**References:**

angelozerr.wordpress.com/2011/08/23/jaxwscxf\_step1

angelozerr.wordpress.com/2011/08/24/jaxwscxf\_step2/

Few months ago, I had to migrate WebServices from my professional project based on [Axis](http://ws.apache.org/axis/java/) to [**JAX-WS**](http://jax-ws.java.net/) which is the **Java API for XML Web Services** supported by the [Java Platform, Standard Edition 6](http://www.oracle.com/technetwork/java/javase/downloads/ea-jsp-142245.html) (Java SE 6).

There are several implementations of JAX-WS like :

* [Glassfish – Metro](http://metro.java.net/).
* [JBoss WS](http://www.jboss.org/jbossws/).
* [Axis2](http://axis.apache.org/axis2/java/core/docs/jaxws-guide.html).
* [Spring JAX-WS](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/remoting.html).
* [Apache CXF](http://cxf.apache.org/) which is based on [XFire](http://xfire.codehaus.org/).

In my case, my professional project doesn’t use [Spring](http://www.springsource.org/). We have chosen [Apache CXF](http://cxf.apache.org/) which **use** [**Spring**](http://www.springsource.org/) **by default** but you can **manage CXF without** [**Spring**](http://www.springsource.org/).

I have **discovered and learned CXF with Eclipse JEE Indigo** (works too with Eclipse JEE Helios) which **provides Wizard to generate JAX-WS annotation and initialize CXF application**. I think that playing with the CXF Eclipse Wizard is a good start point to learn CXF, so I decided to write some articles called [JAX-WS with Apache CXF and Eclipse](http://angelozerr.wordpress.com/jaxwscxf/) (this link is the plan of articles) which explains step by step how to generate WebService and Consumer of WebService with CXF Eclipse Wizard.

In my articles I will use :

* the last version **2.4.2** of CXF.
* **Tomcat 7** as server.
* **Eclipse JEE Indigo**.

But you can do the same thing with another version of CXF and another server. In this article I will explain how to

1. [initialize the CXF Plugin](http://angelozerr.wordpress.com/2011/08/23/jaxwscxf_step1/#InstallCXF)
2. and create an empty [Eclipse Dynamic Web Project with Tomcat 7](http://angelozerr.wordpress.com/2011/08/23/jaxwscxf_step1/#InitializeCXFWebApplication).

**Download**

You can download [jaxwswithcxf\_step1.zip](http://dl.dropbox.com/u/2903680/wordpress-blog/tutoriels/eclipse/cxf/step1/jaxwswithcxf_step1.zip) which is a zip which contains the empty Dynamic Web Project jaxwswithcxf.

**Why CXF?**

**My experimentation with Axis2**

When I started to play with JAX-WS, I decided to test [Axis2](http://axis.apache.org/axis2/java/core/docs/jaxws-guide.html) because I was happy with [Axis](http://ws.apache.org/axis/java/) but I was disappointed with the JAX-WS implementation of Axis2 for serveral reasons :

* documentation is very complex when you are a newbie.
* axis2 distribution provides a lot of JARs and we don’t know which JARs we must select
* default [deploy service](http://axis.apache.org/axis2/java/core/docs/jaxws-guide.html#DeployService) force you to set your WebService JARs in a **servicejars** folder.
* that’s why I tried to implement my own deployer to follow my custom architecure (WebService JARs are stored on another folders) but default axis2 deployer code is not based on JAX-WS [Endpoint#publish(String address, Object implementor)](http://download.oracle.com/javaee/5/api/javax/xml/ws/Endpoint.html#publish%28java.lang.String,%20java.lang.Object%29) to deploy a WebService.
* webservice is not a singleton
* have some bugs with method which don’t have any parameter, etc.

**My experimentation with CXF**

After my Axis2 experimentation, I have noticed that Eclipse JEE Indigo provides a **wizard to generate JAX-WS annotation and initialize a Dynamic Web Project with CXF by using** [**Spring**](http://www.springsource.org/). Unfortunately we have not [Spring](http://www.springsource.org/) in our project, but CXF can work without [Spring](http://www.springsource.org/)! I will explain to you how to do that in the next articles.

Today I’m not disappointed in CXF for several reasons :

* Eclipse JEE Indigo generate well JJAX-WS annotation for the Java class you wish publish as WebService.
* If you can use Spring, the CXF Wizard is very helpfull because it generate you the well web.xml with Spring CXF Servlet and Spring beans which are used to declare classes which must be published as WebService.

[Writing a service with Spring](http://cxf.apache.org/docs/writing-a-service-with-spring.html) explains that.

* If you don’t use Spring, you can still use the CXF Wizard to generate JAX-WS annotation. For the publish of the WebService you need to do that manually with Java code by using JAX-WS API [Endpoint#publish(String address, Object implementor)](http://download.oracle.com/javaee/5/api/javax/xml/ws/Endpoint.html#publish%28java.lang.String,%20java.lang.Object%29) to deploy a WebService :
* MyService serviceInstance = new MyService();
* Endpoint.publish("http://localhost:8080/MyService", serviceInstance);

will deploy your service available at the <http://localhost:8080/MyService>.

[A simple JAX-WS service](http://cxf.apache.org/docs/a-simple-jax-ws-service.html) explains that.

* CXF architecture is based on [CXF Interceptors](http://cxf.apache.org/docs/interceptors.html) (In/Out) to do some process before (In)/after (Out) calling your WebService. You can use existing CXF Interceptor or develop your own Interceptor to manage logging, security, etc.
* CXF Team is very available. As soon as you post your questions/requirements on the CXF Mailing list, you have an answer.

**Install CXF**

CXF distribution provides several [tools](http://cxf.apache.org/docs/tools.html) to generate JAX-WS annotation, WSDL, etc. The CXF Eclipse Plugin use those tools for the CXF wizard.

Installing CXF means :

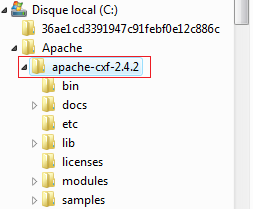
* [download CXF distribution](http://angelozerr.wordpress.com/2011/08/23/jaxwscxf_step1/#DownloadCXF)
* [Initialize CXF Eclipse Plugin](http://angelozerr.wordpress.com/2011/08/23/jaxwscxf_step1/#InitializeCXFPlugin) to set the path of the CXF distribution in the CXF Eclipse Preferences.

**Download CXF distribution**

Go to the [CXF Download](http://cxf.apache.org/download.html) and download the last version of the CXF distribution. In this article **apache-cxf-2.4.2.zip** is used :

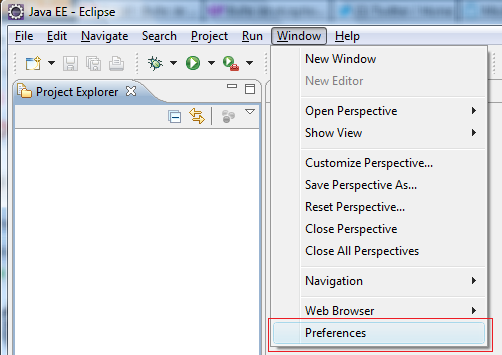


Unzip **apache-cxf-2.4.2.zip** on your Hard Disk wherever you want. In my case I have unzipped to **C:\Apache\apache-cxf-2.4.2** :

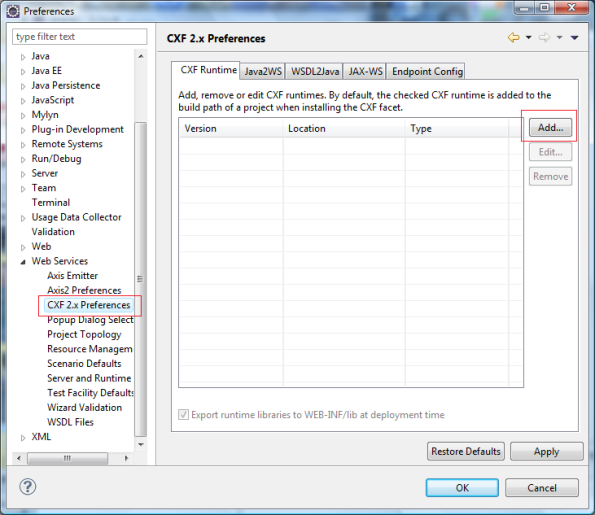


**Initialize CXF Eclipse Plugin**

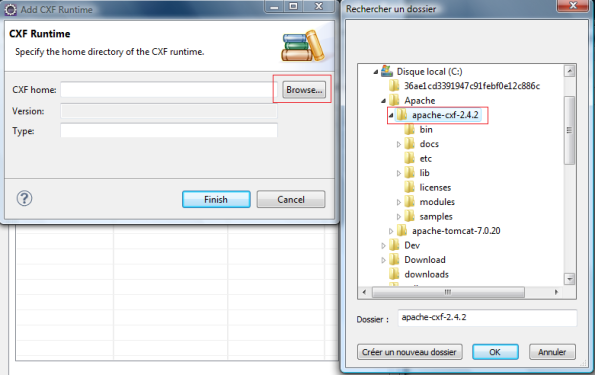
In this section we will set the path of the CXF distribution in the CXF Eclipse Preferences. Open your Eclipse JEE (Helios or Indigo) and go to the **Window / Preferences** menu :



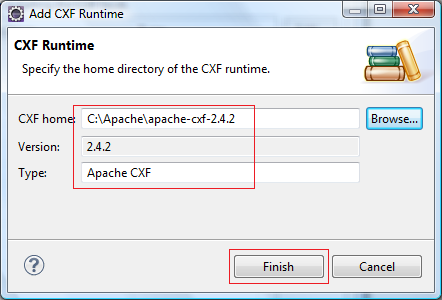
This action opens the **Preferences** dialog. Click on the **Web Services / CXF 2.x Preferences** menu item :



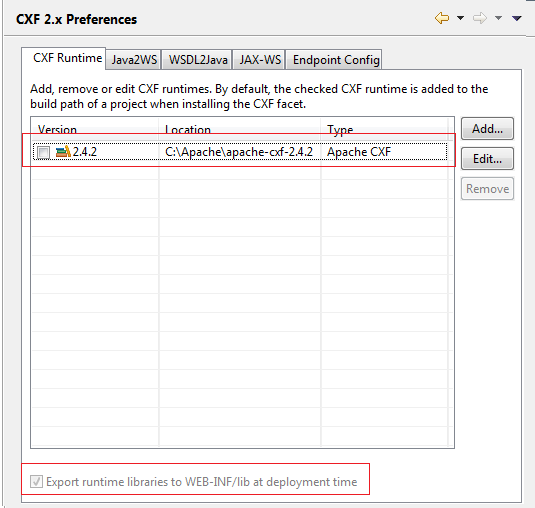
Click on **Add…** button (on the right of the dialog) to select the path of the CXF distribution. This action opens the **Add CXF Runtime**. Add **Browse…** button and select the path of CXF distribution (in my case **C:\Apache\apache-cxf-2.4.2**) :



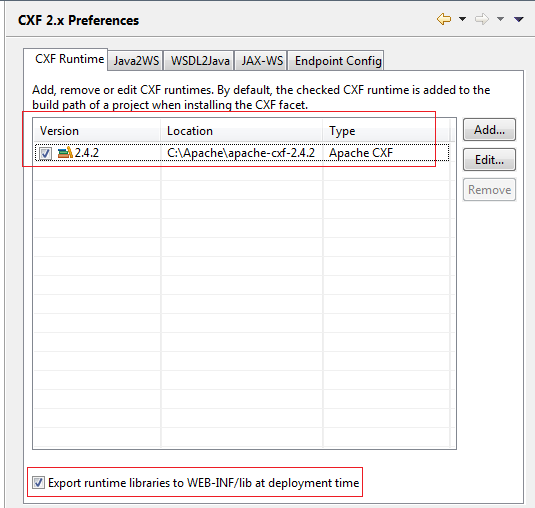
If the path of CXF distribution is valid, the fields of the dialog are filled automatically like this :



Click on **Finish** button, the dialog closes and CXF Runtime tab is filled with the CXF Runtime :



At this step, CXF Runtime is not available. Click on checkbox to make it available :



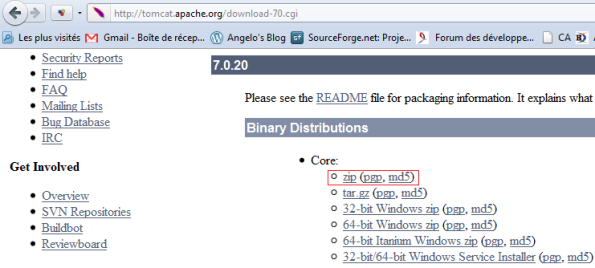
CXF Runtime is now available. In the next article we will see that CXF Runtime will be present in the generic WebServices wizard.

**Initialize CXF Web Application**

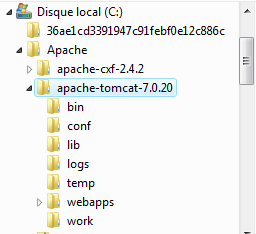
Here we will create an empty **Dynamic Web Project** named **jaxwswithcxf** which will host the JAX-WS WebService.

**Download Tomcat 7**

Go to the [Tomcat 7 Download](http://tomcat.apache.org/download-70.cgi) and download the last version of the Tomcat distribution. In this article **apache-tomcat-7.0.20.zip** is used :

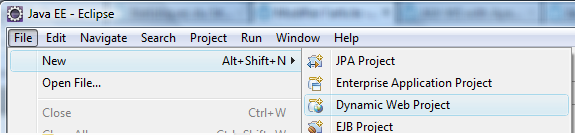


Unzip **apache-tomcat-7.0.20.zip** on your Hard Disk wherever you want. In my case I have unzipped to **C:\Apache\apache-tomcat-7.0.20** :

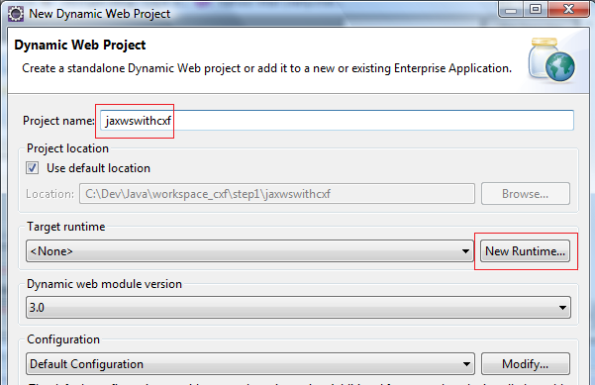


**Create Dynamic Web Project**

In this section we will create an empty **jaxwswithcxf** WEB application with **Dynamic Web Project**. To do that go to menu **File/New /Dynamic Web Project** :

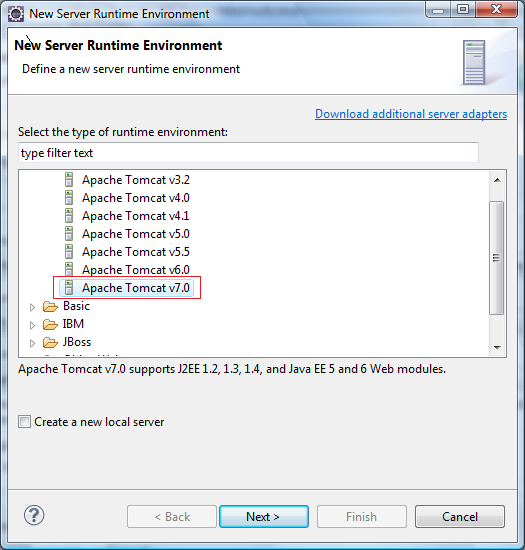


**New Dynamic Web Project** wizard opens. Fill in the project name with **jaxwswithcxf** and select **Target runtime** with your Server. If no server is defined, click on **New Runtime…** button to define it.

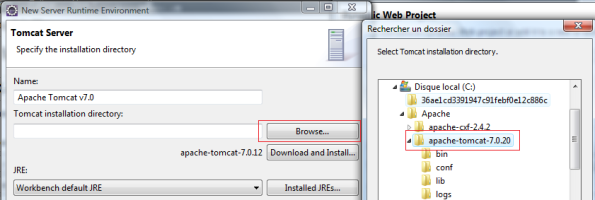


**Define New Server**

Click on **New Runtime…** button, open the wizard to create a server. Select **Tomcat v7.0 Server** :

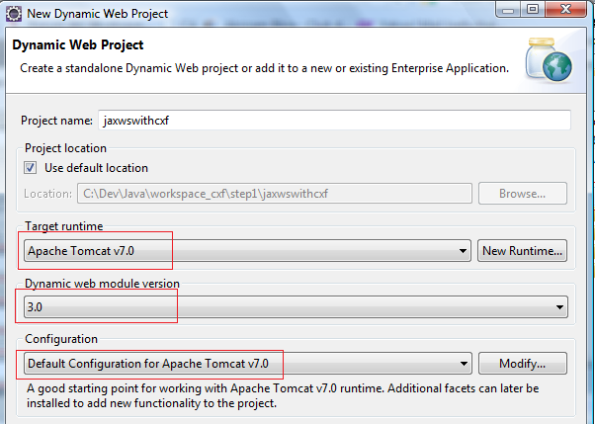


Click on Next button, the wizard page displays the selection of the Tomcat directory (existing Tomcat or empty folder where Tomcat must be installed). Here we will select the installed Tomcat 7. Click on the **Browse…** button to select the Tomcat home (in my case **C:\Apache\apache-tomcat-7.0.20**) :

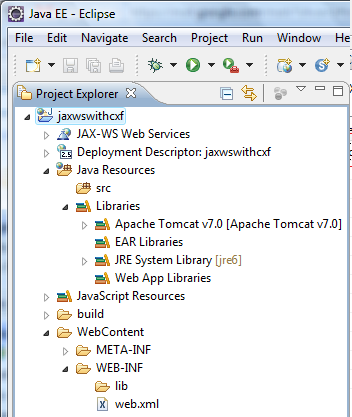


**New Dynamic Web Project configurated**

Click on the **Finish** button, the **New Dynamic Web Project** wizard page appears :

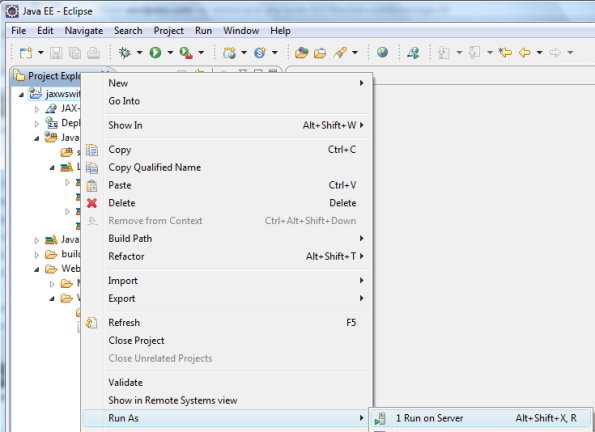


Fields of this wizard are pre-filled like 3.0 Dynamic Module Web version which means that Tomcat 7.0 support 3.0 servlet (you can select another version if you wish). Click on Finish Button, the empty jaxwswithcxf WEB application is generated :

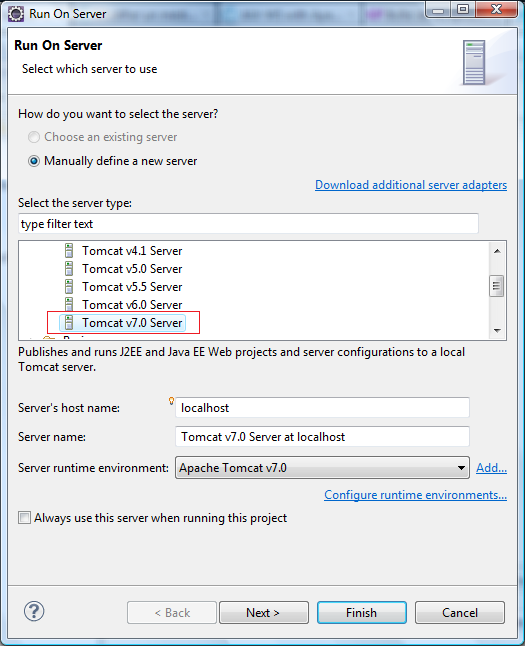


**Launch Empty WEB application**

Now you can start the Tomcat server and launch the empty WEB Application. Select the jaxwswithcxf project, click on right mouse button and select menu **Run As->Run on Server**



This action opens the wizard to select the server :



Select **Tomcat 7.0 server** and click on the **Finish** button. This action create a **Server** folder in your workspace which is the configuration of your Tomcat and launch the **jaxwscxf** WEB Application :



You can notice that you will have a 404 error because the generated web.xml is like this :

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">

<display-name>jaxwswithcxf</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

And your web application contains no such files.

**Conclusion**

In this article we have initialized CXF Eclipse Plugin and created an empty Dynamic Web Project with Tomcat 7.

In the next article [[step2]](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/) we will **create a simple Java class HelloServiceImpl** and we will **use CXF Eclipse wizard to expose it as WebService with JAX-WS and CXF**.

In [[step1]](http://angelozerr.wordpress.com/2011/08/23/jaxwscxf_step1/), we have **configured CXF Eclipse Plugin to use CXF 2.4.2** and created an **empty Dynamic Web project** with **Tomcat 7**.

In this article we will create a sample Java class [HelloServiceImpl](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#HelloServiceImpl) and **publish it as WebService by using CXF Eclipse wizard**. Basicly, this wizard will :

* modify the [HelloServiceImpl to add well-formed JAX-WS annotation.](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#WSHelloServiceImpl)
* modify or create the Spring [beans.xml](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#beans.xml) file which declares the WebServices (in our case HelloServiceImpl) to publish.
* modify the [web.xml](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#web.xml) to declare the **Spring CXF Servlet** used to dispatch to the proper WebService declared in the beans.xml.

**Download**

You can download [jaxwswithcxf\_step2.zip](http://dl.dropbox.com/u/2903680/wordpress-blog/tutoriels/eclipse/cxf/step2/jaxwswithcxf_step2.zip) which is a zip which contains the Dynamic Web Project jaxwswithcxf with the HelloWorldImpl WebServices explained in this article.

**Create Class HelloServiceImpl**

Create in the jaxwswithcxf Dynamic Web Project the class **org.sample.ws.HelloServiceImpl** like this :

package org.sample.ws;

public class HelloServiceImpl {

public String getVersion() {

return "1.0";

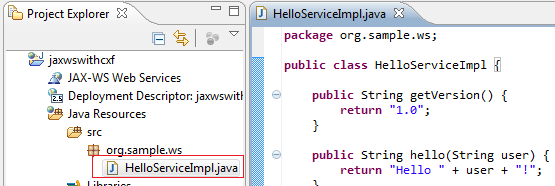
}

public String hello(String user) {

return "Hello " + user + "!";

}

}



**Create WebService HelloServiceImpl**

Web services can be created using two methods:

* **top-down development** : Top-down Web services development involves creating a Web service from a **WSDL file**. WSDL file describes the services (methods, parameters) and defines the contract for the WebService. This WSDL file can also be used to generate the consumer of the WebService (which can be different technolgy (PHP, .Net, etc) than the WebService).
* **bottom-up development** : Bottom-up Web services development involves creating a Web service from a Java bean or enterprise bean.

In our case we will use **bottom-up development** : we will use HelloServiceImpl Java class to create the Web Service.

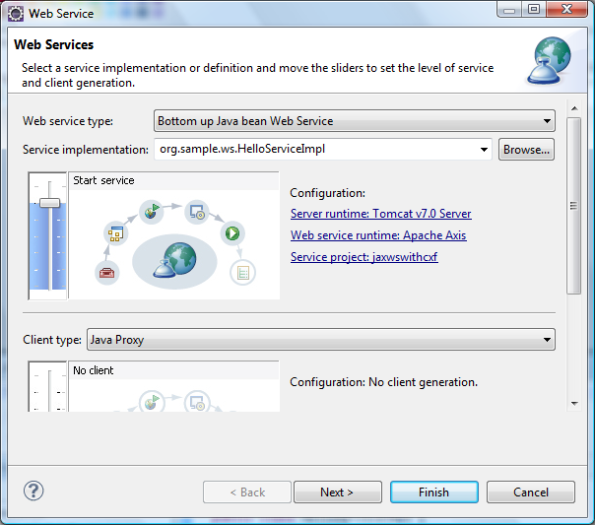
**step1 : select Java class**

Select HelloServiceImpl and click on the right mouse button to click on the **Web Services/Create Web Service** menu item :

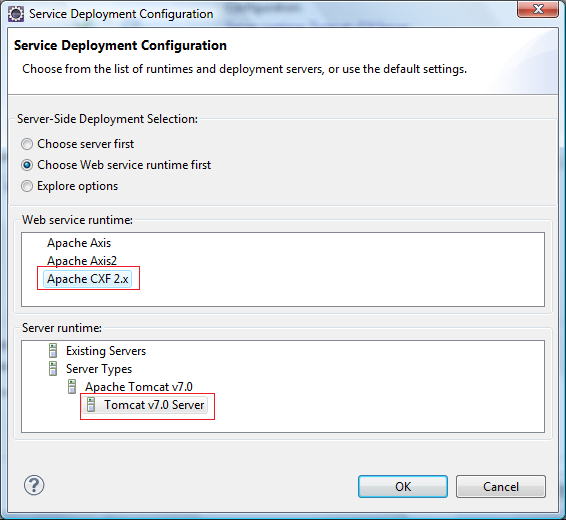
http://angelozerr.files.wordpress.com/2011/08/cxf_helloservice2.png?w=595

**step2 : select CXF runtime**

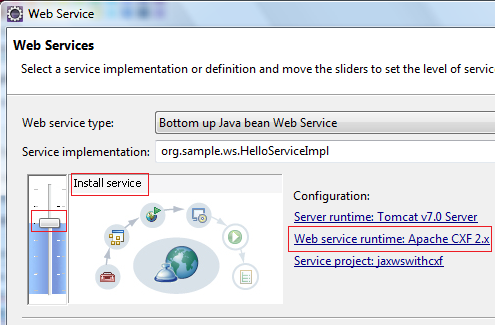
This action opens the generic Wizard **Web Service** to generate WebService with Axis, Axis2, CXF, etc :



In our case we want to generate CXF code. To do that, click on **Web service runtime: Apache Axis**, the **Service Deployment Configuration** opens. Select **Apache CXF 2.x** for Web service runtime :



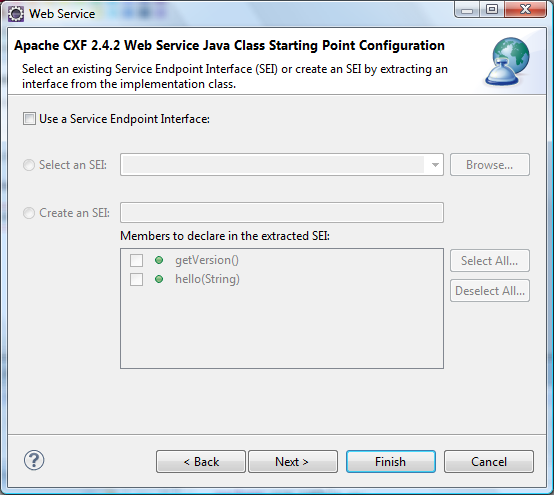
Click on OK button to close the **Service Deployment Configuration** dialog :



You can notice that Web service runtime is **Apache CXF 2.x**. By default the **Start service** is selected which means as soon as wizard is finished, the server will start. I prefer to start my server manually, so I use **Install Service** to avoid launching the server at the end of the wizard :

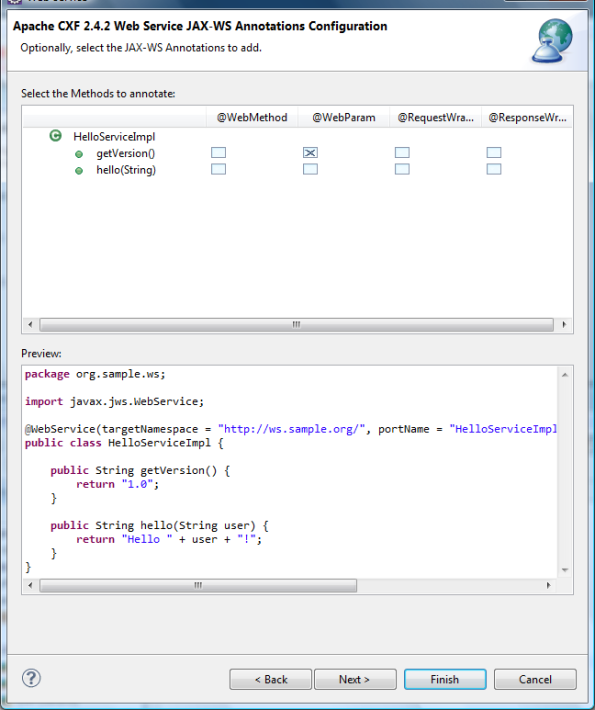
**step3 : Starting Point Configuration**

Click on **Next** button, the **Starting Point Configuration** wizard page displays. This page is used to generate an interface (for the HelloServiceImpl) annoted with JAX-WS annotation. In our case we change nothing and HelloServiceImpl will contain the JAX-WS annotation :



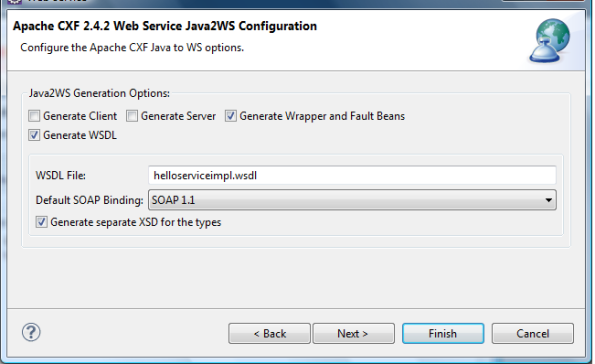
**step4 : Web Service JAX-WS Annotations Configuration**

Click on **Next** button, the **Web Service JAX-WS Annotations Configuration** wizard page displays. This page provides several checkbox that you can select/unselect to generate JAX-WS annotation. You can preview the generated code on the bottom of this page. In our case we don’t change the pre-defined values. We will play with those checkbox in a next article :



**step5 : Web Service Java2WS Configuration**

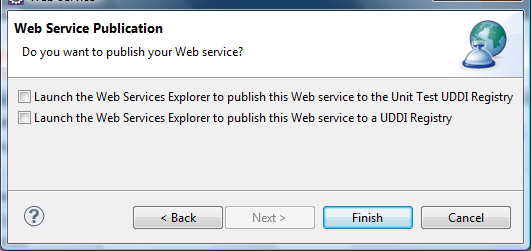
Click on **Next** button, the **Web Service Java2WS Configuration** wizard page displays. This wizard page provides some checkbox to select that you wish generate :



* **Generate client** generate simple client with JAX-WS Service.create()
* **Generate server** generate simple server of the WebService. This code starts a server (with Embedding Jetty) and publish the WebService by using JAX-WS Endpoint#publish.
* **Generate Wrapper and Fault Beans** generate Java class with JAXB annotation. Those classes are generated in the \*.jaxws package (in our case org.sample.ws.jaxws). We will see the use of those classes in the next articles.
* **Generate WSDL** generate WSDL of the WebService.
  + **WSDL file** : name of the WSDL file to generate.
  + **Default SOAP binding**.
  + **Generate seperate XSD for the types** : if this option is selected, the generated WSDL file will not contain the XML Schema which declares the type of the structure used in the method parameters.

**step6 : Web Service Publication**

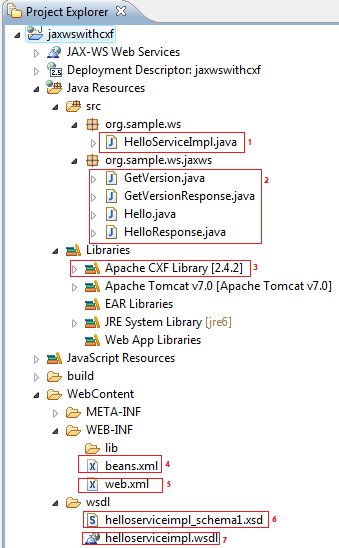
Click on **Next** button, the **Web Service Publication** wizard page displays :



Click on Finish button to generate JAX-WS WebService and CXF component (Spring beans and web.xml).

**Result of CXF Wizard**

After finishing the generation of the CXF wizard, your workspace looks like this :



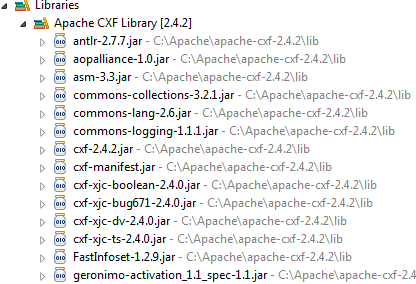
The wizard generate several things :

* **[1] HelloServiceImpl** : modify the Java class [HelloServiceImpl with JAX-WS annotation](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#HelloServiceImpl_JAXWS) :
* package org.sample.ws;
* import javax.jws.WebService;
* @WebService(targetNamespace = "http://ws.sample.org/", portName = "HelloServiceImplPort", serviceName = "HelloServiceImplService")
* public class HelloServiceImpl {
* public String getVersion() {
* return "1.0";
* }
* public String hello(String user) {
* return "Hello " + user + "!";
* }

}

The JAX-WS javax.jws.**WebService** annotation is used to indicate that this class is a WebService.

* **[2] generate some classes** in the **org.sample.ws.jaxws** package. Those classes are generated because the **Generate Wrapper and Fault Beans** of the page [Web Service Java2WS Configuration](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#step5) was checked. At this step, those classes are not used.
* **[3] CXF librarires**. If you open this item you will see that the wizard use the whole JAR of the CXF distribution :



Don’t be afraid of the amount of JARs, CXF needs few JARs.

* **[4] beans.xml** : this Spring file is used to declare with Spring bean the Java class HelloServiceImpl which must be published as WebService :
* <?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-2.5.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/cxf/cxf-extension-soap.xml" />
* <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
* <jaxws:endpoint xmlns:tns="http://ws.sample.org/" id="helloservice"
* implementor="org.sample.ws.HelloServiceImpl" wsdlLocation="wsdl/helloserviceimpl.wsdl"
* endpointName="tns:HelloServiceImplPort" serviceName="tns:HelloServiceImplService"
* address="/HelloServiceImplPort">
* <jaxws:features>
* <bean class="org.apache.cxf.feature.LoggingFeature" />
* </jaxws:features>
* </jaxws:endpoint>

</beans>

* + the following declaration :

<jaxws:endpoint ... implementor="org.sample.ws.HelloServiceImpl"

is used to publish the HelloServiceImpl class.

* + the following declaration :

<jaxws:endpoint ... address="/HelloServiceImplPort"

is used to set the path for the URL of HelloServiceImpl. In our case our service will be available at <http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort>.

* + the following declaration :
  + <jaxws:endpoint ... wsdlLocation="wsdl/helloserviceimpl.wsdl"

set the location of the WSDL. If you remove this wsdlLocation attribute, WSDL will be generated if you access it by the URL <http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort?wsdl>.

* [5] **web.xml** is modified to declare the CXF servlet based on Spring
* <servlet>
* <description>Apache CXF Endpoint</description>
* <display-name>cxf</display-name>
* <servlet-name>cxf</servlet-name>
* <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
* <load-on-startup>1</load-on-startup>
* </servlet>

available at <http://localhost:8080/jaxwswithcxf/services/> :

<servlet-mapping>

<servlet-name>cxf</servlet-name>

<url-pattern>/services/\*</url-pattern>

</servlet-mapping>

This servlet is used to dispatch to the proper WebServices. To know the deployed WebServices, this servlet use the Spring ApplicationContext loaded from the beans.xml :

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>WEB-INF/beans.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

Here is the full code of the web.xml :

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">

<display-name>jaxwswithcxf</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<description>Apache CXF Endpoint</description>

<display-name>cxf</display-name>

<servlet-name>cxf</servlet-name>

<servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>cxf</servlet-name>

<url-pattern>/services/\*</url-pattern>

</servlet-mapping>

<session-config>

<session-timeout>60</session-timeout>

</session-config>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>WEB-INF/beans.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

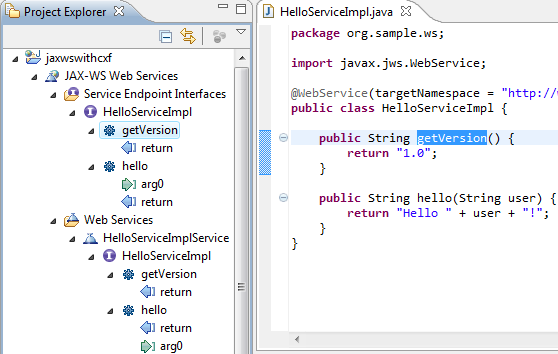
</web-app>

* [6] **helloserviceimpl\_schema1.xsd** : this XML Schema describes the structure used in the WebService. This file is generated because the **Generate seperate XSD for the types** checkbox of the page [Web Service Java2WS Configuration](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#step5) was checked. Here is the content of this file :
* <?xml version="1.0" encoding="utf-8"?><xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://ws.sample.org/" elementFormDefault="unqualified" targetNamespace="http://ws.sample.org/" version="1.0">
* <xs:element name="getVersion" type="tns:getVersion"/>
* <xs:element name="getVersionResponse" type="tns:getVersionResponse"/>
* <xs:element name="hello" type="tns:hello"/>
* <xs:element name="helloResponse" type="tns:helloResponse"/>
* <xs:complexType name="getVersion">
* <xs:sequence/>
* </xs:complexType>
* <xs:complexType name="getVersionResponse">
* <xs:sequence>
* <xs:element minOccurs="0" name="return" type="xs:string"/>
* </xs:sequence>
* </xs:complexType>
* <xs:complexType name="hello">
* <xs:sequence>
* <xs:element minOccurs="0" name="arg0" type="xs:string"/>
* </xs:sequence>
* </xs:complexType>
* <xs:complexType name="helloResponse">
* <xs:sequence>
* <xs:element minOccurs="0" name="return" type="xs:string"/>
* </xs:sequence>
* </xs:complexType>

</xs:schema>

* **[7] helloserviceimpl.wsdl** : is the generated WSDL. This file is generated because the **Generate WSDL** checkbox of the page [Web Service Java2WS Configuration](http://angelozerr.wordpress.com/2011/08/24/jaxwscxf_step2/#step5) was checked. Here is the content of this file :
* <?xml version="1.0" encoding="UTF-8"?>
* <wsdl:definitions name="HelloServiceImplService" targetNamespace="http://ws.sample.org/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://ws.sample.org/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
* <wsdl:types>
* <schema xmlns="http://www.w3.org/2001/XMLSchema">
* <import namespace="http://ws.sample.org/" schemaLocation="helloserviceimpl\_schema1.xsd"/>
* </schema>
* </wsdl:types>
* <wsdl:message name="hello">
* <wsdl:part name="parameters" element="tns:hello">
* </wsdl:part>
* </wsdl:message>
* <wsdl:message name="getVersion">
* <wsdl:part name="parameters" element="tns:getVersion">
* </wsdl:part>
* </wsdl:message>
* <wsdl:message name="helloResponse">
* <wsdl:part name="parameters" element="tns:helloResponse">
* </wsdl:part>
* </wsdl:message>
* <wsdl:message name="getVersionResponse">
* <wsdl:part name="parameters" element="tns:getVersionResponse">
* </wsdl:part>
* </wsdl:message>
* <wsdl:portType name="HelloServiceImpl">
* <wsdl:operation name="getVersion">
* <wsdl:input name="getVersion" message="tns:getVersion">
* </wsdl:input>
* <wsdl:output name="getVersionResponse" message="tns:getVersionResponse">
* </wsdl:output>
* </wsdl:operation>
* <wsdl:operation name="hello">
* <wsdl:input name="hello" message="tns:hello">
* </wsdl:input>
* <wsdl:output name="helloResponse" message="tns:helloResponse">
* </wsdl:output>
* </wsdl:operation>
* </wsdl:portType>
* <wsdl:binding name="HelloServiceImplServiceSoapBinding" type="tns:HelloServiceImpl">
* <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
* <wsdl:operation name="getVersion">
* <soap:operation soapAction="" style="document"/>
* <wsdl:input name="getVersion">
* <soap:body use="literal"/>
* </wsdl:input>
* <wsdl:output name="getVersionResponse">
* <soap:body use="literal"/>
* </wsdl:output>
* </wsdl:operation>
* <wsdl:operation name="hello">
* <soap:operation soapAction="" style="document"/>
* <wsdl:input name="hello">
* <soap:body use="literal"/>
* </wsdl:input>
* <wsdl:output name="helloResponse">
* <soap:body use="literal"/>
* </wsdl:output>
* </wsdl:operation>
* </wsdl:binding>
* <wsdl:service name="HelloServiceImplService">
* <wsdl:port name="HelloServiceImplPort" binding="tns:HelloServiceImplServiceSoapBinding">
* <soap:address location="http://localhost:9090/HelloServiceImplPort"/>
* </wsdl:port>
* </wsdl:service>
* </wsdl:definitions>

You can notice that you have JAX-WS Web Services menu item which shows you information about Java classes which are annotated with JAX-WS :



**Launch WEB Application**

At this step we don’t have any client (consumer of the HelloServiceImpl) but we can test it.

An important information is that our WebService is declared with the org.apache.cxf.feature.**LoggingFeature** :

<jaxws:endpoint id="helloservice"

...>

<jaxws:features>

<bean class="org.apache.cxf.feature.LoggingFeature" />

</jaxws:features>

</jaxws:endpoint>

This feature is very usefull because it gives you the capability to trace for instance the received SOAP message (IN) and the sent SOAP message (OUT). We will check that in the Eclipse Console View.

[Launch jaxwswithcxf](http://angelozerr.wordpress.com/2011/08/23/jaxwscxf_step1#LaunchWEBApplication) to publish our WebService. After the start of the Web Application, you can notice some interesting logs in the Eclipse Console View :

* you can notice that WEB-INF/beans.xml is loaded :
* ...
* INFO: Loading XML bean definitions from ServletContext resource [/WEB-INF/beans.xml]
* ...

And check that HelloServiceImpl is published :

...

INFO: Creating Service {http://ws.sample.org/}HelloServiceImplService from WSDL: wsdl/helloserviceimpl.wsdl

22 août 2011 17:16:13 org.apache.cxf.endpoint.ServerImpl initDestination

INFO: Setting the server's publish address to be /HelloServiceImplPort

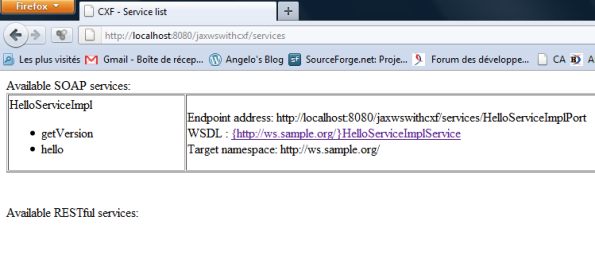
...

**URLs**

At this step we can play with different URLs

**CXF Services list**

If you go at <http://localhost:8080/jaxwswithcxf/services>, you can see the list of WebServices (and REST services) :

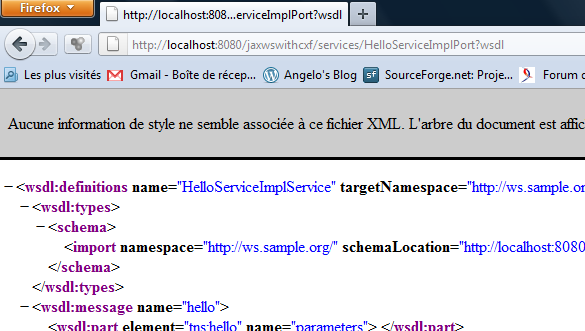


**WSDL**

If you click on the link

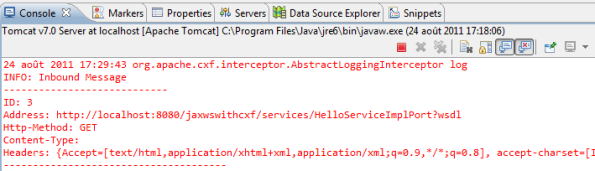
WSDL : [{http://ws.sample.org/}HelloServiceImplService](http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort?wsdl)

you will see the WSDL :



URL of the WSDL is <http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort?wsdl>

If you go to the Eclipse Console View, you will see some logs (because we have LoggingFeatures) :



**getVersion method**

It’s possible to call our getVersion method with the following URL <http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort/getVersion> which returns the SOAP message :

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<ns2:getVersionResponse xmlns:ns2="http://ws.sample.org/">

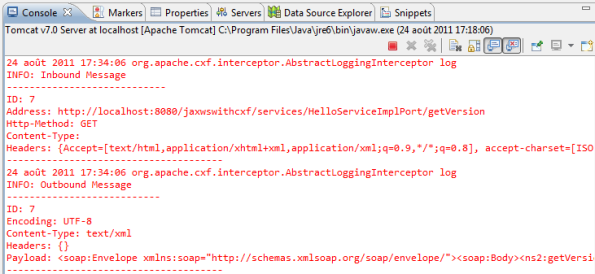
<return>1.0</return>

</ns2:getVersionResponse>

</soap:Body>

</soap:Envelope>

If you go to the Eclipse Console View, you will see some logs (because we have LoggingFeatures)



With this log you can see for instance the sent SOAP Message :

**hello method**

It’s possible to call method with parameter. If you call hello like this, you will that :

<http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort/hello>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<ns2:helloResponse xmlns:ns2="http://ws.sample.org/">

<return>Hello null!</return>

</ns2:helloResponse>

</soap:Body>

</soap:Envelope>

Hello null!

is returned because we have not filled the parameter.

To set a parameter, you must access to this URL :  
<http://localhost:8080/jaxwswithcxf/services/HelloServiceImplPort/hello?arg0=world>

This URL will return the following SOAP message :

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<ns2:helloResponse xmlns:ns2="http://ws.sample.org/">

<return>Hello world!</return>

</ns2:helloResponse>

</soap:Body>

</soap:Envelope>

Why **arg0**? