Chapter 1. Deploying OSGi^{TM} Configurations

Table of Contents

1.1. OSGi Configuration Admin

Configuration Admin[1] is a specification produced by the OSGi^{TM} Alliance[2] that defines how to configure OSGi services in a standard way.

A `Configuration` is basically a `Map` storing key/value. Typically, a management application query the ConfigurationAdmin service for Configuration(s), set the appropriate key/value pairs and update the Configuration. It is them take in charge by the ConfigurationAdmin service that will push the configurations to the interested components.

What is important to understand is that the `ConfigurationAdmin` service is an intermediate tier between the management application (pushing Configurations) and the managed system (receiving Configurations). A `Configuration` targets an OSGi service using the persitent identifier of that service (`PID`). If the target service is not available when the `Configuration` is pushed to the `ConfigurationAdmin` service, the `Configuration` is put on hold and persisted somehow until the service become available. At this moment in time, the `Configuration` is restored and applied to the managed element (the target service), it is said to be "bound to the service".

1.1.1. Managed components

Two kinds of components are recognised by ConfigurationAdmin: `ManagedService` and `ManagedServiceFactory`. They have to be registered in the OSGi^{TM} service registry under one of theses 2 interfaces.

1.1.1.1. ManagedService

A `ManagedService` is an already existing service instance that can be configured (and updated) through ConfigurationAdmin.

It just have to implements the following interface (and be exposed under it when registered as a service):

```
public interface ManagedService {
  /**
```

[1] http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/ConfigurationAdmin.html
[2] http://www.osgi.org

```
   * Update the configuration for a Managed Service.
   * @param properties A copy of the Configuration properties, or null.
   */
  void updated(Dictionary properties) throws ConfigurationException;
}
```

The `Dictionary` parameter contains the updated configuration properties. The implementation have to look for specific (application defined) properties and uses the queried values for configuring its internal state. The implementation may throw a `ConfigurationException` if something goes wrong.

1.1.1.2. ManagedServiceFactory

The other kind of component is a `ManagedServiceFactory`. A `ManagedServiceFactory` instance will receive Configurations not for itself, but in the goal of creating instances (may or may not be OSGi™ services themselves). A factory has to implement the following interface:

```
public interface ManagedServiceFactory {
  /**
   * Return a descriptive name of this factory.
   */
  String getName();

  /**
   * Create a new instance, or update the configuration of an existing instance.
   * @param pid The PID for this configuration.
   * @param properties A copy of the configuration properties.
   */
  void updated(String pid, Dictionary properties) throws ConfigurationException;

  /**
   * Remove the factory instance associated with the PID.
   * @param pid the PID of the service to be removed.
   */
  void deleted(String pid);
}
```

The `updated()` method will be called by the `ConfigurationAdmin` service when it will receive a new `Configuration` (or if an existing matching `Configuration` has been updated).

1.1.2. Identification

A `Configuration` contains both the configuration properties that have to be pushed to a consumer and an identifier that the `ConfigurationAdmin` service will use to find a matching component (the Configuration's consumer).

The identifier to use is the service PID (See Constants.SERVICE_PID[3]) of the component that will receive the `Configuration`.

1.1.3. Asynchronous

Once the `Configuration` is updated by the management application, everything happen asynchronously. That means that another Thread will be used to push the `Configuration` to its consumer (when a matching factory/service will be available).

1.2. Usage

JOnAS supports deploying ConfigurationAdmin Configurations through its *ConfigAdminDeployer* that is an extension of the deployment system.

Every XML file with its root element declared within the configadmin namespace (`http://jonas.ow2.org/ns/configadmin/1.0`) will be handled by this deployer.

