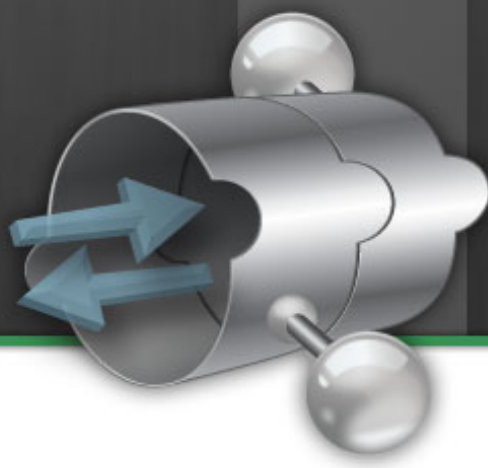
talend*
*open integration solutions



Talend ESB

Development Guide

5.1_b

Talend ESB: Development Guide

Publication date 5 July 2012

Copyleft

Notices

Table of Contents

Chapter 1. Introduction

This document looks at best practices in developing with Talend ESB, in particular using Eclipse and Maven as development tools. While development with Eclipse 3.6.1 is covered within this guide, note the Eclipse-based Talend Enterprise ESB Studio can also be used for these tasks, as it already includes the Eclipse plugins that we'll be covering within this guide.

In Chapter 2, *Eclipse Setup Overview* we provide links and explanations of the software products and where they can be downloaded. Different types of web services (JAX-WS and JAX-RS based) will be covered next in Chapter 3, *Web Services*, and then we'll explore Camel development in Chapter 4, *Camel Routes Overview*. Finally, developing with Talend ESB specific services such as Service Activity Monitoring and the Service Locator is covered in Chapter 5, *Talend ESB Services Overview*.

Chapter 2. Eclipse Setup Overview

This section shows how to install the Eclipse IDE and configure it with additional plugins useful for development of web services and Camel routing mechanisms. The Eclipse Helios (3.6.2) version is covered and tested here but you may wish to try working with later release versions of this IDE. Note if you already have a working Eclipse instance, or are working with the Talend Enterprise ESB Studio (an Eclipse-based product that has most helpful plugins already installed), this process can be skipped, except for any optional plugins you may wish to install.

Steps to configure an useful Eclipse environment for development of web services and Camel routes:

Procedure 2.1. Downloading and Configuring Eclipse

1. **Download Java SE 6**

   Java SE 6 can be obtained from: http://www.oracle.com/technetwork/java/javase/downloads/index.html. Be sure to download the JDK and not the JRE version.

2. **Download Eclipse Helios**

   From the download page of Eclipse Helios, download and extract the Eclipse IDE for Java EE Developers version of this IDE. The download page provides versions for Linux, Windows, and Mac OSX, in both 32-bit and 64-bit versions. After extracting the application, double-clicking the Eclipse icon located in the Eclipse root directory should bring up the IDE; note your specific distribution may provide additional convenient options (menu items, desktop icons) for activating Eclipse.

   After installing, have Eclipse point to the JDK you downloaded in the previous step (using menu item: Windows | Preferences, and from the Preferences Dialog, selecting Java | Installed JREs from the left-side menu tree.)

3. **Download Maven 3.0.3**

   Maven 3.0.3 is available from: http://maven.apache.org/download.html

   After the download is completed, extract the zip file to somewhere, for example, "C:\Maven3". You can change the default configuration by modifying the configuration file under "C:\Maven3\conf\settings.xml".

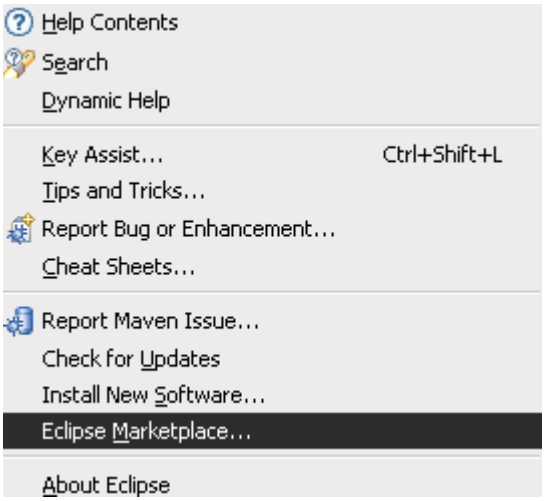
For more details on how to configure Maven, and best practices with Maven, please refer to http://maven.apache.org/users/index.html

4. **Install Maven2Eclipse Plugin**

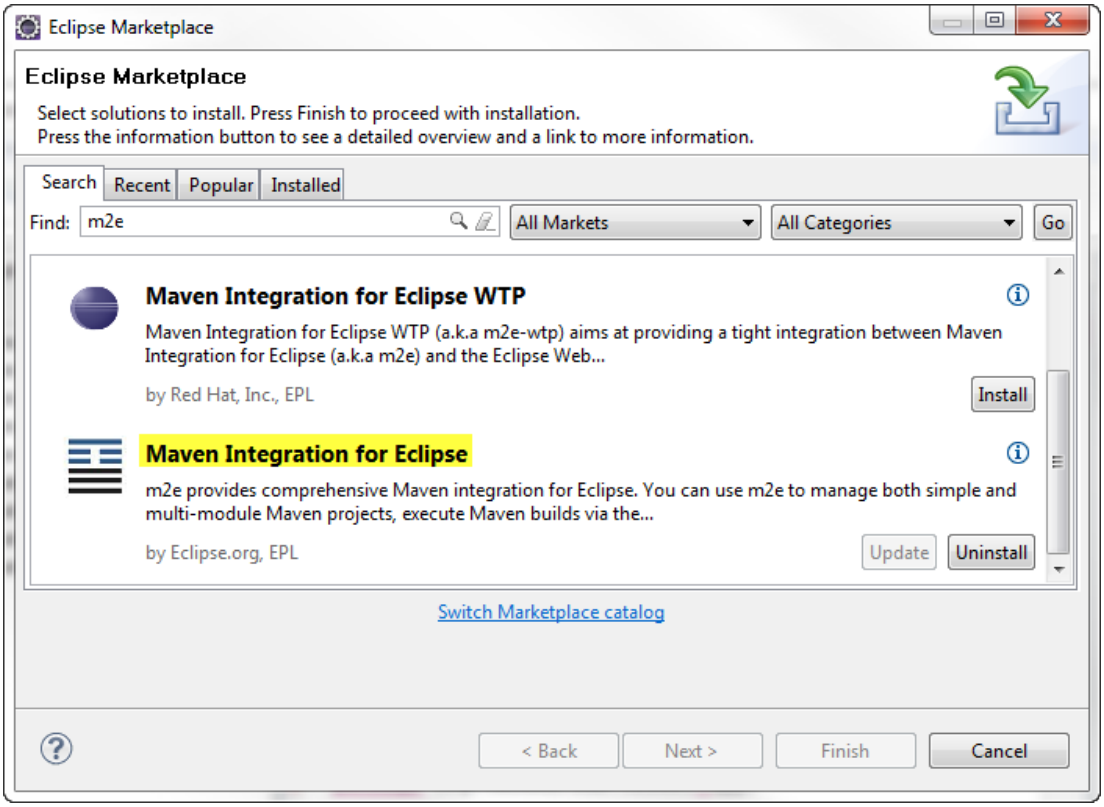
Maven is a very popular project management tool that can be run either directly from the Eclipse IDE itself, using the m2e plugin or from a separate command-line window (while still using Eclipse for coding/software development) using the regular Maven download. (Talend Enterprise ESB Studio already includes the m2e plugin.) There are several ways to install the m2e plug-ins into Eclipse. An easy way is using Eclipse Marketplace. In order to install m2e, you can:

1. Start Eclipse

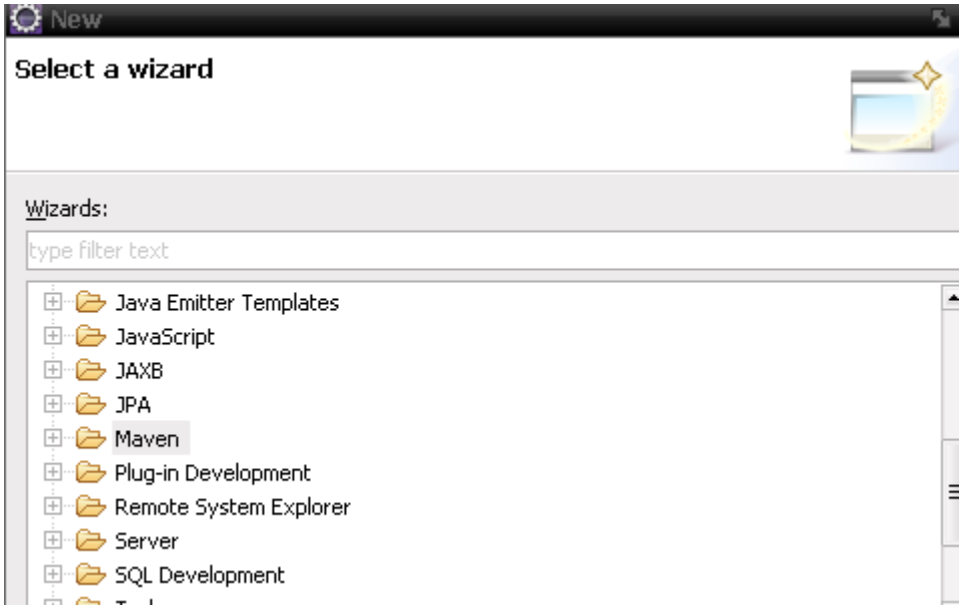
2. From Main Menu, select: Help -> Eclipse Marketplace...



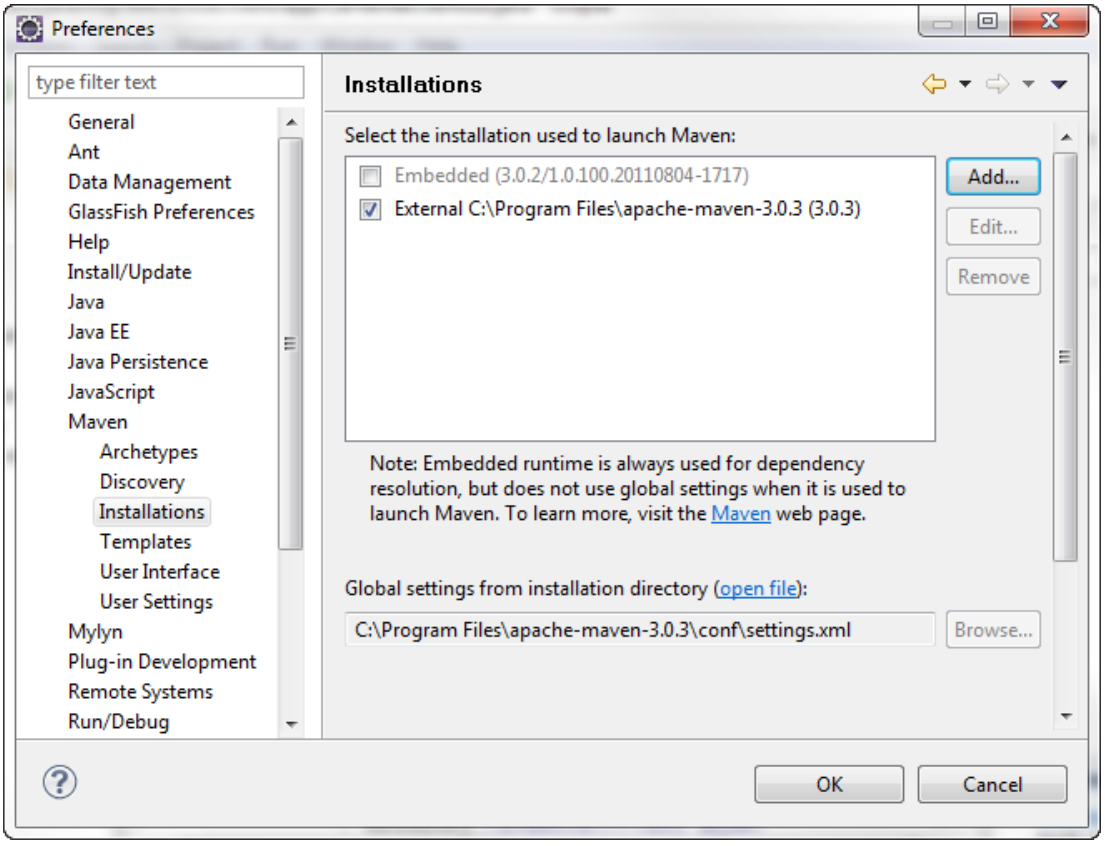
3. Then Search for m2e in the popup dialog



4. Click on the "Install" button below, the m2e will be installed. Then restart the IDE. It is now available and can be seen by selecting "File->New->Other":



By default Eclipse's embedded Maven is used. However, it's recommended to use the external Maven 3.0.3 installed above. Open Main Menu "Window -> Preferences -> Maven -> Installations" to change the default setting:



Click Add, specify the path of where you installed Maven, and click Ok

5. **Install soapUI Plugin**

   The soapUI SOAP/REST request/response tool provides an Eclipse plugin for convenient usage of this tool from the IDE. Talend Enterprise ESB Studio already includes this plugin by default. If you're using the standard Eclipse distributions, see the soapUI plugin page for instructions on how to install this tool into your IDE.

Chapter 3. Web Services

3.1. CXF Overview

Talend ESB helps you to create new web services or to service-enable your existing applications and interfaces for use with the Web, using technologies based on Apache CXF. CXF supports all important web services standards including the Java API for XML Web Services (JAX-WS) and Java API for RESTful Web Services (REST) specifications. JAX-WS defines annotations that allow you to define how your standalone Java application should be represented in a web services context. Three main styles of web services development with CXF are available:

1. Code-first development:

   Used in JAX-WS development, here we start out with a Java class and then let the web service framework handle the job of generating a WSDL contract for you. This is the easiest mode of development, but it also means that the tool (CXF in this case) is in control of what the contract will be. If you wish to have greater control over the WSDL, then contract-first approach is recommended; note you can also start with code to generate a WSDL and then modify that WSDL using the contract-first approach.

2. Contract-first development:

   Another JAX-WS option, this time a WSDL (Web Services Description Language) file is used to define the operations and types a web service provides. This file is often referred to as the web services contract, and in order to communicate with a web service, you must satisfy the contract. Contract-first development involves starting out by writing a WSDL file (either by hand or with the help of tooling), and then generating stub Java class implementations from the WSDL file by using tools such as those provided by CXF.

3. JAX-RS (REST) services:

   REST is a more recent paradigm for simpler HTTP-based services which takes advantage of HTTP verbs (GET, POST, PUT, DELETE), an intuitively designed http URL string, and (in some cases) HTTP message body for responses and requests. It's paradigm is so simple that frequently usage of a web browser alone is sufficient to

make and receive REST calls, however REST is not up to the level of providing the advanced WS-* support (security and reliability) available with JAX-WS.

We look at how to do development using these models in Section 3.2, “Code-first development”, Section 3.7, “Contract-first development”, and Section 3.8, “REST Services”. A general flowchart would be to determine the type of web service you're interested in developing (SOAP or REST). If SOAP, choose whether code-first or contract-first, and then finally the deployment environment (servlet container or OSGi) desired. Also note the Eclipse-based Talend Enterprise ESB Studio provides additional graphical options, such as a RouteBuilder, if less programmatic methods of service development are desired.

3.2. Code-first development

Code-first development means starting from existing code from which a WSDL can automatically generated. CXF's Java2ws tool, configured within the cxf-java2ws-plugin Maven plugin, is used for this process. For example, given a simple web service interface:

```
package service;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public interface DoubleItPortType {
  public int doubleIt(int numberToDouble);
}
```

The code first developer will implement the web service, adding annotations to indicate desired web service configuration information:

```
package service;

import javax.jws.WebService;

@WebService(targetNamespace = "http://www.example.org/contract/DoubleIt",
            endpointInterface = "service.DoubleItPortType",
            serviceName = "DoubleItService",
            portName = "DoubleItPort")
public class DoubleItPortTypeImpl implements DoubleItPortType {

   public int doubleIt(int numberToDouble) {
      return numberToDouble * 2;
   }

}
```

If the Maven pom.xml has the cxf-java2ws-plugin configured as follows:

```
<plugin>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-java2ws-plugin</artifactId>
   <version>${cxf.version}</version>
   <executions>
      <execution>
         <id>process-classes</id>
         <phase>process-classes</phase>
         <configuration>
            <className>service.DoubleItPortTypeImpl</className>
            <genWsdl>true</genWsdl>
            <verbose>true</verbose>
         </configuration>
         <goals>
            <goal>java2ws</goal>
         </goals>
      </execution>
   </executions>
</plugin>
```

An autogenerated two-part WSDL supporting this web service will be created, as shown below (certain areas truncated for brevity). The first file, `DoubleItPortTypeImpl.wsdl` in this example contains message input and output information as well as the generic wsdl:portType listing the method calls available. The wsdl:portType value incorporates the name of the web service interface. This interface also provides (from its doubleIt method) the name of the specific operation and its parameters.

```
<wsdl:definitions name="DoubleItPortType" targetNamespace="http://service/">
   <wsdl:types>
      <xs:schema elementFormDefault="unqualified"
         targetNamespace="http://service/" version="1.0">
         <xs:element name="doubleIt" type="tns:doubleIt" />
         <xs:element name="doubleItResponse" type="tns:doubleItResponse" />
         <xs:complexType name="doubleIt">
            <xs:sequence>
               <xs:element name="arg0" type="xs:int" />
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="doubleItResponse">
            <xs:sequence>
               <xs:element name="return" type="xs:int" />
            </xs:sequence>
         </xs:complexType>
      </xs:schema>
   </wsdl:types>
   <wsdl:message name="doubleIt">
      <wsdl:part name="parameters" element="ns1:doubleIt"/>
   </wsdl:message>
   <wsdl:message name="doubleItResponse">
      <wsdl:part name="parameters" element="ns1:doubleItResponse"/>
   </wsdl:message>
   <wsdl:portType name="DoubleItPortType">
      <wsdl:operation name="doubleIt">
         <wsdl:input name="doubleIt" message="ns1:doubleIt"/>
         <wsdl:output name="doubleItResponse"
            message="ns1:doubleItResponse"/>
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

The second file, `DoubleItPortType.wsdl` imports the former file and provides the explicit binding and service connection information. The wsdl:service incorporates the service name and port name values specified on the Java web service implementation above.

```
<wsdl:definitions name="DoubleItService"
   targetNamespace="http://www.example.org/contract/DoubleIt">
   <wsdl:import namespace="http://service/" location="DoubleItPortType.wsdl"/>
   <wsdl:binding name="DoubleItServiceSoapBinding"
      type="ns1:DoubleItPortType">
      <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="doubleIt">
         <soap:operation soapAction="" style="document" />
         <wsdl:input name="doubleIt">
            <soap:body use="literal" />
         </wsdl:input>
         <wsdl:output name="doubleItResponse">
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="DoubleItService">
      <wsdl:port name="DoubleItPort"
         binding="tns:DoubleItServiceSoapBinding">
         <soap:address location="http://localhost:9090/DoubleItPort" />
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

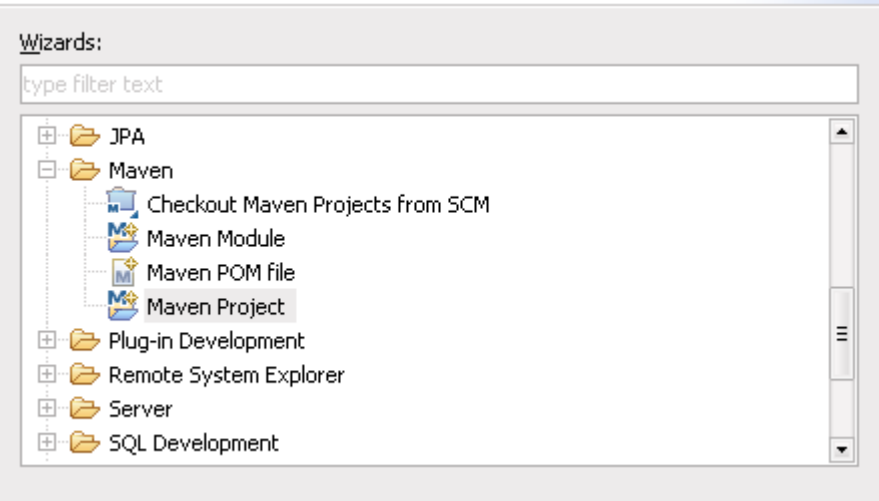
Prior to deploying the web service on the OSGi container two levels of testing should be done. The first would be testing of the DoubleItPortTypeImpl.java class itself without web service interaction. JUnit is a popular and recommended tool for this task. Next would be to test the web service's handling of actual SOAP calls from clients. CXF's internal web service container (embedded Jetty) can be used for this task.

"${Talend-ESB-Version}/examples/apache/cxf/java_first_jaxws" will be our example for code-first development, except we'll be starting from scratch. First, we'll need to generate our project skeleton.

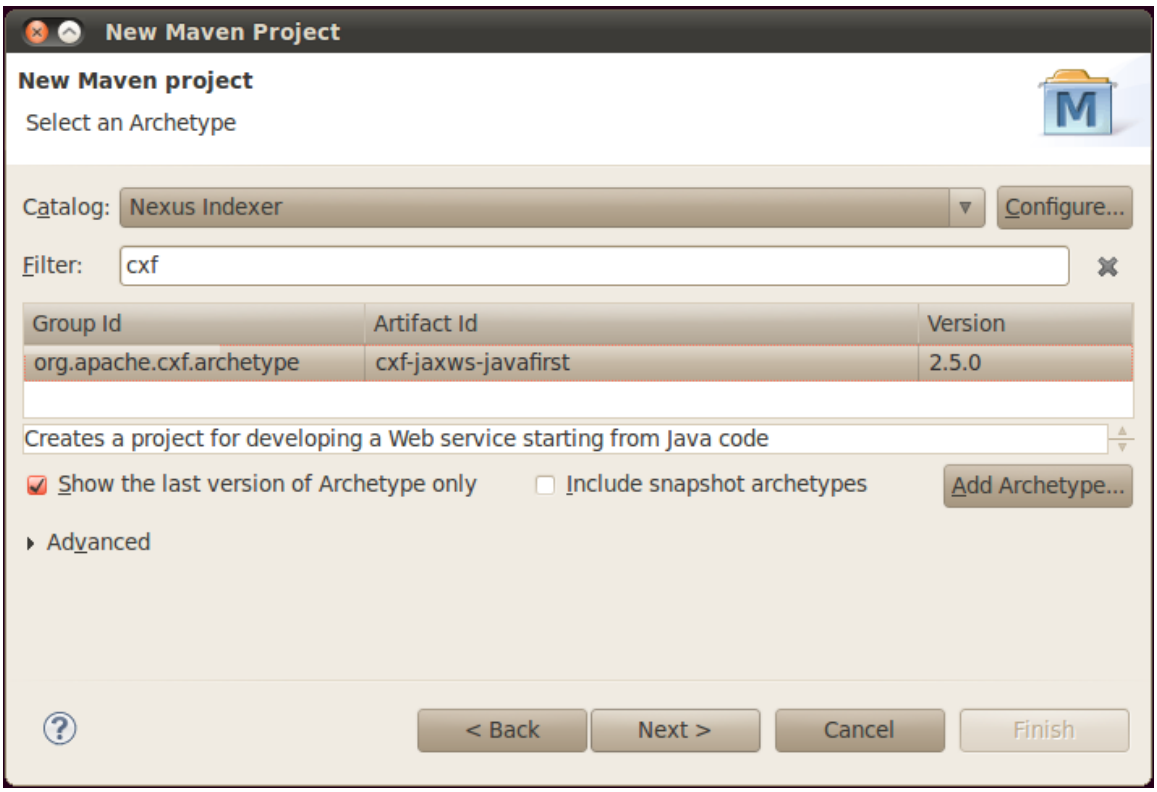
3.3. Project Skeleton

Since we have already installed the m2eclipse plug-in, let's use that to generate the project skeleton. Open Eclipse:

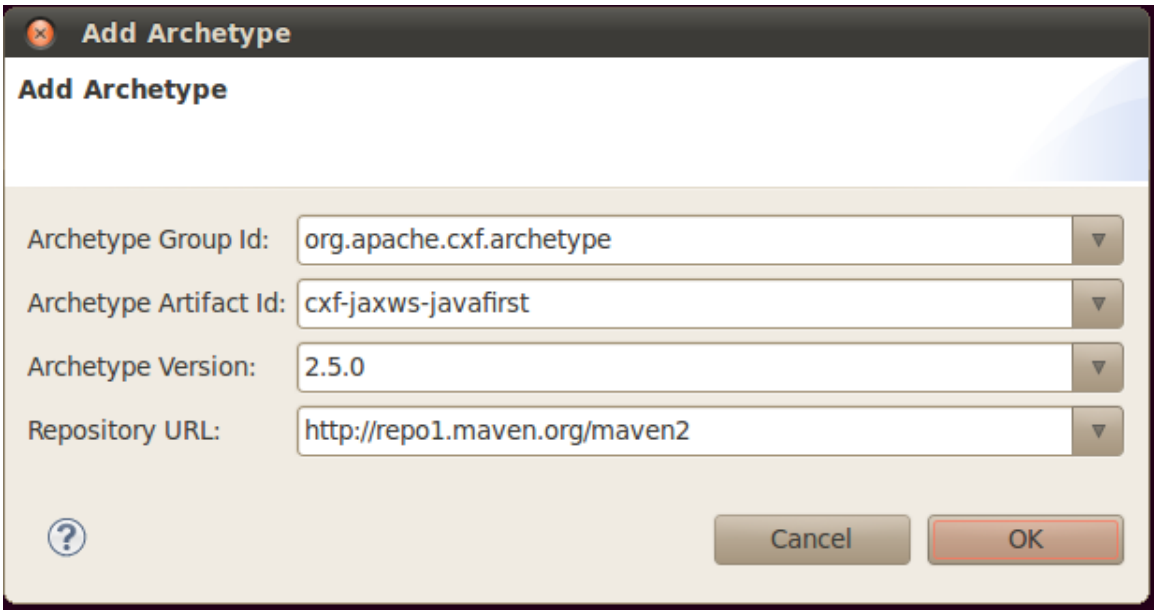
1. From Main Menu: File>New>Other, Select the "Maven Project" under "Maven" category:



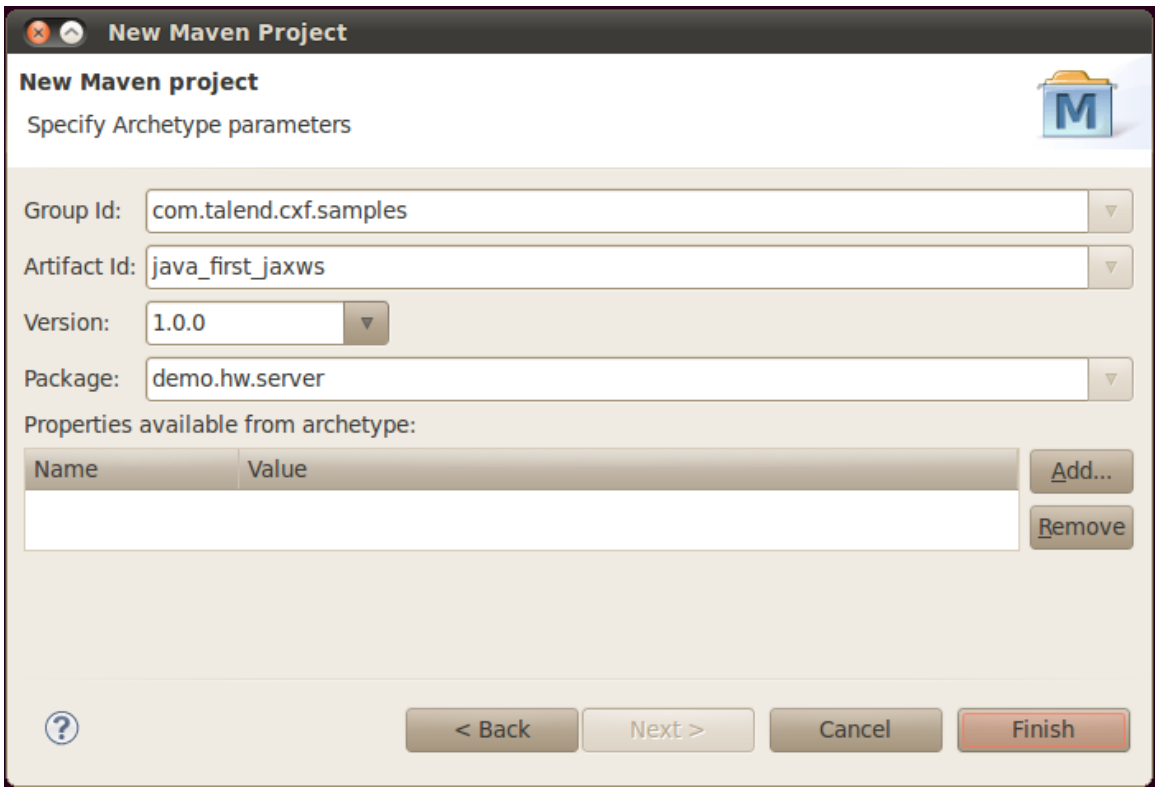
2. Then, Click "Next", "Next". On this page, to simplify the process, let's use "cxf-archetype" as the prototype:



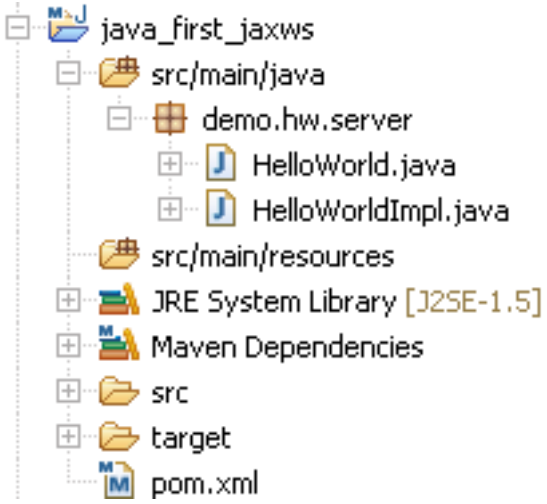
If the CXF archetype is not already available in your Eclipse installation, add it by selecting Add Archetype... with the following information:



3. Then, Click "Next". We need to fill the required fields, which we can do as below:

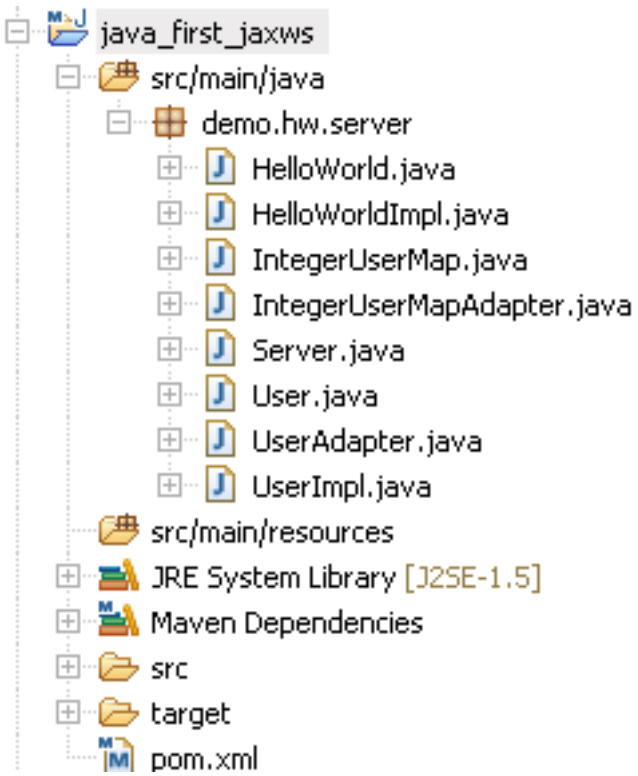


4. Click "Finish", the project skeleton will be generated with a structure as below:



We have the project skeleton that we can start from now. We want to use the code-first development in this section, so it's time to look at the code. As mentioned earlier, we'll illustrate this process by using the "java_first_jaxws" sample under `${Talend-ESB-Version}/examples/apache/cxf`.

