Developing a top-down Web Service project

www.javacodegeeks.com/2014/09/developing-a-top-down-web-service-project.html

Francesco Marchioni

This is a sample chapter taken from the Advanced JAX-WS Web Services book edited by Alessio Soldano.

The **bottom-up** approach for creating a Web Service endpoint has been introduced in the first chapter. It allows exposing existing beans as Web Service endpoints very quickly: in most cases, turning the classes into endpoints is a matter of simply adding few annotations in the code.

However, when developing a service with an already defined contract, it is far simpler (and effective) to use the **top-down** approach, since a **wsdl-to-java** tool can generate the annotated code matching the WSDL. This is the preferred solution in multiple scenarios such as the following ones:

- Creating a service that adheres to the XML Schema and WSDL that have been developed by hand up front;
- Exposing a service that conforms to a contract specified by a third party (e.g. a vendor that calls the service using an already defined set of messages);
- Replacing the implementation of an existing Web Service while keeping compatibility with older clients (the contract must not change).

In the next sections, an example of **top-down** Web Service endpoint development is provided, as well as some details on constraints the developer has to be aware of when coding, regardless of the chosen approach.

Creating a Web Service using the top-down approach

In order to set up a full project which includes a Web Service endpoint and a JAX-WS client we will use two Maven projects. The first one will be a standard webapp-javaee7 project, which will contain the Web Service Endpoint. The second one, will be just a quickstart Maven project that will execute a Test case against the Web Service.

Let's start creating the server project as usual with:

```
mvn -DarchetypeGroupId=org.codehaus.mojo.archetypes -DarchetypeArtifactId=webapp-
javaee7  -DarchetypeVersion=0.4-SNAPSHOT -
DarchetypeRepository=https://nexus.codehaus.org/content/repositories/snapshots  -
DgroupId=com.itbuzzpress.chapter2.wsdemo -DartifactId=ws-demo2 -Dversion=1.0 -
Dpackage=com.itbuzzpress.chapter2.wsdemo  -Darchetype.interactive=false --batch-mode
--update-snapshots archetype:generate
```

Next step will be creating the Web Service interface and stubs from a WSDL contract. The steps are similar to those for building up a client for the same contract. The only difference is that the *wsconsume* script will output the generated source files into our Maven project:

```
$ wsconsume.bat -k CustomerService.wsdl -o ws-demo-
wsdl\src\main\java
```

In addition to the generated classes, which we have discussed at the beginning of the chapter, we need to provide a **Service Endpoint Implementation** that contains the Web Service functionalities:

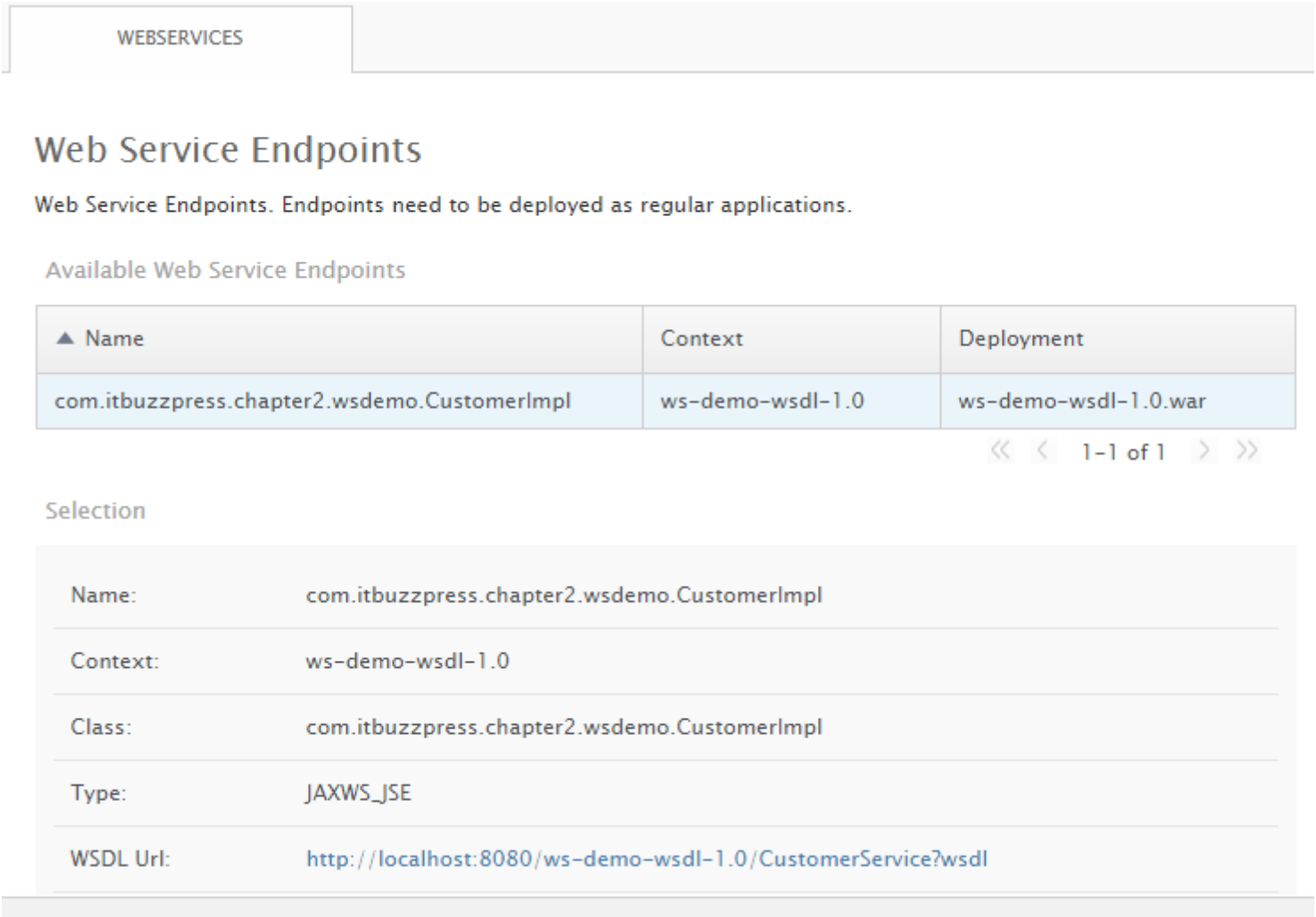
```
@WebService(endpointInterface="org.jboss.test.ws.jaxws.samples.webresult.Customer")
public class CustomerImpl implements Customer {
    public CustomerRecord locateCustomer(String firstName, String lastName, USAddress
address) {
        CustomerRecord cr = new CustomerRecord();
        cr.setFirstName(firstName);
        cr.setLastName(lastName);
        return cr;
    }
}
```

The endpoint implementation class implements the endpoint interface and references it through the *@WebService* annotation. Our WebService class does nothing fancy, just create a CustomerRecord object using the parameters received as input. In a real world example, you would collect the CustomerRecord using the Persistence Layer for example.

Once the implementation class has been included in the project, the project needs to be packaged and deployed to the target container, which will expose the service endpoint with the same contract that was consumed by the tool.

> *It is also possible to reference a local WSDL file in the @WebService* ***wsdlLocation*** *attribute in the Service Interface and include the file in the deployment. That would make the exact provided document be published.*

If you are deploying the Web Service to WildFly application server, then you can check from a management instrument like the Admin Console that the endpoint is now available. Select the Upper Runtime tab and click on the Web Services link contained in the left Subsystem left option:



Requirements of a JAX-WS endpoint

Regardless of the approach chosen for developing a JAX-WS endpoint, the actual implementation needs to satisfy some requirements:

- The implementing class must be annotated with either the j*avax.jws.WebService* or the *javax.jws.WebServiceProvider* annotation.
- The implementing class may explicitly reference a service endpoint interface through the *endpointInterface* element of the *@WebService* annotation but is not required to do so. If no *endpointInterface* is specified in *@WebService*, the service endpoint interface is implicitly defined for the implementing class.
- The business methods of the implementing class must be public and must not be declared static or final.
- The *javax.jws.WebMethod* annotation is to be used on business methods to be exposed to web service clients; if no method is annotated with *@WebMethod*, all business methods are exposed.
- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types.
- The implementing class must not be declared final and must not be abstract.
- The implementing class must have a default public constructor and must not define the finalize method.
- The implementing class may use the *javax.annotation.PostConstruct* or the *javax.annotation.PreDestroy* annotations on its methods for lifecycle event callbacks.

Requirements for building and running a JAX-WS client

A JAX-WS client can be part of any Java project and is not explicitly required to be part of a JAR/WAR archive deployed on a JavaEE container. For instance, the client might simply be contained in a quickstart Maven project as follows:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -
DarchetypeArtifactId=maven-archetype-quickstart -
DgroupId=com.itbuzzpress.chapter2.wsdemo -DartifactId=client-demo-wsdl -Dversion=1.0
-Dpackage=com.itbuzzpress.chapter2.wsdemo -Dversion=1.0 -Darchetype.interactive=false
--batch-mode
```

As your client needs to reference the endpoint interface and stubs, you need to provide them either copying them from the server project or generating them again using wsconsume:

```
$ wsconsume.bat -k CustomerService.wsdl -o client-demo-
wsdl\src\main\java
```

Now include a minimal Client Test application, which is part of a JUnit test case:

```
public class AppTest extends TestCase {

 public void testApp()  {
  CustomerService service = new CustomerService();
  Customer port = service.getCustomerPort();
  CustomerRecord record = port.locateCustomer("John", "Li", new USAddress());
  System.out.println("Customer record is " +record);
  assertNotNull(record);
 }
}
```

Compiling and running the test

In order to run successfully running a WS client application, a classloader needs to be properly setup to include the desired JAX-WS implementation libraries (and the required transitive dependencies, if any). Depending on the environment the client is meant to be run in, this might imply adding some jars to the classpath, or adding some artifact dependencies to the Maven dependency tree, setting the IDE properly, etc.

Since Maven is used to build the application containing the client, you can configure your ***pom.xml*** as follows so that it includes a dependency to the JBossWS:

```
<dependency>
    <groupId>org.jboss.ws.cxf</groupId>
    <artifactId>jbossws-cxf-client</artifactId>
    <version>4.2.3.Final</version>
    <scope>provided</scope>
</dependency>
```

Now, you can execute the testcase which will call the JAX-WS API to serve the client invocation using JBossWS.

```
mvn clean package
test
```

Focus on the JAX-WS implementation used by the client

The JAX-WS implementation to be used for running a JAX-WS client is selected at runtime by looking for ***META-INF/services/javax.xml.ws.spi.Provider*** resources through the application classloader. Each JAX-WS implementation has a library (jar) including that resource file which internally references the proper class implementing the JAX-WS SPI Provider.

On WildFly 8.0.0.Final application server the JAX-WS implementation is contained in the ***META-INF/services/javax.xml.ws.spi.Provider*** of the file *jbossws-cxf-factories-4.2.3.Final*:

```
org.jboss.wsf.stack.cxf.client.ProviderImpl
```

Therefore, it is extremely important to control which artifacts or jar libraries are included in the classpath the application classloader is constructed from. If multiple implementations are found, order matters, hence the first implementation in the classpath will be used.

> *The safest way to avoid any classpath issue (and thus load another JAX-WS implementation) is to set the java.endorsed.dirs system property to include the jbossws-cxf-factories.jar; if you don't do that, make sure you don't include ahead of your classpath other META-INF/services/javax.xml.ws.spi.Provider resources which will trigger another JAX-WS implementation.*

Finally, if the JAX-WS client is meant to run on WildFly as part of a JavaEE application, the JBossWS JAX-WS implementation will be automatically selected for serving the client.

- Getting into the details of JAX-WS types: Java to XML mapping and XML to Java
- Developing SOAP Web services on WildFly 8 and Tomcat. Running native Apache CXF on WildFly.
- Securing Web services. Applying authentication policies to your services. Encrypting the communication.