bundle -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include>
          </instructions>
        </configuration>
      </plugin>

      <!-- Manipulate the bundle (iPOJO) -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-ipojo-plugin</artifactId>
        <version>${ipojo.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>ipojo-bundle</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

</project>
```

Here we go into more details about the different sections of this file.

```
  <groupId>com.company.jonas</groupId>
  <artifactId>myservice</artifactId>
  <version>1.0.0</version>
  <packaging>bundle</packaging>
```

- `groupId`, `artifactId` and `version` elements are identifying uniquely the bundle in the maven repository
- `packaging` defines the type of the artifact produces by this module. In this case, an OSGi™ bundle.

```
  <properties>
    <jonas.version>5.1.2</jonas.version>
    <ipojo.version>1.4.0</ipojo.version>
  </properties>
```

The `properties` section defines re-usables key/value pairs. Here, it is used to store the JOnAS version that will be used to compile the module against. It also defines the ipojo version to be used.

Important

Notice that there is a relationship between JOnAS and iPOJO version:

- JOnAS 5.1.x uses iPOJO 1.4.0
- JOnAS 5.2.x uses iPOJO 1.6.x

```
  <dependencies>
    <!-- Imports the AbsService and required interfaces in our module -->
    <dependency>
      <groupId>org.ow2.jonas</groupId>
      <artifactId>jonas-commons</artifactId>
      <version>${jonas.version}</version>
      <scope>provided</scope>
```

```
    </dependency>
  </dependencies>
```

The `dependencies` section of the POM declares library dependencies of the module. In this case, the new service requires `jonas-commons` because it extends the `AbsServiceImpl` class (that is in the `jonas-commons` module). The `scope` element is used to limit the transitive maven dependency resolution mechanism.

```
      <!-- Creates the bundle -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions> <!-- Do not forget the extensions element ! -->
        <configuration>
          <instructions>
            <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include>
          </instructions>
        </configuration>
      </plugin>
```

The `build/plugins` section contains the definition (and configuration) of all the plugin(s) that will be invoked during the build.

This snippet show the configuration to be used for the maven-bundle-plugin (in charge of the manifest and bundle generation): it simply states that it must read a `.bnd` file (located in `src/main/resources/META-INF/***.bnd`) named after the project's artifactId value. If this file is not present, default values [http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html#ApacheFelixMavenBundlePlugin%28BND%29-defaultbehavior] will be used.

```
      <!-- Manipulate the bundle (iPOJO) -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-ipojo-plugin</artifactId>
        <version>${ipojo.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>ipojo-bundle</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

This important section hooks iPOJO in the build process. It will reads the `src/main/resources/metadata.xml` file as input. Then, it manipulate the bundle (modify the classes, update the manifest) and save it on disk.

1.2.2.1.3. iPOJO component descriptor (metadata.xml)

iPOJO is a Service Component Runtime aiming to simplify OSGi application development. You have to create the iPOJO component which will define your JOnAS service. This component must declare the provided and required services, the start/stop callback methods and the service properties.

The iPOJO metadata file should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ipojo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="org.apache.felix.ipojo"
       xsi:schemaLocation="org.apache.felix.ipojo http://felix.apache.org/ipojo/
schemas/1.4.0/core.xsd" >

  <component classname="com.company.jonas.myservice.internal.DefaultMyService"
             immediate="false">
    <provides specifications="com.company.jonas.myservice.IMyService" />

    <!-- Required dependencies -->
    <requires optional="false"
              specification="org.ow2.jonas.properties.ServerProperties">
      <callback type="bind" method="setServerProperties" />
    </requires>
```

```
    <!-- LifeCycle Callbacks -->
    <callback transition="validate" method="start" />
    <callback transition="invalidate" method="stop" />

    <!-- Configuration properties -->
    <properties propagation="true">
      <property name="property" method="setProperty" />
    </properties>
  </component>

</ipojo>
```

iPOJO component description

- The component *`classname`* represents the service implementation class
- The *`provides`* element list all OSGi services provided by the component, here only the service interface
- In this example, the component requires the *`ServerProperties`* service which will be injected to the component instance during activation
- Two callbacks are defined, one for the component validation during service startup and one for the component invalidation for the service shutdown
- Property names must match service properties defined in the `jonas.properties` file. The property value will be injected to the component instance via setters

1.2.2.1.4. Build execution

Simply type `mvn clean install` to launch the build, that will delete the `target/` directory (that contains only generated/compiled stuff) and then compile and package everything, including generating the bundle with its manifest and manipulating it with iPOJO.

```
>$ mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JOnAS Service 1.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.4.2:resources (default-resources) @ myservice ---
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.3:compile (default-compile) @ myservice ---
[INFO] Compiling 1 source file to /home/sauthieg/sandboxes/playground/myservice/target/
classes
[INFO]
[INFO] --- maven-resources-plugin:2.4.2:testResources (default-testResources) @ myservice
 ---
[INFO] skip non existing resourceDirectory /home/sauthieg/sandboxes/playground/myservice/
src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3:testCompile (default-testCompile) @ myservice ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.5:test (default-test) @ myservice ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-bundle-plugin:2.1.0:bundle (default-bundle) @ myservice ---
[INFO]
[INFO] --- maven-ipojo-plugin:1.4.0:ipojo-bundle (default) @ myservice ---
[INFO] Start bundle manipulation
[INFO] Metadata file : /home/sauthieg/sandboxes/playground/myservice/target/classes/
metadata.xml
[INFO] Input Bundle File : /home/sauthieg/sandboxes/playground/myservice/target/
myservice-1.0.0.jar
[INFO] Bundle manipulation - SUCCESS
[INFO]
[INFO] --- maven-install-plugin:2.3:install (default-install) @ myservice ---
[INFO] Installing /home/sauthieg/sandboxes/playground/myservice/target/myservice-1.0.0.jar
 to /home/sauthieg/.m2/repository/com/company/jonas/myservice/1.0.0/myservice-1.0.0.jar
[INFO]
```

```
[INFO] --- maven-bundle-plugin:2.1.0:install (default-install) @ myservice ---
[INFO] Installing com/company/jonas/myservice/1.0.0/myservice-1.0.0.jar
[INFO] Writing OBR metadata
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 8.088s
[INFO] Finished at: Tue Jul 06 10:37:43 GMT+01:00 2010
[INFO] Final Memory: 9M/315M
[INFO] ------------------------------------------------------------------------
```

1.2.3. Preparing Service Installation

1.2.3.1. Bundle Repository

The generated bundle has to be placed in a maven repository that is available to the JOnAS instance that will host the service.

2 simples solutions:

- Bundle was locally produced using maven, so it's already available in the local developer's repository (usually in `~/.m2/repository/`). This repository has to be declared to the JOnAS instance.

  The easiest way to declare a repository is to edit the `$JONAS_BASE/conf/initial-repositories.xml` and make sure that a similar entry is present:

  ```
  <repository id="maven2-local-repository">
    <type>maven2</type>
    <url>file:///home/john/.m2/repository</url>
  </repository>
  ```

- Bundle is not already present in a maven repository. It's necessary to place it in the appropriate location (according to its maven coordinates: groupId, artifactId, version, ...):

  ```
  $JONAS_ROOT/repositories/maven2-internal/<groupId>/<artifactId>/<version>/<artifactId>-
  <version>.jar
  ```

  

  Note

  groupId is expressed with '.' as separator in POMs, but '.' are to be replaced with '/' in the location

  The path follows the Maven repository structure. By default, each JOnAS service is located in the `$JONAS_ROOT/repositories/maven2-internal` directory. This directory can be seen as a Maven local repository where JOnAS looks for OSGi™ bundles

1.2.3.2. Service Deployment Plan

After the setup of the maven repository hosting the new service bundle, a deployment plan (XML file) referencing this resource has to be created. This deployment plan must be placed in the `$JONAS_ROOT/repositories/url-internal` directory . The file name must be the same than the service name: `myservice.xml` in the example.

Important

Deployment plan filename HAS TO BE equals to the service name used in the `jonas.properties`.

Here is an example of deployment plan that will trigger the installation of a bundle available as a maven artifact.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment-plan xmlns="http://jonas.ow2.org/ns/deployment-plan/1.0"
                 xmlns:m2="http://jonas.ow2.org/ns/deployment-plan/maven2/1.0"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="deployment-plan-1.0.xsd"
                 atomic="false">

  <deployment id="com.company.jonas:myservice:jar"
              xsi:type="m2:maven2-deploymentType"
              reloadable="false"
              start="true"
              reference="true"
              startlevel="1"
              starttransient="true">
    <m2:groupId>com.company.jonas</m2:groupId>
    <m2:artifactId>myservice</m2:artifactId>
    <m2:version>1.0.0</m2:version>
  </deployment>

</deployment-plan>
```

As the service name and the deployment plan name are equal, JOnAS will automatically try to deploy this deployment plan during the service startup. This will trigger the deployment of the OSGi™ bundle.

More information about deployment plans here [deployment-plans_guide.html].

1.2.4. Modifying the jonas.properties file

The service is defined and its initialization parameters specified in the `jonas.properties` file. First, choose a name for the service (e.g. "`myservice`"), then do the following:

- add this name to the `jonas.services` property; this property defines the set of services (comma-separated) that will be started with JOnAS.
- add a `jonas.service.myservice.class` property specifying the service implementation class.
- add `jonas.service.myservice.XXX` properties which specify the service configuration. These properties will be set to the implementation class before the service startup.

This is illustrated as follows:

```
jonas.services                  .......,myservice
jonas.service.myservice.class       com.company.jonas.myservice.internal.DefaultMyService
jonas.service.myservice.property    value
```

1.3. Using the new service

When started, the new JOnAS service will espose an OSGi service as defined in the iPOJO component declaration which will be accessible through the OSGi registry. The specification of this service is defined by the Java interface and could be concretely looked up by specifying the *com.company.jonas.myservice.IMyService* interface. There are many ways to get the OSGi service reference:

1. Getting the OSGi service using the BundleContext:

```
BundleContext bundleContext = …
ServiceReference serviceReference =
 bundleContext.getServiceReference(IMyService.class.getName());
IMyService myService = (IMyService) bundleContext.getService(serviceReference);
```

2. Getting the OSGi service using iPOJO. iPOJO components can declare service requirements which will be dynamically injected to the component instance.

3. Using another Service Component Runtime (Declarative Service, Dependency Manager, ...)

1.4. Advanced understanding

Refer to the JOnAS sources for more details about the classes mentioned in this section.

1.4.1. JOnAS built-in services

The existing JOnAS services are the following:

Service name	**Service class**
registry	CarolRegistryService
jmx	JOnASJMXService
wc	JOnASWorkCleanerService
wm	JOnASWorkManagerService
ejb2	JOnASEJBService
ejb3	EasyBeansService
versioning	VersioningServiceImpl
web	Tomcat6Service / Jetty6Service
jaxrpc	AxisService
wsdl-publisher	DefaultWSDLPublisherManager
jaxws	CXFService / Axis2Service
ear	JOnASEARService
dbm	JOnASDataBaseManagerService
jtm	JOTMTransactionService
mail	JOnASMailService
resource	JOnASResourceService
security	JonasSecurityServiceImpl
discovery	JgroupsDiscoveryServiceImpl / MulticastDiscoveryServiceImpl
cmi	CmiServiceImpl
ha	HaServiceImpl
depmonitor	DeployableMonitorService
resourcemonitor	JOnASResourceMonitorService
smartclient	SmartclientServiceImpl

1.4.2. The ServiceException

The `org.ow2.jonas.service.ServiceException` exception is defined for Services. Its type is `java.lang.RuntimeException` and it can encapsulate any `java.lang.Throwable` .