The "HelloWorld.java" is the interface which we wish to implement. To save time, let's copy all files from `${Talend-ESB-Version}/examples/apache/cxf/java_first_jaxws/src/main/java/demo/hw/server` to the package "demo.hw.server", replacing even the HelloWorld and HelloWorldImpl classes already there. The resulting project structure is below:



As mentioned earlier, prior to OSGi deployment we should do unit testing of the service definition implementation (HelloWorldImpl.java) as well as testing the web service using an embedded server. For unit testing, this could be had by adding the following JUnit dependency to project's pom.xml:

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.8.2</version>
   <scope>test</scope>
</dependency>
```

Next, add the JUnit test case class under `src/test/java/demo/hw/server` in the project:

```
package demo.hw.server;

import org.junit.Test;
import static org.junit.Assert.assertEquals;
import java.util.Map;

public class HelloWorldImplTest {

   @Test
   public void testSayHi() {
      HelloWorldImpl hwi = new HelloWorldImpl();
      String response = hwi.sayHi("Bob");
      assertEquals("SayHi isn't returning expected string",
         "Hello Bob", response);
   }

   @Test
   public void testSayHiToUser() {
      HelloWorldImpl hwi = new HelloWorldImpl();
      User sam = new UserImpl("Sam");
      String response = hwi.sayHiToUser(sam);
      assertEquals("SayHiToUser isn't returning expected string",
         "Hello Sam", response);
   }

   @Test
   public void testGetUsers() {
      HelloWorldImpl hwi = new HelloWorldImpl();
      User mike = new UserImpl("Mike");
      hwi.sayHiToUser(mike);

      User george = new UserImpl("George");
      hwi.sayHiToUser(george);
      Map<Integer, User> userMap = hwi.getUsers();
      assertEquals("getUsers() not returning expected number of users",
         userMap.size(), 2);
      assertEquals("Expected user Mike not found", "Mike",
         userMap.get(1).getName());
      assertEquals("Expected user George not found", "George",
         userMap.get(2).getName());
   }

}
```

Finally run **`mvn test`** or **`mvn clean install`** (the latter command includes testing) from a command prompt window in this project's base folder to run the JUnit tests. Maven will show any test errors in the console output and also provide more detailed test information in a `target/surefire-reports` folder that it creates.

The java_first_jaxws example in the software distribution also shows a workable method of doing integration testing by configuring Maven profiles within the pom.xml, one for the service and the other for a test client. Once done, simply running the **`mvn -Pserver`** and **`mvn -Pclient`** commands from separate terminal windows will allow you to see the results of client requests against the web service provider. Once web service provider behavior is shown to be as expected we can deploy the web service on an OSGi container, as discussed next.

3.4. Deployment

This chapter shows how to deploy a web service as an OSGi bundle in Talend ESB. For more information about OSGi, please visit the OSGi home page athttp://www.osgi.org/Main/HomePage.

In order to deploy our service as a OSGi bundle, we need a OSGi container. In Talend ESB, Talend ESB is based on Apache Karaf, and is provided as the default OSGi container. For more details about Karaf, please visit http://karaf.apache.org/.

Now to create our OSGi bundle. Let's package "java_first_jaxws" as an OSGi bundle.

We can use Apache Felix in Maven to package the application as an OSGi bundle. For more information about Felix, please visit http://felix.apache.org/site/index.htm; for how to use Felix with Maven, please refer to http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html.

3.4.1. Package the application as a bundle

In order to package the application as a bundle, first we need to add the Felix Maven dependency:

```
<dependency>
   <groupId>org.apache.felix</groupId>
   <artifactId>org.osgi.core</artifactId>
   <version>1.4.0</version>
</dependency>
```

Also, we need to add the Felix plugin used for creating the bundle. Under element "project/build/plugins/" add:

```
        <plugin>
          <groupId>org.apache.felix</groupId>
          <artifactId>maven-bundle-plugin</artifactId>
          <extensions>true</extensions>
          <version>2.3.7</version>
          <configuration>
            <instructions>
              <Bundle-SymbolicName>${pom.groupId}.${pom.artifactId}</Bundle-SymbolicName>
              <Bundle-Name>${pom.name}</Bundle-Name>
              <Bundle-Version>${pom.version}</Bundle-Version>
              <Export-Package>demo.hw.server</Export-Package>
              <Bundle-Activator>demo.hw.server.Activator</Bundle-Activator>
            </instructions>
          </configuration>
        </plugin>
```

Since we want to package as an OSGi bundle, the package type is changed from `war` to `bundle`. We used Felix's maven-bundle-plugin for the packaging, and "demo.hw.server" is exported as the bundle name. In addition, a "Bundle-Activator" implementation is given. This allows us to activate other processes on bundle startup and shutdown. For bundle activation, instead of using the `Server.java` Java class, we'll start our service in the start method of Activator, and stop the service in the stop method of Activator:

Activator.java:

```
package demo.hw.server;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
   private Server server;

   public void start(BundleContext arg0) throws Exception {
      try {
         HelloWorldImpl implementor = new HelloWorldImpl();
         String address = "http://localhost:9000/helloWorld";
         Endpoint.publish(address, implementor);
         System.out.println("Server is started...");
      } catch (Exception e) {
         e.printStackTrace();
         throw e;
      }
   }

   public void stop(BundleContext arg0) throws Exception {
      try {
         server.stop();
      } catch (Exception e) {
         e.printStackTrace();
         throw e;
      }
   }
}
```

All changes needed have been made. Now it's time to package and deploy our service bundle:

Select the "Run As> Maven Clean" and then "Run As> Maven Install" from the popup menu on `pom.xml`. the application will be packaged and installed into your Maven Local Repository. You can go to "${MavenRepository}/com/talend/liugang/cxf/java_first_jaxws/1.0.0" to view the new bundle.

If your Maven location is not under the standard <user home>/.m2 location, check the *`localRepository`* field in your `${Maven_HOME}/conf/settings.xml` file.

3.5. Deploy the bundle

Once the application has been installed by Maven into your local Maven repository, it can be deployed into Karaf. You can find Karaf under "${Talend-ESB-Version}/container". Note you can have more than one container at one time, for more information, please refer to **Talend ESB Getting Started User Guide**.

Go into `${Talend-ESB-Version}/container/bin`, start the container by running `trun.bat` on Windows, or `trun` on Linux. When the container starts up, you will see a short introduction (similar to the one below) followed by the OSGi console command prompt:

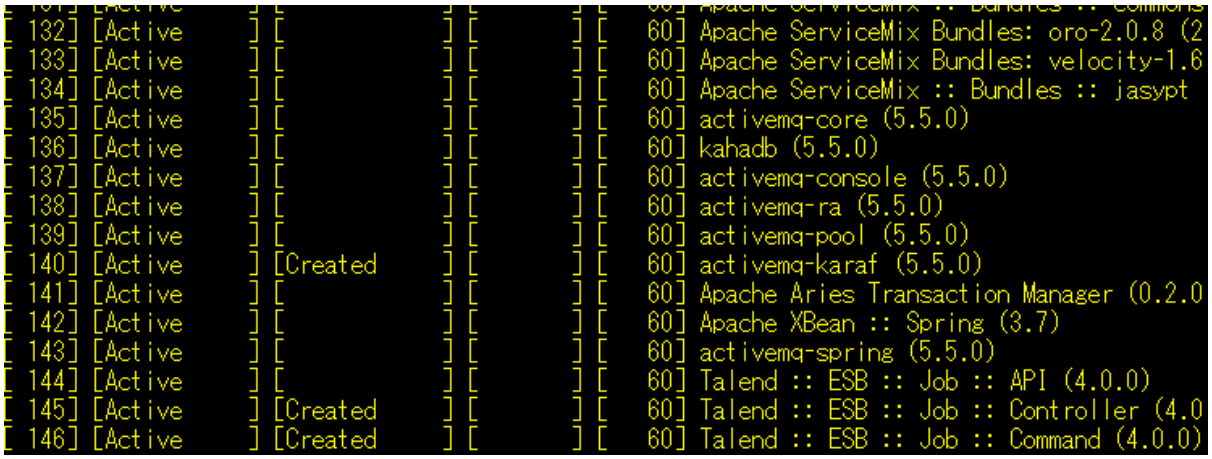
```
bsmith@bsmith-work:~/products/TESB_SE-V5.0.0/container/bin$ ./trun

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown the TRUN.

karaf@trun>
```

In order to check that the container has started up and activated all components, enter the command: **`list`**

You should get a listing like the following fragment:



You can then deploy the bundle by an install command similar to this one:

