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 a.b.myservice;

public interface MyServiceInterface {
    void doSomething();
}
```

The service class should look like the following:

```
package a.b.myservice.internal;

import a.b.myservice.MyServiceInterface;

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

.....

public class MyService extends AbsServiceImpl implements MyServiceInterface {
    private String property;

    public void doStart() throws ServiceException {
        ....
    }

    public void doStop() throws ServiceException {
        ....
    }

    public void doSomething() {
        ....
    }

    public String getProperty() {
        return property;
    }

    public void setProperty1(final String property) {
        this.property1 = property;
    }
}
```

1.2.2. Defining the service 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 make easier the service creation, configuration and management of possible dependencies with other JOnAS services, it is recommanded to use iPOJO [http://felix.apache.org/site/apache-felix-ipojo.html] to build your JOnAS services. This guide will present the iPOJO solution.

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

a/b/myservice	contains the service interface(s)
a/b/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: a.b.myservice
Private-Package: a.b.myservice.internal
Bundle-Version: 5.1.0
Bundle-Name: MyService
Bundle-SymbolicName: a.b.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.

1.2.3. 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. "serv1"), 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.serv1.class` property specifying the service implementation class.
- add `jonas.service.serv1.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                  .......,serv1
jonas.service.serv1.class       a.b.myservice.internal.MyService
jonas.service.serv1.property    value
```

1.2.4. Defining the iPOJO metadata file

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 and 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/1.2.0/core.xsd" >

  <component classname="a.b.myservice.internal.MyService"
             immediate="false">
    <provides specifications="a.b.myservice.MyServiceInterface" />

    <!-- 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" />

    <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

1.2.5. Adding the new service to JOnAS

1. Once the OSGi bundle (JAR file) containing the new service has been built, place it into the proper location:

   ```
   $JONAS_ROOT/repositories/maven2-internal/a/b/myservice/1.0.0/myservice-1.0.0-ipojo.jar
   ```

   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. In order to complete the bundle path, you have to concat the Maven artifact attributes which are the package name (groupId: *a.b*), the service name (artifactId: *myservice*), the version (1.0.0) and finally the service name, the version and the classifier (ipojo).

2. Then you have to create a Maven2 deployment plan (XML file) referencing this resource. 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: `serv1.xml` in our example and should look like the following:

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

     <deployment id="a.b.myservice:myservice:jar:ipojo" xsi:type="m2:maven2-deploymentType"
    reloadable="false" start="true" reference="true" startlevel="1" starttransient="true">
       <m2:groupId>a.b</m2:groupId>
       <m2:artifactId>myservice</m2:artifactId>
       <m2:version>1.0.0</m2:version>
       <m2:type>jar</m2:type>
       <m2:classifier>ipojo</m2:classifier>
     </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].

1.3. 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 *a.b.myservice.MyServiceInterface* interface. There are many ways to get the OSGi service reference:

1. Getting the OSGi service using OSGi instructions:

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