[3] http://www.osgi.org/javadoc/r4v42/org/osgi/framework/Constants.html#SERVICE_PID

Example 1.1. ConfigurationAdmin XML file example

```
<configadmin xmlns="http://jonas.ow2.org/ns/configadmin/1.0">

  <configuration pid="service.pid.of.a.ManagedService">
    <property name="name">Guillaume</property>
    <property name="role">Developer</property>
  </configuration>

  <factory-configuration pid="service.pid.of.a.ManagedServiceFactory">
    <property name="name">OW2 JOnAS</property>
    <property name="category">Java EE</property>
  </factory-configuration>

</configadmin>
```

This generic format allows to produce both simple and factory `Configurations` without interacting with the ConfigurationAdmin API programmatically. The deployer handles mapping between the XML format and the ConfigurationAdmin API model.

Note

Only `java.lang.String` properties are supported right now.

Some component model convert from `java.lang.String` to the desired type automatically (acting as a type adapter between ConfigurationAdmin and the real component), but it is not mandatory.

1.2.1. Configuration

A <configuration> element represents the configuration to be applied to a `ManagedService` instance.

The `pid` attribute is the service PID of the targeted `ManagedService` (match the `service.pid` service property of a registered OSGi^{TM} service).

Every key/value pair is expressed using a <property> element. The name attribute denotes (indeed) the property name used as a key in the `Configuration`'s `Dictionary`. The text node's value is used as the key's value.

When the configuration XML file is undeployed, all the Configurations are deleted, ant the `ManagedService.updated()` method is called with a `null` argument. The `ManagedService` instance is then responsible to take the appropriate actions.

Example 1.2. Example of a <configuration>

```
<configuration pid="service.pid.of.a.ManagedService">
  <property name="name">Guillaume</property>
  <property name="role">Developer</property>
</configuration>
```

1.2.2. Factory Configuration

A <factory-configuration> element represents the configuration to be applied to a `ManagedServiceFactory` instance.

The `pid` attribute is the service PID of the targeted `ManagedServiceFactory` (match the `service.pid` service property of a registered OSGi^{TM} service).

Every key/value pair is expressed using a `<property>` element. The `name` attribute denotes (indeed) the property name used as a key in the Configuration's Dictionary. The text node's value is used as the key's value.

Every <factory-configuration> will trigger the creation of a *new instance* by the targeted service factory.

When the configuration XML file is undeployed, all the created instances are stopped and destroyed.

Example 1.3. Example of a <factory-configuration>

```
<factory-configuration pid="org.ow2.jonas.web.tomcat7.internal.Tomcat7AjpConnector">
  <property name="address">localhost</property>
  <property name="port">9011</property>
</factory-configuration>
```

1.3. Extensibility

JOnAS has, out of the box, a generic ConfigurationAdmin support (ie <configuration> and <factory-configuration>). Whilst this support can handle everything that can be described through ConfigurationAdmin, it may be a bit verbose and non user-friendly.

Here are some of the drawbacks of this approach:

- The user have to know before hand the exact service PID to use.
- Impossible to know in advance what is the set of required properties. It's possible to push an invalid configuration for example.
- Property typing: a property could be typed (Integer / String / ...) and with a generic schema it's possible to select the wrong type for a given property, leading to later Exceptions when the Configuration is actually applied to the service..

1.3.1. Expressiveness

The extensibility layer supported by the ConfigurationAdmin deployer allows to use different schema element inside the XML configuration file. Theses schema will provides a dedicated format/syntax for a specific domain, something like an XML DSL (Domain Specific Language).

Example 1.4. Mail Configuration Example

```
<configadmin xmlns="http://jonas.ow2.org/ns/configadmin/1.0"
             xmlns:mail="http://jonas.ow2.org/ns/configadmin/mail/1.0">

  <!-- This syntax is provided as an example only... -->
  <mail:session jndi-name="MailSession"
                from="no-reply@ow2.org">
    <transport name="smtp" />
    <store name="pop3"
           host="localhost" />
  </mail:session>

</configadmin>
```

1.3.2. APIs

The ConfigurationAdminDeployer provides a simple API for developers to extends the initial set of supported syntaxes.