```
osgi:install mvn:com.talend.liugang.cxf/java_first_jaxws/1.0.0
```

For the above bundle, `mvn` refers to the protocol (`http://` and `file:/` are other common alternatives, while the remaining a/b/c portion refers to the Maven group ID, artifact ID, and version, respectively.)

Now, if you execute **`list`**, you can see that java_first_jaxws has been installed, and the status is `installed`:

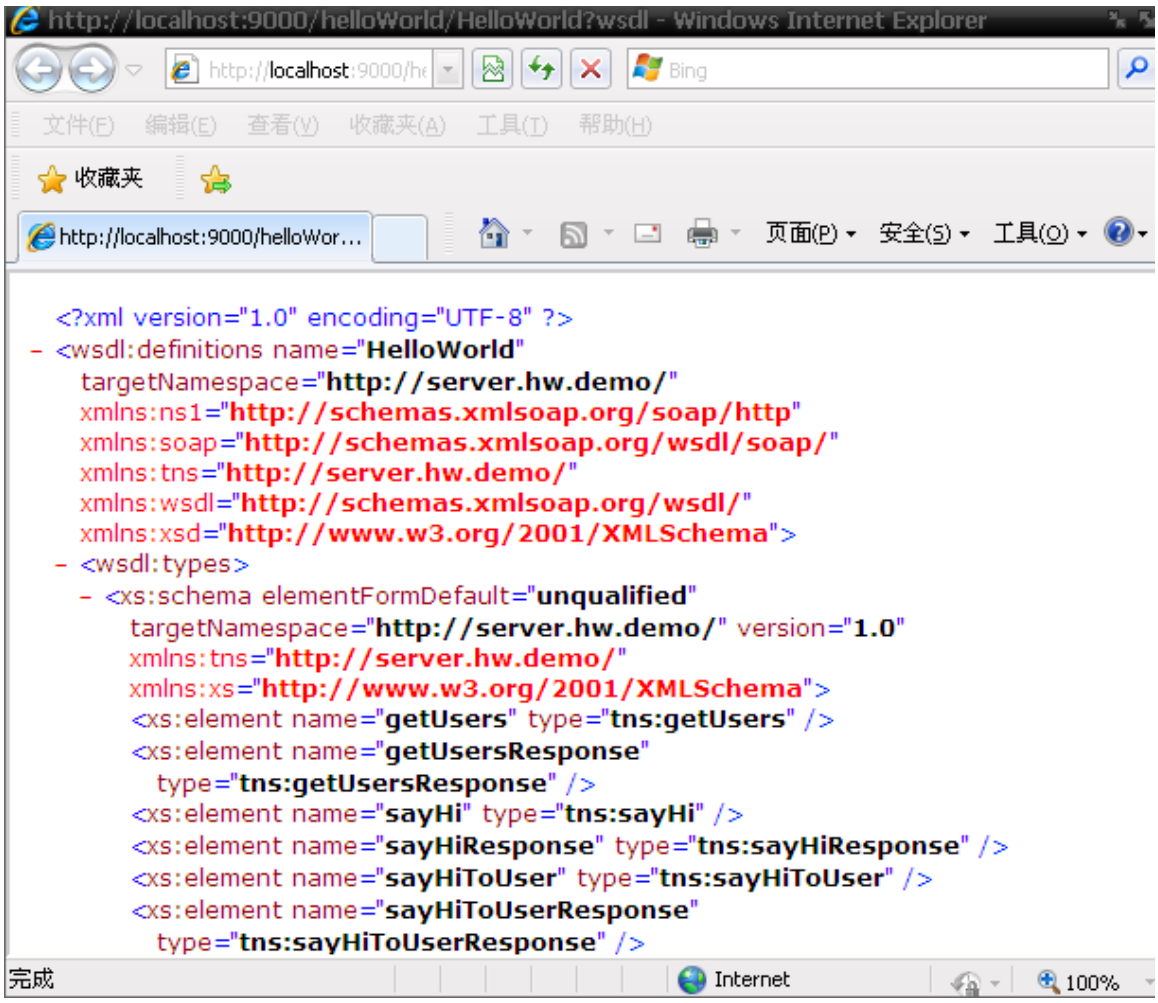
Then, we can start it by running **`osgi:start 154`**. If this is successful, you will see "Service is started..." and you can test the service.

For more information on commands within the OSGi scope, see **Talend ESB Container Administration Guide**.

3.6. Test the service

1. Assuming you have started the server already, you can check it by accessing http://localhost:9000/helloWorld/HelloWorld?wsdl

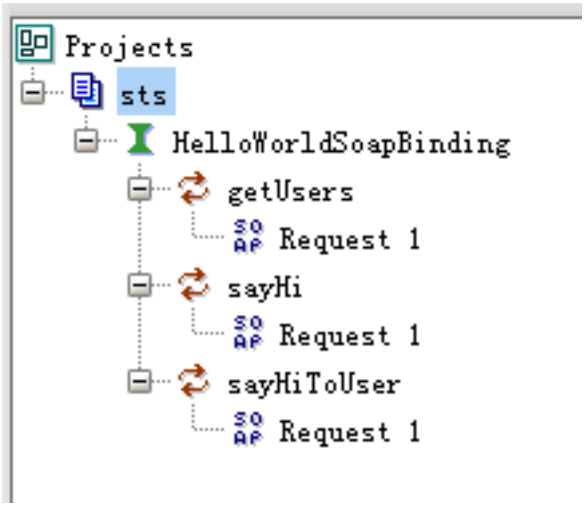
   You should see a response similar to the below:



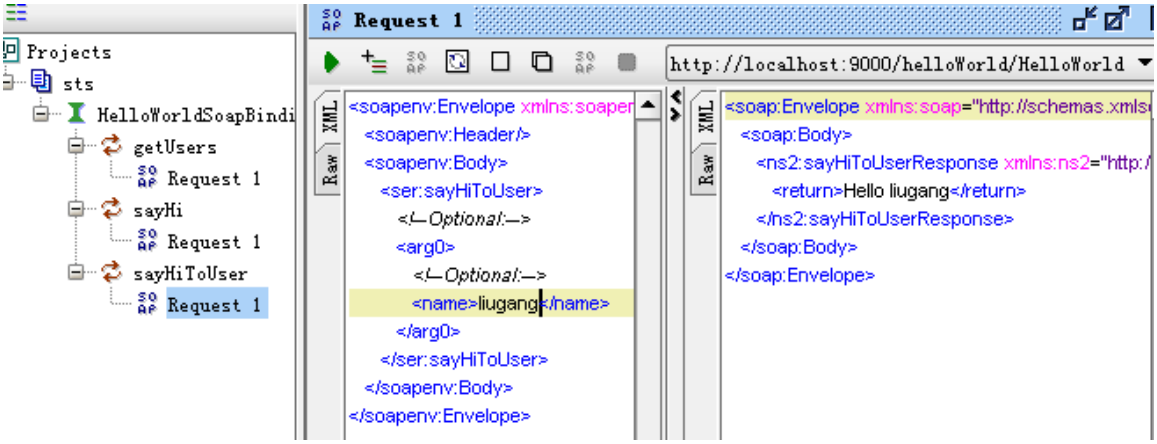
2. You may wish to save the WSDL as a file, such as `helloWorld.wsdl`, for subsequent usage by soapUI. However, soapUI can also easily read WSDLs from the http://localhost:9000/helloWorld/HelloWorld?wsdl.

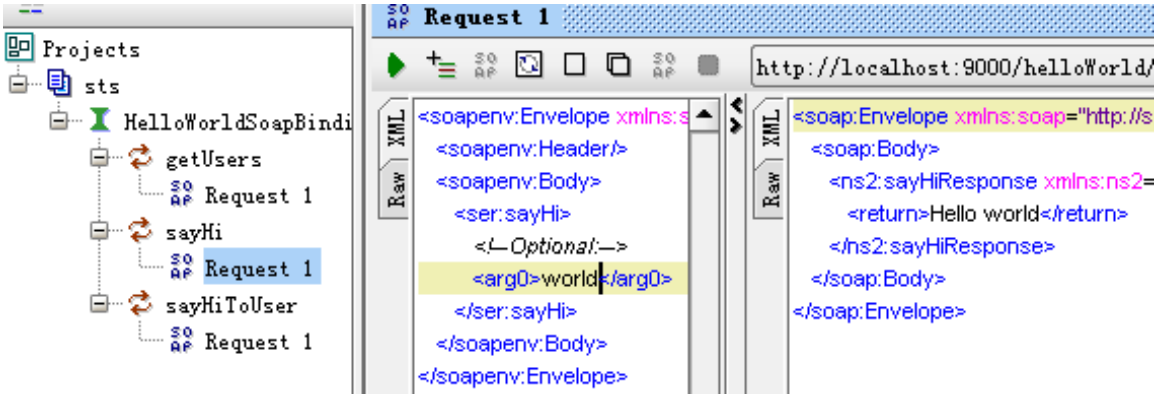
3. We can test the service by invoking the interfaces, using SOAPUI. For more details, please refer to http://www.soapui.org.

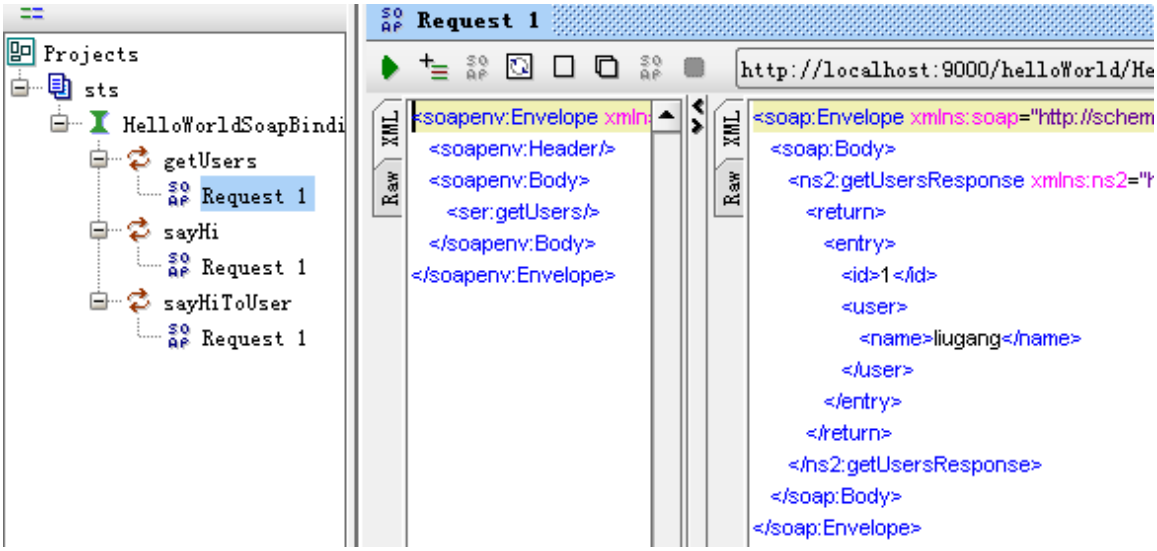
   Open SOAPUI, create a project and then import the saved wsdl file into SOAPUI. Then, expand the project node, and there will be a node name "HelloWorldSoapBinding". Now, expand the "HelloWorldSoapBinding", there will be three nodes there, which correspond to methods defined in "HelloWorld.java".



Then test the interfaces one by one. Here are some screenshots to illustrate this:







That completes the description of code-contract. Next we will look at contract-first development.

3.7. Contract-first development

1. **Project Skeleton**

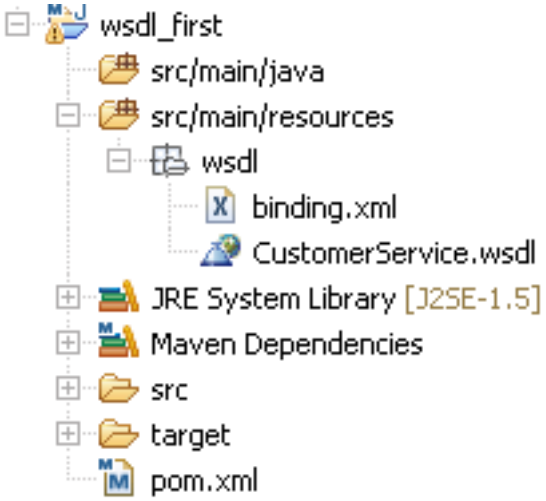
   We need to generate the project skeleton first; the steps are exactly the same as Section 3.3, “Project Skeleton” except for the project name. We'll name this project "wsdl_first" here, because we want to use the `${Talend-ESB-Version}/examples/apache/cxf/wsdl_first` as the sample, removing all previously generated source code.

   For simplicity, this sample uses the same http://customerservice.example.com/ namespace for both the contract and the XML Schema elements used in the contract's messages. However, for greater reuse of the same schema across multiple web service contracts you'll find it helpful to use separate namespaces for the schema and the contract(s) using it.

2. **Prepare WSDL**

   Create a source root `src/main/resources`, and create a folder `wsdl` under it. Next, copy the wsdl file `CustomerService.wsdl` and `binding.xml` from `${Talend-ESB-Version}/examples/apache/cxf/wsdl_first/wsdl` to it. Now the project structure should appear as below:

   (We'll explore the functionality of `binding.xml` later in this section.)

   We have prepared the WSDL file. Now, we are ready to generate the code from this WSDL. But before continuing, we need to setup the project configuration.

3. **Modify configuration**

   Update the file to use a recent version of CXF (we use `2.6.0` here):

```
<properties>
    <cxf-version>2.6.0</cxf-version>
 </properties>
 <dependencies>
    <dependency>
       <groupId>org.apache.cxf</groupId>
       <artifactId>cxf-rt-frontend-jaxws</artifactId>
       <version>${cxf-version}</version>
    </dependency>
    <dependency>
       <groupId>org.apache.cxf</groupId>
       <artifactId>cxf-rt-transports-http</artifactId>
       <version>${cxf-version}</version>
    </dependency>
 </dependencies>
```

We want to generate the Java artifacts from WSDL - in order to generate code, we need to add the following declaration in our pom.xml:

```
<build>
    <finalName>wsdlfirst</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-codegen-plugin</artifactId>
        <version>${cxf-version}</version>
        <executions>
          <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
              <sourceRoot>${basedir}/src/main/java</sourceRoot>
              <wsdlOptions>
                <wsdlOption>
                  <wsdl>${basedir}/src/main/resources/wsdl/CustomerService.wsdl</wsdl>
                  <bindingFiles>
                    <bindingFile>${basedir}/src/main/resources/wsdl/binding.xml</bindingFile>
                  </bindingFiles>
                  <extraargs>
                    <extraarg>-impl</extraarg>
                  </extraargs>
                </wsdlOption>
              </wsdlOptions>
            </configuration>
            <goals>
              <goal>wsdl2java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

Here, within the "generate-sources" phase, the goal is "wsdl2java" in plugin "cxf-codegen-plugin". Also defined is the WSDL that will be used, where the code will be put into, and the "bindingFile" configuration element is set to `binding.xml`. Please check the CXF website for more information on the Maven cxf-codegen-plugin and the wsdl-to-java process in particular.

4. **Binding files**

The binding files are a way to customize the output of the artifacts that CXF generates. For example, the default mapping of xsd:dateTime/xsd:time/xsd:date is XMLGregorianCalendar, which is often not desired. So we define the binding.xml to customize the date mapping:

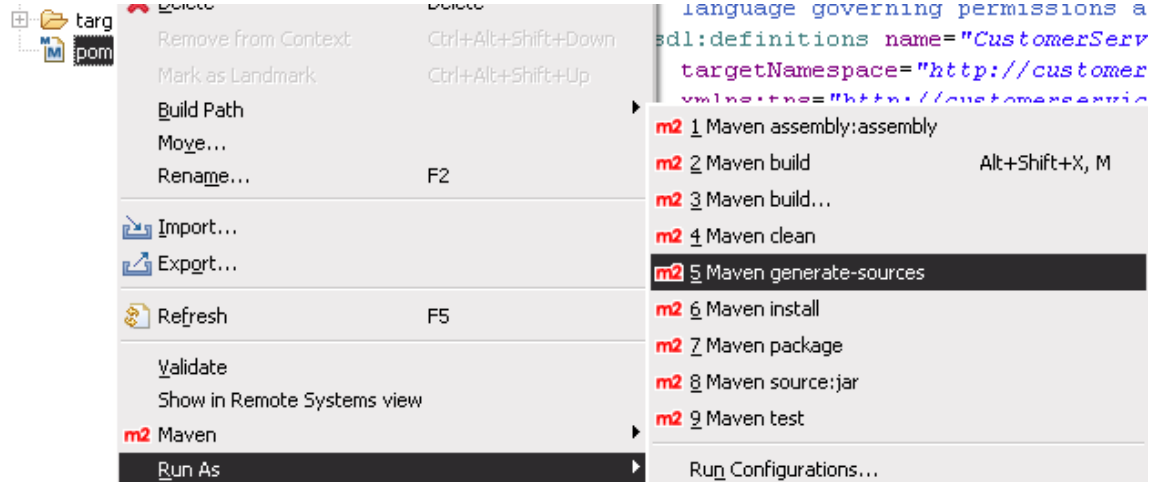
```
<jaxws:bindings wsdlLocation="CustomerService.wsdl"
        xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
 <!-- <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping> -->
 <jaxws:bindings  node="wsdl:definitions/wsdl:types/xs:schema">
    <jxb:globalBindings>
      <jxb:javaType name="java.util.Date" xmlType="xs:dateTime"
                parseMethod="org.apache.cxf.tools.common.DataTypeAdapter.parseDateTime"
                printMethod="org.apache.cxf.tools.common.DataTypeAdapter.printDateTime"/>
      <jxb:javaType name="java.util.Date" xmlType="xs:date"
                parseMethod="org.apache.cxf.tools.common.DataTypeAdapter.parseDate"
                printMethod="org.apache.cxf.tools.common.DataTypeAdapter.printDate"/>
    </jxb:globalBindings>
 </jaxws:bindings>
</jaxws:bindings>
```

For more details about how to customize the generated codes, please refer to the JAX-WS specification http://download.oracle.com/otndocs/jcp/jaxws-2_0-pfd-spec-oth-JSpec/.

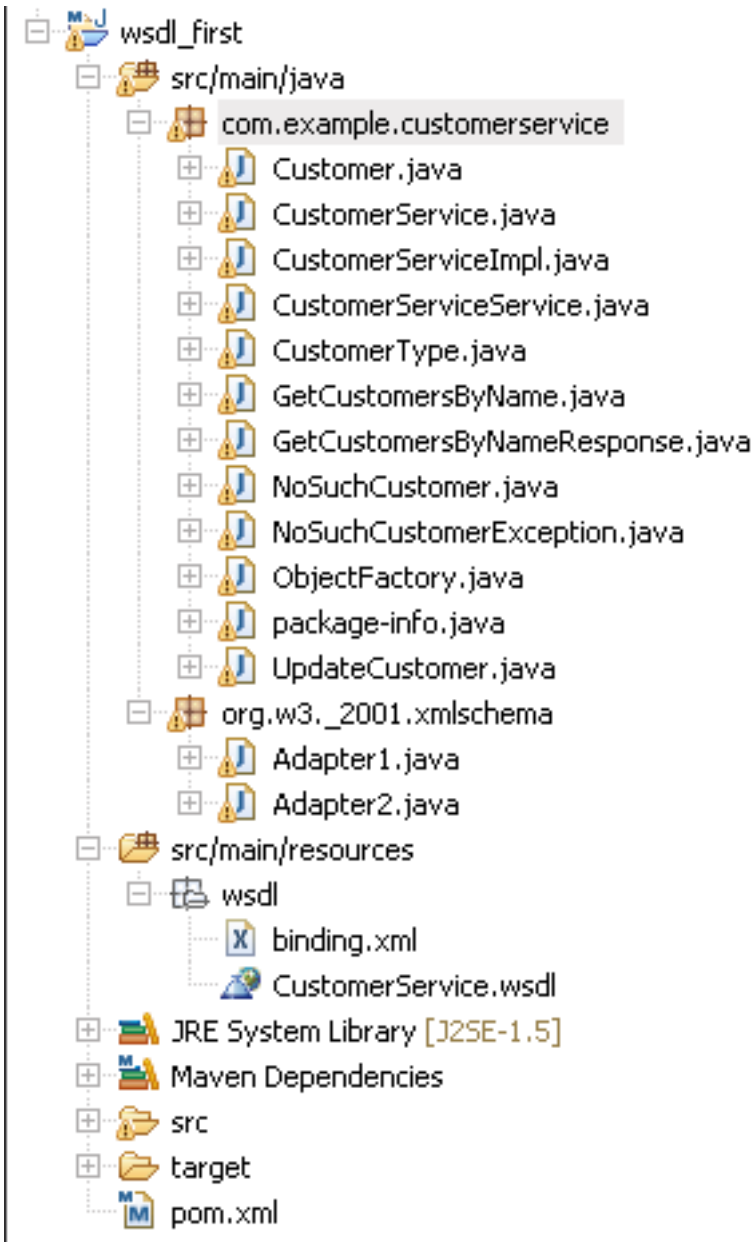
Now when you run **`mvn generate-sources`**, the codes will be generated under `${basedir}/src/main/java`, and the source WSDL file is `${basedir}/src/main/resources/wsdl/CustomerService.wsdl`. For details on how to configure the generator, please visit http://cxf.apache.org/docs/maven-cxf-codegen-plugin-wsdl-to-java.html .