1.3.2.1. XmlConfigurationAdapter

Most of the work is done in an `XmlConfigurationAdapter`. A `XmlConfigurationAdapter` is responsible to convert a DOM `Element` into one or multiple `ConfigurationInfo` (equivalent to the ConfigurationAdmin `Configuration` object).

This object will be given a DOM `Element` (child of the <configadmin> root element). The adapter will then read the XML format and produces one (or more) `ConfigurationInfo` objects, each of them representing one ConfigurationAdmin `Configuration`.

```
package org.ow2.jonas.configadmin;

public interface XmlConfigurationAdapter {

    /**
     * Convert the given Node into one or more ConfigurationInfo instance(s).
     * @param node DOM Element to be converted
     * @return Adapted ConfigurationInfo(s)
     * @throws AdapterException if the Element cannot be converted into ConfigurationInfo(s)
     */
    Set<ConfigurationInfo> convert(Element node) throws AdapterException;
}
```

1.3.2.2. XmlConfigurationAdapterRegistry

The `XmlConfigurationAdapterRegistry` contains the adapters *related to a namespace URI*. When an element from a namespace is found, the deployer select the dedicated registry and then ask it for the adapter supporting the current element's name.

Each implementation of this interface has to be exposed as an OSGi service with the '`namespace`' service property.

```
package org.ow2.jonas.configadmin;

public interface XmlConfigurationAdapterRegistry {

    /**
     * Returns the adapter handling the given Xml Element local-name.
     * @param name Xml Element local-name (not prefixed)
     * @return the associated adapter or null if none found
     */
    XmlConfigurationAdapter getAdapter(String name);
}
```

1.3.3. Example

This example will show how to add support of the <mail:session> extension element.

Apache Felix iPOJO[4] will be used to ease the implementation.

First, start with the adapter. Its code will take care of creating the internal `ConfigurationInfo` object. The adapter knows before hand the target Service PID that will accept the resulting configuration properties. The adapter then parse (or navigate through) the given DOM `Element` and extract the needed information from it, manipulating it if required (type convertion for example).

```
public class MailSessionAdapter implements XmlConfigurationAdapter {

    public static final String MAIL_SESSION_PID = "org.ow2...mail.Session";

    public Set<ConfigurationInfo> convert(Element node) throws AdapterException {

        // Create an empty configuration that will store the info from Xml
        // Hard-code the target PID
        ConfigurationInfo info = new ConfigurationInfo(MAIL_SESSION_PID, true);

        // Transfer from Xml format into internal format
        Map<String, Object> props = info.getProperties();
        props.put("jndi.name", node.getAttribute("jndi-name"));
        props.put("mail.from", node.getAttribute("from"));

        Element transport = getChildElement(node, "transport");
        props.put("mail.transport.protocol", transport.getAttribute("name"));

        // Remaining parsing omitted for brevity ...
```

[4] http://felix.apache.org/site/apache-felix-ipojo.html

```
        return Collections.singleton(info);
    }
}
```

Note

A simple implementation choice is to perform the Xml parsing by hand (navigating directly through the DOM structure), but it is also possible to use other parsing technics: XPath or JAXB[5] (useful when schema is pre-compiled before hand).

Once the adapter is finished, it is needed to create an `XmlConfigurationAdapterRegistry`. This one is the component that will be registered as an OSGi^{TM} service with a '`namespace`' service property.

```
@Component
@Provides
@Instantiate
public class MailAdapterRegistry implements XmlConfigurationAdapterRegistry {

    @ServiceProperty(value = "http://jonas.ow2.org/ns/example/mail")
    private String namespace;

    private XmlConfigurationAdapter sessionAdapter = new MailSessionAdapter();

    public XmlConfigurationAdapter getAdapter(String name) {
        if ("session".equals(name)) {
            return sessionAdapter;
        }
        return null;
    }
}
```