5. **Generate code**

We are now prepared to generate code. you can run **`mvn generated-sources`** from popup menu on `pom.xml`:



If successful, the project structure should be similar to that below:



6. **Implement the service**

If we investigate into the `CustomerService.wsdl`, we can find there is only one portType "CustomerService", with two operations "updateCustomer" and "getCustomersByName". We know that `CustomerService.java` is the service interface, and `CustomerServiceImpl.java` is the implementor according to our configuration in pom.xml. We need to complete our service by filling in the methods of `CustomerServiceImpl.java`. We can do this by copying them from the `wsdl_first` sample of `${Talend-ESB-Version}`, which located in: `${Talend-ESB-Version}/examples/apache/cxf/wsdl_first/src/main/java/com/example/customerservice/server`:

```
/**
 * The WebServiceContext can be used to retrieve special attributes like the
 * user principal. Normally it is not needed
 */
@Resource
WebServiceContext wsContext;

public List<Customer> getCustomersByName(String name)
                    throws NoSuchCustomerException {
   if ("None".equals(name)) {
      NoSuchCustomer noSuchCustomer = new NoSuchCustomer();
      noSuchCustomer.setCustomerName(name);
      throw new NoSuchCustomerException(
                        "Did not find any matching customer for name="
                        + name,
                        noSuchCustomer);
   }

   List<Customer> customers = new ArrayList<Customer>();
   for (int c = 0; c < 2; c++) {
      Customer cust = new Customer();
      cust.setName(name);
      cust.getAddress().add("Pine Street 200");
      Date bDate = new GregorianCalendar(2009, 01, 01).getTime();
      cust.setBirthDate(bDate);
      cust.setNumOrders(1);
      cust.setRevenue(10000);
      cust.setTest(new BigDecimal(1.5));
      cust.setType(CustomerType.BUSINESS);
      customers.add(cust);
   }

   return customers;
}

public void updateCustomer(Customer customer) {
   System.out.println("update request was received");
   try {
      Thread.sleep(10000);
   } catch (InterruptedException e) {
      // Nothing to do here
   }
   System.out.println("Customer was updated");
}
```

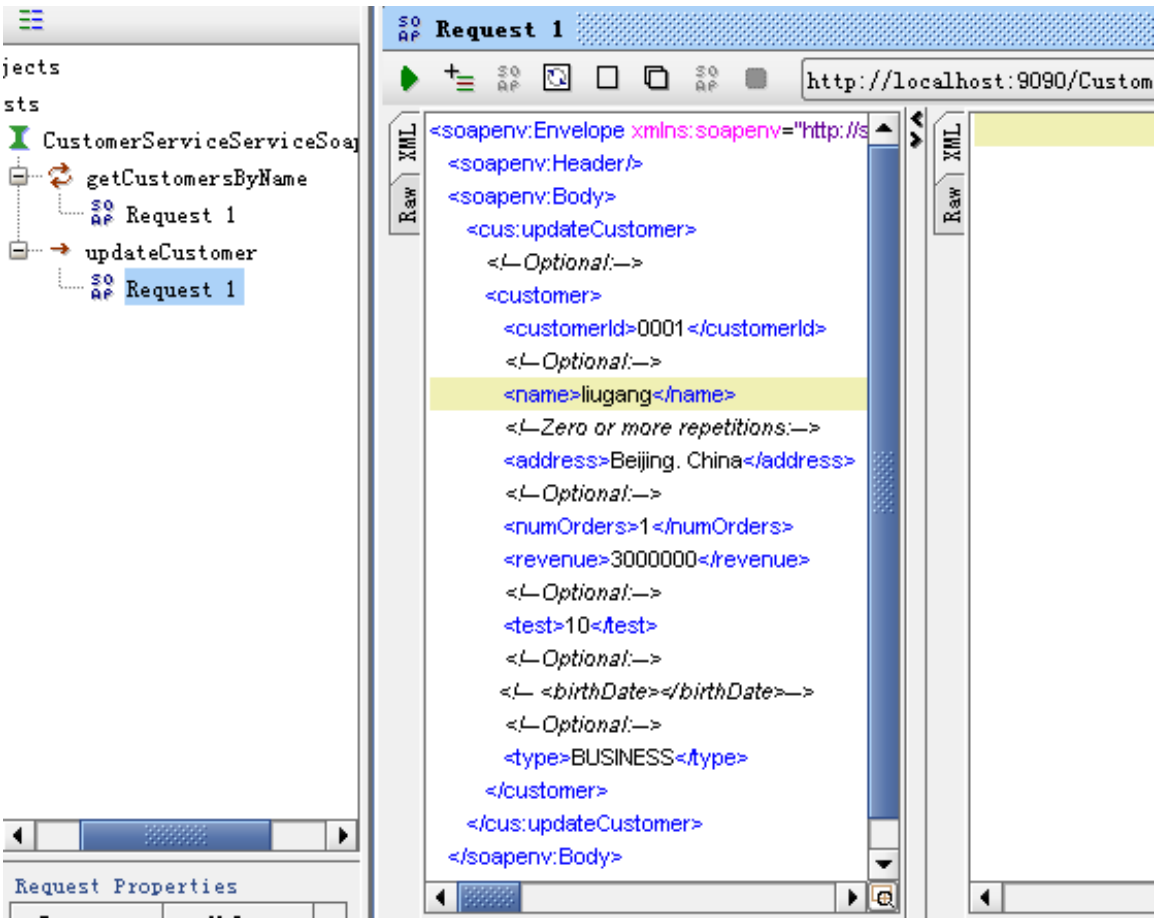
Now that we have finished our service, we can test it.

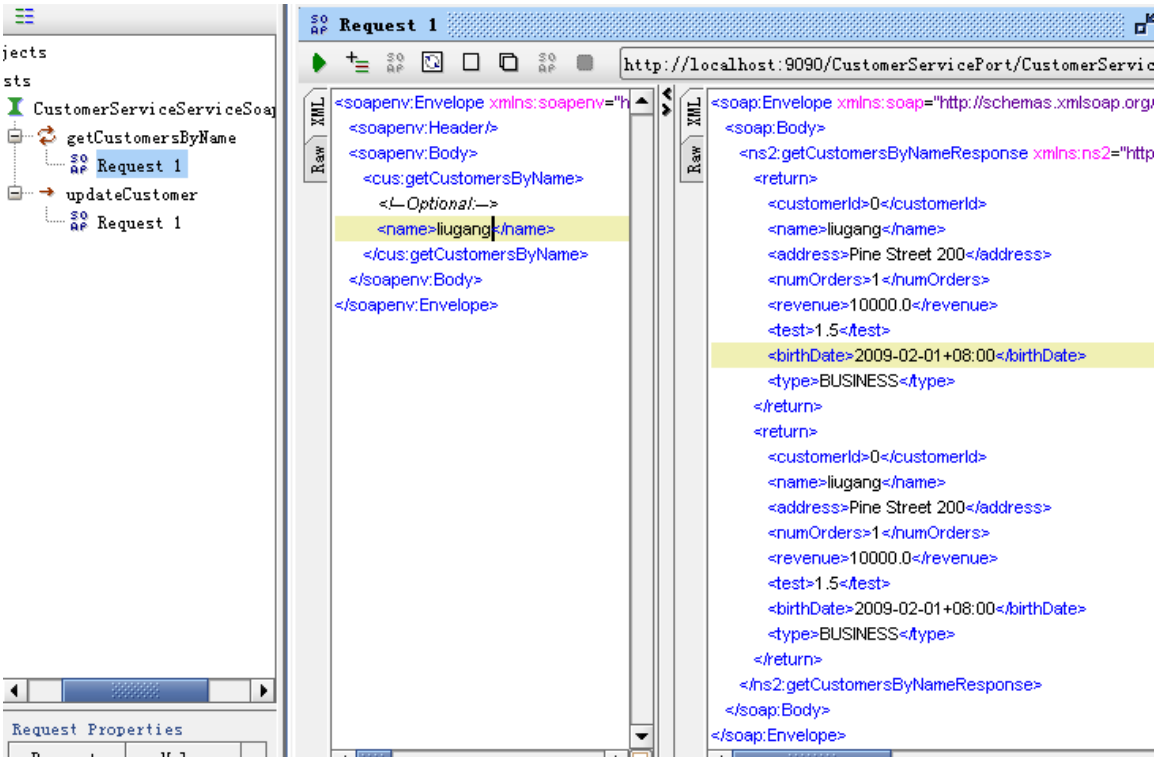
7. **Test the service**

The steps are exactly same as testing the service in Section 3.2, “Code-first development” and Section 3.6, “Test the service”, please refer to these for details.

You can check the WSDL by accessing http://localhost:9090/CustomerServicePort/CustomerServiceService?wsdl

Save the WSDL, and then start SOAPUI. Import the saved WSDL file. You can now test by sending the SOAP messages. The below are some screenshots to illustrate:


jects
sts
CustomerServiceServiceSoa
getCustomersByName
Request 1
updateCustomer
Request 1
Request Properties
Request 1
http://localhost:9090/Custom
XML
Raw
<soapenv:Envelope xmlns:soapenv="http://s
<soapenv:Header/>
<soapenv:Body>
<cus:updateCustomer>
<!--Optional:-->
<customer>
<customerId>0001</customerId>
<!--Optional:-->
<name>liugang</name>
<!--Zero or more repetitions:-->
<address>Beijing. China</address>
<!--Optional:-->
<numOrders>1</numOrders>
<revenue>3000000</revenue>
<!--Optional:-->
<test>10</test>
<!--Optional:-->
<!-- <birthDate></birthDate>-->
<!--Optional:-->
<type>BUSINESS</type>
</customer>
</cus:updateCustomer>
</soapenv:Body>


jects
sts
CustomerServiceServiceSoa
getCustomersByName
Request 1
updateCustomer
Request 1
Request Properties
Request 1
http://localhost:9090/CustomerServicePort/CustomerServic
XML
Raw
<soapenv:Envelope xmlns:soapenv="h
<soapenv:Header/>
<soapenv:Body>
<cus:getCustomersByName>
<!--Optional:-->
<name>liugang</name>
</cus:getCustomersByName>
</soapenv:Body>
</soapenv:Envelope>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/
<soap:Body>
<ns2:getCustomersByNameResponse xmlns:ns2="http
<return>
<customerId>0</customerId>
<name>liugang</name>
<address>Pine Street 200</address>
<numOrders>1</numOrders>
<revenue>10000.0</revenue>
<test>1.5</test>
<birthDate>2009-02-01+08:00</birthDate>
<type>BUSINESS</type>
</return>
<return>
<customerId>0</customerId>
<name>liugang</name>
<address>Pine Street 200</address>
<numOrders>1</numOrders>
<revenue>10000.0</revenue>
<test>1.5</test>
<birthDate>2009-02-01+08:00</birthDate>
<type>BUSINESS</type>
</return>
</ns2:getCustomersByNameResponse>
</soap:Body>
</soap:Envelope>

3.8. REST Services

The JAX-RS Section on the Apache CXF website provides a solid background to implementing REST services and also provides the latest information on the newest RESTful features offered by CXF.

For an example of working with a RESTful application in Eclipse and deploying the service to either Tomcat or the Talend OSGi container, let's look at the JAXRS-Intro sample provided in the `examples/talend` folder of the Talend ESB installation. The demo lists the Persons who are part of a generic membership, and allows GETs to retrieve a single member or all members, POSTs to add members, and PUTs for updates to membership information.

The JAX-RS Server provides one service via the registration of a root resource class, MembershipService which relies on within-memory data storage. MembershipService provides a list of its members, which are individual Person objects containing name and age. New persons can be added to the MembershipService, and individual members can have their information updated. The RESTful client uses CXF JAX-RS WebClient to traverse all the information about an individual Person and also add a new child.

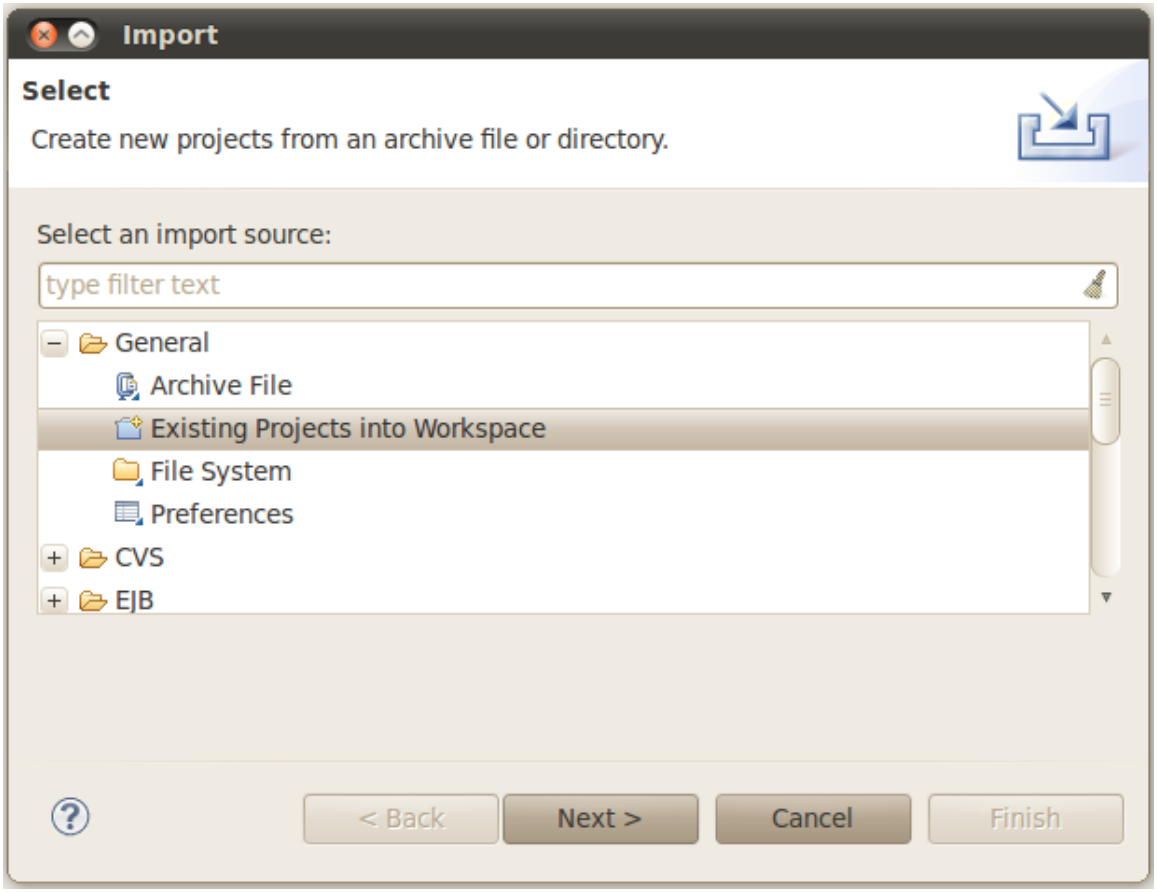
This sample consists of four subfolders:

Folder	Description
client	This is a sample client application that uses the CXF JAX-RS API to create HTTP-centric and proxy clients and makes several calls with them.
common	This directory contains the code that is common for both the client and the server. POJOs and the REST interface is kept here.
service	This is the JAX-RS service holding the Membership root resources packaged as an OSGi bundle.
war	This module creates a WAR archive containing the code from common and service modules.

Procedure 3.1. Working with a REST sample in Eclipse

1. From a command-line windows, navigate to the jax-rs folder and type **`mvn clean install eclipse:eclipse`**. This will create an Eclipse project out of this sample that we can import into Eclipse.

2. From Eclipse we can now import the project. From the Menu row, select **`File : Import...`**, and from the resulting Import popup, choose Existing Projects into Workspace (see illustration below). Select Next.



3. Select the four Eclipse projects comprising this example: `jaxrs-intro-client`, `jaxrs-intro-common`, `jaxrs-intro-service-bundle`, and `jaxrs-intro-service-war`. You'll see them listed in the left-side Eclipse Navigator and Project Explorer views. At this stage any of the files can be viewed and modified. Be sure to run `mvn clean install` from the `jaxrs-intro` folder within a command prompt window after any changes made.

4. Prior to running the client, we'll need to activate the REST service, which we can do in at least two ways:

   - To run the example within Talend ESB, we'll need to create the Karaf features file that contains the definition for this service. First, from a command prompt navigate to the `features` folder (sibling to `jaxrs-intro`) and run **`mvn clean install`**. Next, from the command prompt enter **`features:addurl mvn:com.talend.sf.examples/osgi/1.0/xml/features`** to install the features file followed by **`features:install tsf-example-jaxrs-intro`** to install the JAXRS-Intro service.

   - To run the example within CXF's internal (Jetty-based) servlet container, navigate to the `war` folder and run **`mvn jetty:run`**.

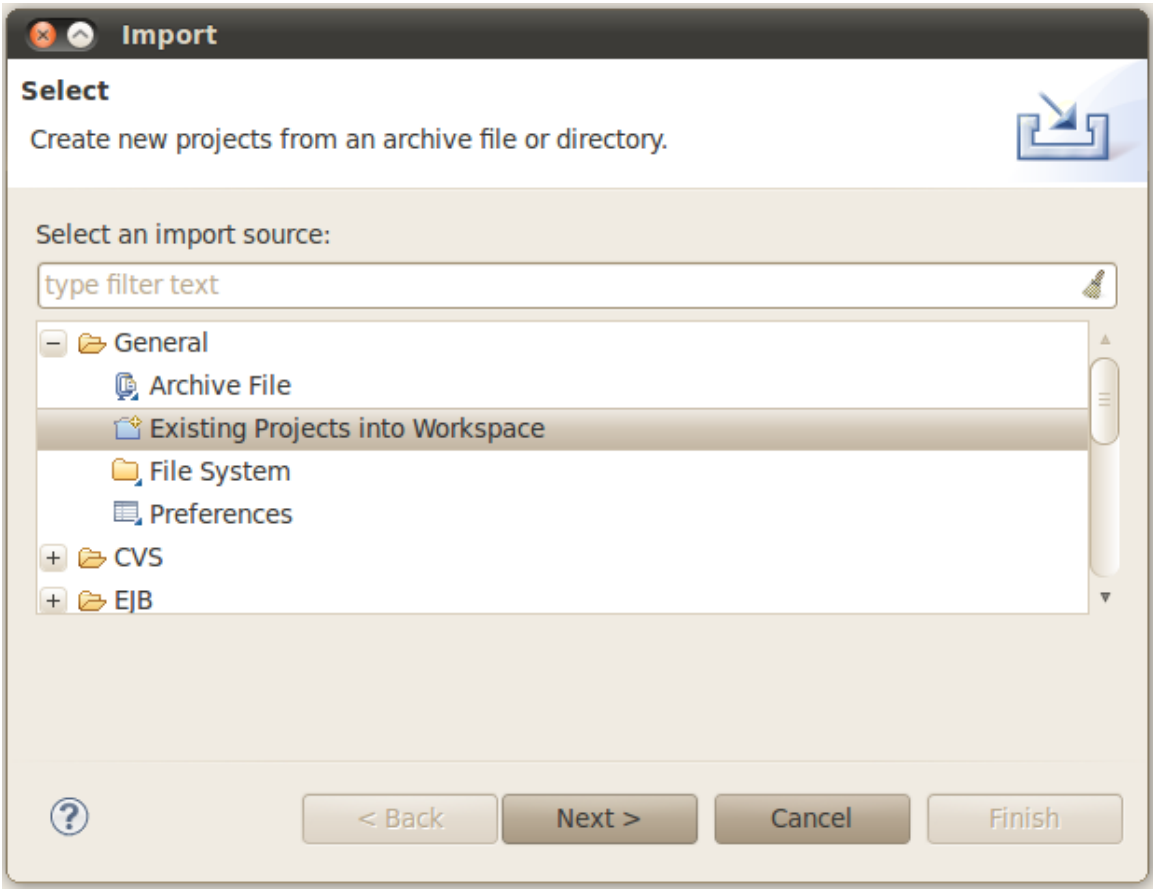
5. To run the client, from a command prompt in the jaxrs-intro/client folder, run `mvn exec:java`.

Chapter 4. Camel Routes Overview

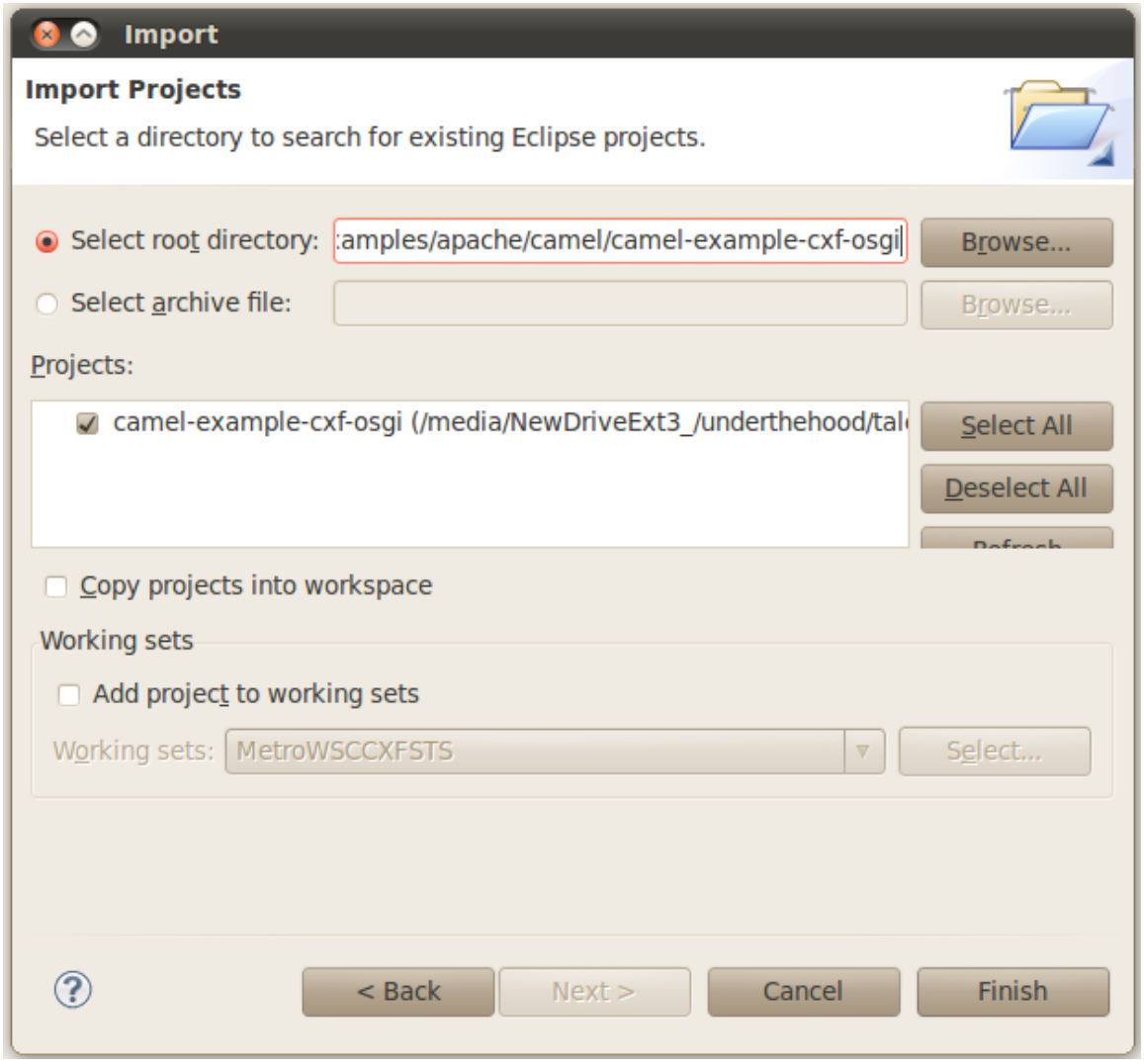
To demonstrate the usage and deployment of a Camel route within Talend ESB using Eclipse, we'll use Camel's `camel-example-cxf-osgi` example, located within the `examples/apache/camel` folder of the Talend ESB distribution. Steps:

1. From a command-line windows, navigate to this folder and type **`mvn clean install eclipse:eclipse`**. This will create an Eclipse project out of this sample that we can import into Eclipse.

2. From Eclipse we can now import the project. From the Menu row, select **`File : Import...`**, and from the resulting Import popup, choose Existing Projects into Workspace (see illustration below). Select Next.



3. For the root directory navigate to the `examples/apache/camel/camel-example-cxf-osgi` folder and select the `camel-example-cxf-osgi` example from the Projects list. Select "Finish" and you'll see it in the Eclipse Package Explorer. Here would be a good time to open up the project source files and look at the code (this example is explained on the Apache Camel site at http://camel.apache.org/cxf-example-osgi.html.)

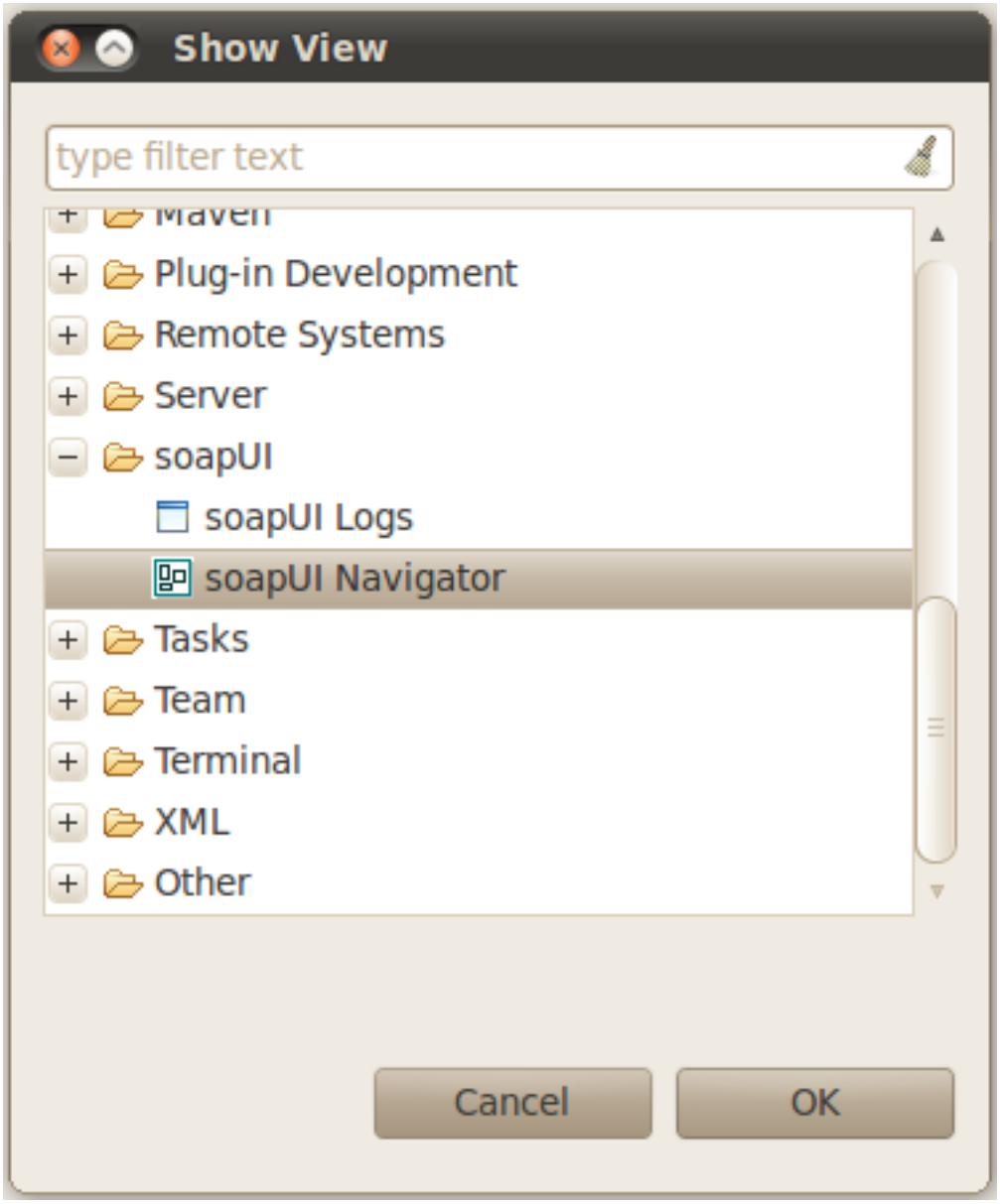


4. As this example runs in the Talend OSGi container, we'll need to start the container prior to running this example. Navigate to the [Talend-ESB-Version/container/bin] folder and enter **`trun.bat`** or **`./trun`**. Once the TESB shell activates the following commands to install the example will be needed:

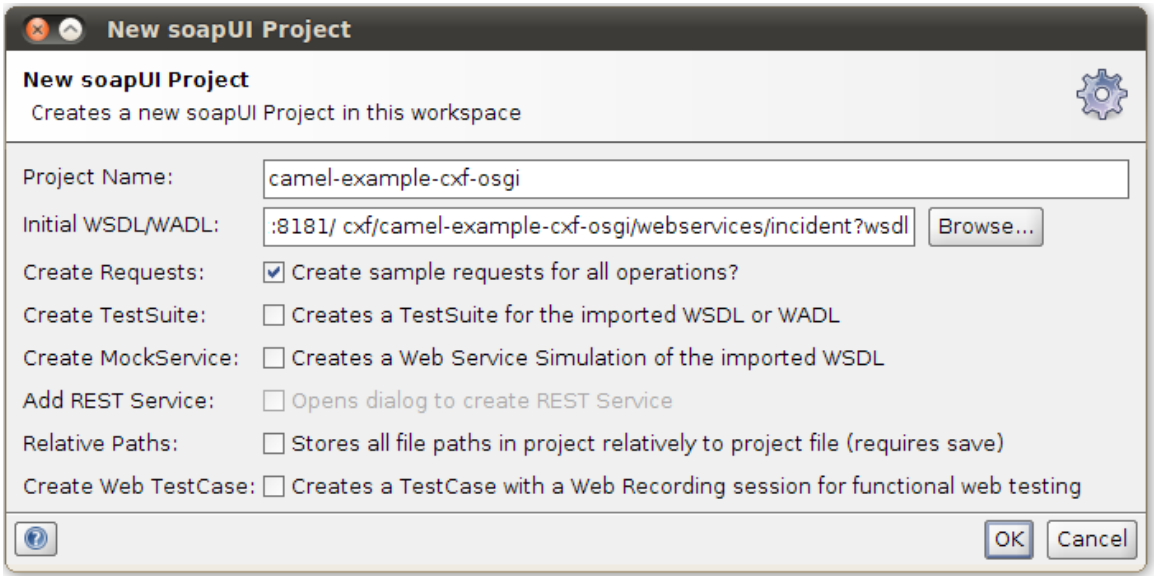
```
features:addUrl mvn:org.apache.camel.karaf/apache-camel/2.9.2/xml/featur
es
features:install war
features:install camel-spring
features:install camel-jaxb
features:install camel-cxf
osgi:install -s mvn:org.apache.camel/camel-example-cxf-osgi/2.9.2
```

5. Open a web browser and make sure you can view the above web service WSDL at http://localhost:8040/services/camel-example-cxf-osgi/webservices/incident?wsdl before continuing.

6. We'll make a SOAP call using soapUI in this step. Make sure you've already installed soapUI in Eclipse as discussed in Chapter 2, *Eclipse Setup Overview* (standalone soapUI is also fine.) From Eclipse, select Menu Item Window | Show View | Other..., and select soapUI Navigator from the View list (see illustration below.)



7. Create a new project called camel-example-cxf-osgi. Point to the following url: http://localhost:8181/cxf/camel-example-cxf-osgi/webservices/incident?wsdl



8. In the soapUI Navigator view, open the request 1 (under camel-example-cxf-osgi --> ReportIncidentBinding --> ReportIncident) and copy and paste the following SOAP Message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header />
   <soap:Body>
      <ns2:inputReportIncident xmlns:ns2="http://reportincident.example.camel.apache.org">
         <incidentId>111</incidentId>
         <incidentDate>2011-10-05</incidentDate>
         <givenName>Bob</givenName>
         <familyName>Smith</familyName>
         <summary>incident summary</summary>
         <details>incident summary details</details>
         <email>bobsmith@email.com</email>
         <phone>123-456-7890</phone>
      </ns2:inputReportIncident>
   </soap:Body>
</soap:Envelope>
```

9. Press the green arrow in the soapUI navigator to make the SOAP call. Within the Navigator View you'll see the SOAP response stating that the incident report was accepted. Also, checking a new `target/inbox` folder under the `camel-example-cxf-osgi` sample directory you'll see a file was created storing the SOAP request, completing the Camel route.

Chapter 5. Talend ESB Services Overview

5.1. Service Locator

The Service Locator provides service consumers with a mechanism to register, and also discover service endpoints at runtime, thus isolating consumers from the knowledge about the physical location of the endpoint. Talend ESB uses Apache ZooKeeper as its service locator server, located within the Talend ESB distribution at "${Talend-ESB-version}/zookeeper". Please see http://zookeeper.apache.org/" for more information about ZooKeeper. Also note the examples folder of the Talend ESB distribution provides a "locator" example you can learn from.

We'll look at a simple greeting example to show you how to use the Service Locator.

5.1.1. Service interface

Within Eclipse:

1. Create a Maven project, following the same steps as Section 3.3, “Project Skeleton”, and call it "locator_common" for the purposes of this example.

2. Remove all default sources, as well as test source folder.

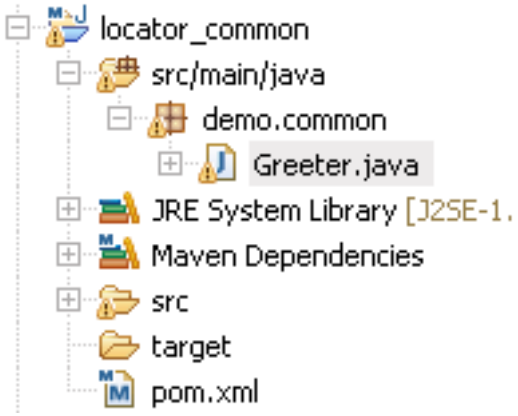
3. Create a package named "demo.common", and create a interface `Greeter.java`.

```
package demo.common;

import javax.jws.WebService;

@WebService(targetNamespace = "http://talend.org/esb/examples/",
        name = "Greeter")
public interface Greeter {
    String greetMe(String requestType);
}
```

Greeter.java will be the service interface. Now, the project structure will look like the following:



4. The common application will be deployed as an OSGi bundle. So we'll need to edit the `pom.xml`. (Please refer to Section 3.6, “Test the service” for details.)

```
<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.talend.liugang.cxf</groupId>
   <artifactId>locator_common</artifactId>
   <version>1.0.0</version>
   <packaging>bundle</packaging>
   <name>locator_common</name>
   <url>http://maven.apache.org</url>

   <dependencies>
      <dependency>
         <groupId>org.apache.felix</groupId>
         <artifactId>org.osgi.core</artifactId>
         <version>1.4.0</version>
      </dependency>
   </dependencies>
   <build>
      <plugins>
         <plugin>
            <groupId>org.apache.felix</groupId>
            <artifactId>maven-bundle-plugin</artifactId>
            <extensions>true</extensions>
            <version>2.3.7</version>
            <configuration>
               <instructions>
                  <Bundle-SymbolicName>
                     ${project.artifactId}
                  </Bundle-SymbolicName>
                  <Export-Package>
                     demo.common
                  </Export-Package>
               </instructions>
            </configuration>
         </plugin>
      </plugins>
   </build>
</project>
```

Now we've finished the definition of the service. Select "Run As->Maven Install" from the M2Eclipse Popup menu on the pom.xml to install the application into your Maven repository.

Next we'll look at the implementation of this service.

5.1.2. Service implementation

This time, we'll create a "locator_service" project first, following the steps as above. Then create a `GreeterImpl.java` which implements the Greeter interface defined above. The contents of GreeterImpl.java are:

```
import javax.jws.WebService;

import demo.common.Greeter;
@WebService(targetNamespace = "http://talend.org/esb/examples/",
      serviceName = "GreeterService")
public class GreeterImpl implements Greeter {

   public String greetMe(String me) {
      System.out.println("Executing operation greetMe");
      System.out.println("Message received: " + me + "\n");
      return "Hello " + me;
   }
}
```

For each input, a statement "'Hello '+input" will be returned. Now it's time to bring the ServiceLocator in. As we said at the beginning of this section, the Service Locator is a mechanism to discover service endpoints at runtime. In order to make the Implementation discoverable, we need to register it first.

There are two ways to register a service: by Spring configuration or by code directly.

For Spring configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
   <import resource="classpath:META-INF/cxf/cxf.xml" />
   <import resource="classpath:META-INF/tesb/locator/beans-osgi.xml" />

   <jaxws:endpoint xmlns:tns="http://talend.org/esb/examples/"
      id="greeter" implementor="demo.service.GreeterImpl" serviceName="tns:GreeterService"
      address="/GreeterService">
      <jaxws:features>
         <bean class="org.talend.esb.servicelocator.cxf.LocatorFeature">
         </bean>
      </jaxws:features>
   </jaxws:endpoint>
</beans>
```

Note for the above Spring file the OSGi import line (`classpath:META-INF/tesb/locator/beans-osgi.xml`) is the only difference from a standard Spring configuration file.

Then load it by using "ClassPathXmlApplicationContext". It's important to include the configuration file in exported bundle and also add the necessary dependencies for Spring configuration.

Note the <jaxws:features></jaxws:features> lines, which add a feature "org.talend.esb.servicelocator.cxf.LocatorFeature", for using the Service Locator.

The alternative code version is:

```
    LocatorFeature locatorFeature = new LocatorFeature();
      Greeter greeterService = new GreeterImpl();
      svrFactory = new JaxWsServerFactoryBean();
      svrFactory.setServiceClass(Greeter.class); // WSDL operations that service will implement
      svrFactory.setAddress("http://localhost:8082/services/Greeter"); // endpoint service will listen on
      svrFactory.setServiceBean(greeterService); // implementation of WSDL operations
      svrFactory.getFeatures().add(locatorFeature); // attach LocatorFeature to web service provider
      svrFactory.create();
```

Similar to Section 5.1, “Service Locator”, we'll export "locator_service" as a bundle, so the BundleActivator is the best place to register or remove this service:

```
import org.apache.cxf.jaxws.JaxWsServerFactoryBean;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.talend.esb.servicelocator.cxf.LocatorFeature;

import demo.common.Greeter;

public class Activator implements BundleActivator {

    private JaxWsServerFactoryBean svrFactory;

    public void start(BundleContext context) throws Exception {
        LocatorFeature locatorFeature = new LocatorFeature();
        Greeter greeterService = new GreeterImpl();
        svrFactory = new JaxWsServerFactoryBean();
        svrFactory.setServiceClass(Greeter.class);
        svrFactory.setAddress("http://localhost:8082/services/Greeter");
        svrFactory.setServiceBean(greeterService);
        svrFactory.getFeatures().add(locatorFeature);
        svrFactory.create();
    }

    public void stop(BundleContext context) throws Exception {
        svrFactory.destroy();
    }
}
```

There's all the code we need to provide. Next, we need to configure the pom.xml, add necessary dependencies, and configure the exported bundle information.

Finally, the contents of pom.xml are:

```
<groupId>com.talend.liugang.cxf</groupId>
  <artifactId>locator_service</artifactId>
  <version>1.0.0</version>
  <packaging>bundle</packaging>

  <name>locator_service</name>
  <url>http://maven.apache.org</url>
  <properties>
     <cxf.version>2.6.0</cxf.version>
  </properties>

  <dependencies>
     <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-api</artifactId>
        <version>${cxf.version}</version>
     </dependency>

     <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>${cxf.version}</version>
        <scope>compile</scope>
     </dependency>

     <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http-jetty</artifactId>
        <version>${cxf.version}</version>
     </dependency>

     <dependency>
        <groupId>org.talend.esb</groupId>
        <artifactId>locator</artifactId>
        <version>5.1.1</version>
     </dependency>

     <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>locator_common</artifactId>
        <version>${project.version}</version>
     </dependency>

     <dependency>
        <groupId>org.apache.felix</groupId>
        <artifactId>org.osgi.core</artifactId>
        <version>1.4.0</version>
     </dependency>
  </dependencies>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <extensions>true</extensions>
      <version>2.3.7</version>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
          <Import-Package>demo.common,javax.jws,
            org.apache.cxf.endpoint,org.apache.cxf.jaxws,
            org.osgi.framework,org.talend.esb.locator
          </Import-Package>
          <Bundle-Activator>demo.service.Activator</Bundle-Activator>
          <Require-Bundle>org.apache.cxf.bundle;version="2.6.0"</Require-Bundle>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
```

And also, we installed it into Maven by running "Maven Install".

So far, we have defined the Service interface, and given it an implementation. It's time to write a client which will consume the service.

5.1.3. Service Consumer

This time we will try to consume the service above by using Service Locator instead of referencing the implementor directly.

Same as with service registration, you can use Spring configuration or code directly.

For Spring configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:jaxws="http://cxf.apache.org/jaxws"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd
  http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util-3.0.xsd ">

  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:/META-INF/tesb/locator/beans-osgi.xml" />

  <jaxws:client id="greeterService" address="locator://more_useful_information"
    serviceClass="demo.common.Greeter">
    <jaxws:features>
      <bean class="org.talend.esb.servicelocator.cxf.LocatorFeature">
      </bean>
    </jaxws:features>
  </jaxws:client>
</beans>
```

The alternative code version is:

```
    JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
      LocatorFeature locatorFeature = new LocatorFeature();
      factory.getFeatures().add(locatorFeature);
      factory.setServiceClass(Greeter.class);
      factory.setAddress("locator://more_useful_information");
      Greeter client = (Greeter) factory.create();
      String response = client.greetMe("MyName");
```

An important point to note: We must use the locator protocol for client 'address="locator://more_useful_information"'. And also, we will export the project as a OSGi bundle, so we'll need to setup the test fragment in start() method of BundleActivator:

```
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.talend.esb.servicelocator.cxf.LocatorFeature;

import demo.common.Greeter;

public class Client implements BundleActivator {

   public void start(BundleContext context) throws Exception {
      JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
      LocatorFeature locatorFeature = new LocatorFeature();
      factory.getFeatures().add(locatorFeature);
      factory.setServiceClass(Greeter.class);
      factory.setAddress("locator://more_useful_information");
      Greeter client = (Greeter) factory.create();
      String response = client.greetMe("MyName");
      System.out.println(response);
   }

   public void stop(BundleContext context) throws Exception {
   }
}
```

The contents of pom.xml are:

```
<groupId>com.talend.liugang.cxf</groupId>
  <artifactId>locator_client</artifactId>
  <version>1.0.0</version>
  <packaging>bundle</packaging>

  <name>locator_client</name>
  <url>http://maven.apache.org</url>

  <properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <cxf.version>2.6.0</cxf.version>
  </properties>

  <dependencies>
     <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-api</artifactId>
        <version>${cxf.version}</version>
     </dependency>
     <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>${cxf.version}</version>
        <scope>compile</scope>
     </dependency>
     <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http-jetty</artifactId>
        <version>${cxf.version}</version>
     </dependency>
     <dependency>
        <groupId>org.talend.esb</groupId>
        <artifactId>locator</artifactId>
        <version>5.1.1</version>
     </dependency>
     <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>locator_common</artifactId>
        <version>${project.version}</version>
     </dependency>
  </dependencies>

  <build>
     <plugins>
        <plugin>
           <groupId>org.apache.felix</groupId>
           <artifactId>maven-bundle-plugin</artifactId>
           <extensions>true</extensions>
           <version>2.3.7</version>
           <configuration>
              <instructions>
                 <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                 <Bundle-Activator>demo.client.Client</Bundle-Activator>
                 <Require-Bundle>locator_common</Require-Bundle>
              </instructions>
           </configuration>
        </plugin>
     </plugins>
  </build>
```

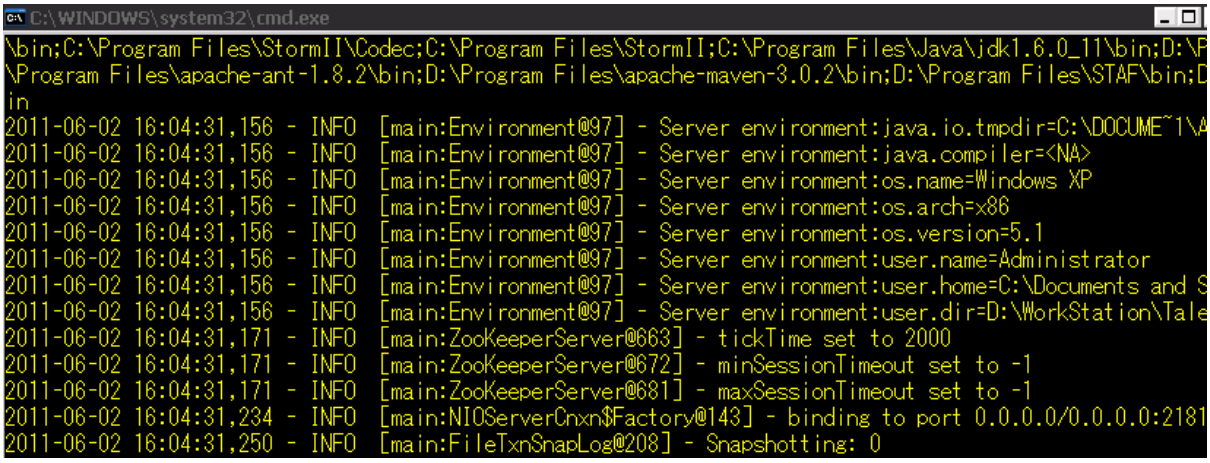
Then execute "Run As -> Maven Install" to install this bundle. Thus far we have finished all bundles, now we will see how to install them and also combine them with ServiceLocator.

5.1.4. Setup ZooKeeper

Go into "${Talend-ESB-version}/zookeeper". To start the Service Locator we need to provide a configuration file. The default configuration file is "${Talend-ESB-Version}/zookeeper/conf/zoo.cfg", There's a "zoo_sample.cfg" there, you can just rename the "zoo_sample.cfg" to "zoo.cfg". For a standalone configuration, you can just give the configuration like below:

```
tickTime=2000
dataDir=./var/locator
clientPort=2181
```

The tickTime refers to the basic unit of time measurement used by ZooKeeper, used for later configuration of timeouts and other parameters. The dataDir holds database snapshots and transaction logs. (Check the ZooKeeper Administration Manual for information on all possible parameters.) Notice that, the "clientPort" number should be same as the "endpointPrefix" defined in LocatorFeature above. Once we have the "zoo.cfg", we can start or stop the zooKeeper by running "${Talend-ESB-Version}/zookeeper/bin/zkServer.cmd start/stop" on Windows or "${Talend-ESB-Version}/zookeeper/bin/zkServer.sh start/stop" on Linux:



Then startup the OSGi Container, for details on this please refer to Section 3.5, “Deploy the bundle”. After the container has started, we need to deploy our bundles and start them. In addition to the above three bundles, we need to install and start another two bundles "org.talend.esb.locator" and "org.apache.zookeeper.zookeeper". Since our bundles are dependent on them, execute the following commands sequentially. (Note the bundle numbers returned with each osgi:install command will probably be different from those used in this sample; use those numbers instead.) Then all bundles will be installed and the environment is ready to test:

```
karaf@trun> osgi:install mvn:org.apache.zookeeper/zookeeper/3.3.3
karaf@trun> osgi:install mvn:org.talend.esb/locator/5.1.1
karaf@trun> osgi:install mvn:org.talend.esb.examples.locator/locator_common/1.0.0
karaf@trun> osgi:install mvn:org.talend.esb.examples.locator/locator_service/1.0.0
karaf@trun> osgi:install mvn:org.talend.esb.examples.locator/locator_client/1.0.0
karaf@trun> osgi:start 154
karaf@trun> osgi:start 155
karaf@trun> osgi:start 156
karaf@trun> osgi:start 157
```

Since we want to test the service, we'll leave the locator_client stopped. After executing "osgi:start 157", change to the ZooKeeper console, and you will some log messages like:

```
2011-06-02 16:17:02,031 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn$Factory@251] -
Accepted socket connection from /127.0.0.1:1102
2011-06-02 16:17:02,031 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@777] - Client
attempting to establish new session at /127.0.0.1:1102
2011-06-02 16:17:02,046 - INFO  [SyncThread:0:NIOServerCnxn@1580] - Established session 0x1304f61dba10001
with negotiated timeout 5000 for client /127.0.0.1:1102
```

That means the locator_service tried to connect to the ZooKeeper. If successful, the service will be registered, then we can use the service.

If everything is OK, now, execute "osgi:start 158" on OSGi Container console, you can get some output message in the console:

```
karaf@trun> osgi:start 158
Executing operation greetMe
Message received: MyName

Hello MyName
```

If you get the message like this, then you did the right thing. Turn to zooKeeper console, you can see the connection message like below:

```
2011-06-02 16:24:19,671 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn$Factory@251] -
Accepted socket connection from /127.0.0.1:1126
2011-06-02 16:24:19,671 - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@777] - Client
attempting to establish new session at /127.0.0.1:1126
2011-06-02 16:24:19,703 - INFO  [SyncThread:0:NIOServerCnxn@1580] - Established session 0x1304f61dba10002
with negotiated timeout 5000 for client /127.0.0.1:1126
```

For more test, execute "osgi:stop 158" and the "osgi:start 158" on OSGi Container console. you can get the same result as we described above.

Thus far we showed you how to use Service Locator. In the next section we'll look into Service Activity Monitoring.

5.2. Service Activity Monitoring (SAM)

Service Activity Monitoring (SAM) allows you to log and/or monitor service calls done with the Apache CXF Framework. Typical use cases are usage statistics and fault monitoring. The solution consists of two parts: Agent (sam-agent) and Monitoring Server (sam-server). The Agent creates events out of the requests and replies on service consumer and provider side. The events are first collected locally and then sent to the monitoring server periodically to not disturb the normal message flow. The Monitoring Server receives events from the Agent, optionally filters/handlers events and stores them into a Database. The Agent is packaged as a JAR that needs to be on the classpath of the service consumer and provider. The Monitoring Server is deployed as a WAR in a servlet container and needs access to a database.

Let's look at how to use Service Activity Monitoring in Talend ESB.

5.2.1. Run the Monitoring Server

The Monitoring Server can be run on the Talend Runtime container or in an embedded Jetty server. We'll only cover the Talend Runtime container here. For more details about how to run it on Tomcat or Jetty, please see **Talend ESB Getting Started User Guide**.

The Service Activity Monitoring configuration file is "${Talend-ESB-Version}/container/etc/org.talend.esb.sam.server.cfg". Please edit this file if you want to change the default values:

```
monitoringServiceUrl=/MonitoringServiceSOAP
db.driver=org.apache.derby.jdbc.ClientDriver
db.url=jdbc:derby://localhost:1527/db;create=true
db.username=test
db.password=test
db.incrementer=derbyIncrementer

db.recreate=true
db.createsql=create.sql
```

Then start the OSGi container by running "${Talend-ESB-Version}/container/bin/trun.bat" on Windows or "${Talend-ESB-Version}/container/bin/trun" on Linux. Once the OSGi Container is started, you can setup the server by executing the following commands on the OSGi Console:

```
features:install tesb-derby-starter
features:install tesb-sam-server
features:install tesb-sam-agent
```

The the Database, sam-server and sam-agent will be installed and are ready to used.

5.2.2. Prepare the sample

We'll reuse the sample from Section 3.6, “Test the service”.

In Section 3.7, “Contract-first development”, in order to use the Service Locator, we knew that we needed to add a LocatorFeature. This time we need to add "org.talend.esb.sam.agent.feature.EventFeature".

We'll use a Spring configuration to do that. The simplest way is to copy the "beans.xml" from `examples/talend/tesb/samsam-example-osgi` to `src/main/resources/META-INF/spring/beans.xml` of this project. Below are the main contents:

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/tesb/agent-context.xml" />
  <context:annotation-config/>

  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
     <property name="ignoreUnresolvablePlaceholders" value="true" />
     <property name="location" value="classpath:agent.properties"/>
  </bean>

  <bean id="eventFeature" class="org.talend.esb.sam.agent.feature.EventFeature">
     <property name="mapper" ref="eventMapper" />
     <property name="eventSender" ref="eventCollector" />
     <property name="logMessageContent" value="${log.messageContent}" />
  </bean>

  <bean id="eventMapper" class="org.talend.esb.sam.agent.eventproducer.MessageToEventMapperImpl">
     <property name="maxContentLength" value="${log.maxContentLength}"/>
  </bean>

  <bean id="eventCollector" class="org.talend.esb.sam.agent.collector.EventCollectorImpl" >
     <!-- Default interval for scheduler. Start every X milliseconds a new scheduler -->
     <property name="defaultInterval" value="${collector.scheduler.interval}" />
     <!-- Number of events within one service call. This is a maximum number.
        If there are events in the queue, the events will be processed. -->
     <property name="eventsPerMessageCall" value="${collector.maxEventsPerCall}" />
     <property name="monitoringServiceClient" ref="monitoringServceV1Wrapper" />
     <property name="executor" ref="defaultExecutor" />
     <property name="scheduler" ref="defaultScheduler" />
  </bean>

  <bean id="memoryQueue" class="java.util.concurrent.ConcurrentLinkedQueue"/>
  <task:scheduler id="defaultScheduler" pool-size="2" />
  <task:executor id="defaultExecutor" pool-size="10" />

  <bean id="monitoringServceV1Wrapper"
     class="org.talend.esb.sam.agent.serviceclient.MonitoringServiceWrapper">
     <property name="monitoringService" ref="monitoringServiceV1Client" />
     <!-- Number of retries Default: 5 -->
     <property name="numberOfRetries" value="${service.retry.number}" />
     <!-- Delay in milliseconds between the next attemp to send. Thread is blocked
        for this time. Default: 1000 -->
     <property name="delayBetweenRetry" value="${service.retry.delay}" />
  </bean>

  <bean id="fixedProperties"
     class="org.talend.esb.sam.common.handler.impl.CustomInfoHandler">
     <property name="customInfo">
        <map>
           <entry key="Application name" value="Service2" />
        </map>
     </property>
  </bean>

  <jaxws:endpoint id="customerService" address="/CustomerServicePort"
     implementor="com.example.customerservice.CustomerServiceImpl">
     <jaxws:features>
        <ref bean="eventFeature"/>
     </jaxws:features>
  </jaxws:endpoint>
```

And we also need to give an agent.properties file used to configure sam-agent:

```
collector.scheduler.interval=500
collector.maxEventsPerCall=10

log.messageContent=true
log.maxContentLength=-1

service.url=http://localhost:8040/services/MonitoringServiceSOAP
service.retry.number=3
service.retry.delay=5000
```

We can give a logging.properties file for logger:

```
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler

# Set the default logging level for the root logger
.level = INFO

# Set the default logging level for new ConsoleHandler instances
java.util.logging.ConsoleHandler.level = INFO

# Set the default logging level for new FileHandler instances
java.util.logging.FileHandler.level = ALL

# Set the default formatter for new ConsoleHandler instances
#java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.ConsoleHandler.formatter = org.sopera.monitoring.util.CustomLogFormatter

# Set the default logging level for the logger named com.mycompany
#org.talend.esb.sam.level = FINE
#org.eclipse.persistence.level = INFO
org.talend.esb.sam.level = FINE
org.eclipse.persistence.level = WARNING
```

We also need to change pom.xml to add more dependencies and plugins:

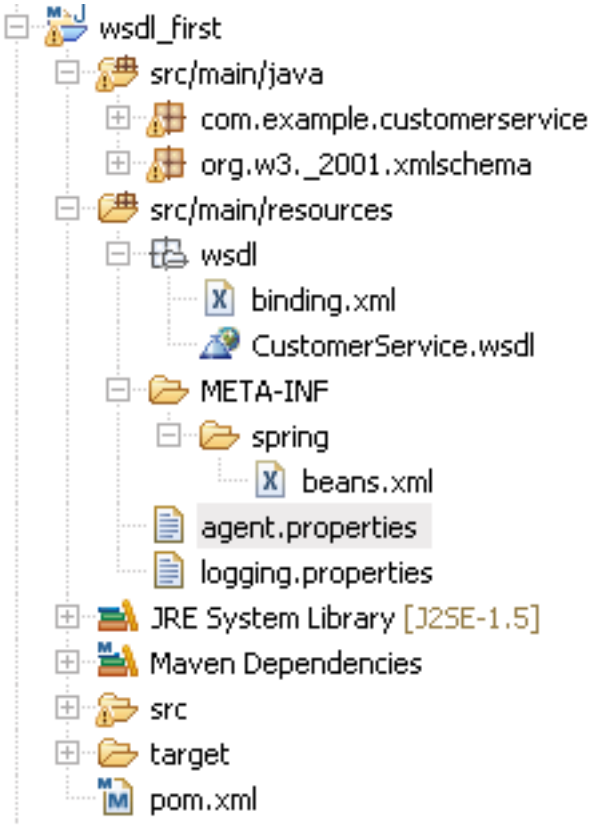
Add a dependency for sam-agent:

```
    <dependency>
        <groupId>org.talend.esb</groupId>
        <artifactId>sam-agent</artifactId>
        <version>5.1.1</version>
      </dependency>
```

Change the configuration of maven-bundle-plugin as following:

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <version>2.3.7</version>
    <configuration>
      <instructions>
        <Bundle-SymbolicName>${pom.groupId}.${pom.artifactId}</Bundle-SymbolicName>
        <Bundle-Name>${pom.name}</Bundle-Name>
        <Bundle-Version>${pom.version}</Bundle-Version>
        <Export-Package>com.example.customerservice</Export-Package>
        <Bundle-Activator>com.example.customerservice.Activator</Bundle-Activator>
        <Require-Bundle>org.apache.cxf.bundle;version="2.6.0",
        org.apache.cxf.bundle,
        org.springframework.beans,
        org.springframework.context,
        sam-agent</Require-Bundle>
      </instructions>
    </configuration>
 </plugin>
```

Now the project structure is:



We finished our configuration, now let's install and deploy it into the OSGi Container, as described in previous chapters.

5.2.3. Test SAM Server

Thus far, we have deployed it, so it can now be tested.

For testing, we can use the sam-example-client in the Talend ESB examples.

The file agent.properties:

```
collector.scheduler.interval=500
collector.maxEventsPerCall=10

log.messageContent=true
log.maxContentLength=-1

service.url=http://localhost:8040/services/MonitoringServiceSOAP
service.retry.number=3
service.retry.delay=5000
```

Spring configuration file contents:

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

  <import resource="classpath:META-INF/tesb/agent-context.xml" />

  <bean id="fixedProperties"
     class="org.talend.esb.sam.common.handler.impl.CustomInfoHandler">
     <property name="customInfo">
        <map>
           <entry key="Application name" value="Client" />
        </map>
     </property>
  </bean>

  <bean class="org.talend.esb.sam.examples.client.CustomerServiceTester">
     <property name="customerService" ref="customerService" />
  </bean>

  <jaxws:client id="customerService"
     address="${serviceUrl}"
     serviceClass="com.example.customerservice.CustomerService">
     <jaxws:features>
        <!-- <bean class="org.apache.cxf.feature.LoggingFeature" />-->
        <ref bean="eventFeature"/>
     </jaxws:features>
  </jaxws:client>
```

Then provide the following code:

```
    //read configuration
    System.setProperty("serviceUrl", "http://localhost:9090/CustomerServicePort/CustomerServiceService");
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("/client.xml");
    CustomerServiceTester tester = context.getBean(CustomerServiceTester.class);
    tester.testCustomerService();
```

As you see we sent the value of "serviceUrl" with service URL, which will fill the contents of bean "customerService". Now we'll turn to the CustomerServiceTester to do a real test:

```
public void testCustomerService() throws NoSuchCustomerException {
    List<Customer> customers = null;

    // First we test the positive case where customers are found and we retrieve
    // a list of customers
    System.out.println("Sending request for customers named Smith");
    customers = customerService.getCustomersByName("Smith");
    System.out.println("Response received");
    Assert.assertEquals(2, customers.size());
    Assert.assertEquals("Smith", customers.get(0).getName());
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e1) {
    }
    // Then we test for an unknown Customer name and expect the NoSuchCustomerException
    try {
        customers = customerService.getCustomersByName("None");
        Assert.fail("We should get a NoSuchCustomerException here");
    } catch (NoSuchCustomerException e) {
        System.out.println(e.getMessage());
        Assert.assertNotNull("FaultInfo must not be null", e.getFaultInfo());
        Assert.assertEquals("None", e.getFaultInfo().getCustomerName());
        System.out.println("NoSuchCustomer exception was received as expected");
    }

    System.out.println("All calls were successful");
}
```

As you see, we did some testing by requesting a customer. Every time when you execute the test, a log message will be recorded into the database we configured in "org.talend.esb.sam.server.cfg". You can use a database viewer to check that, for example DbVisualizer: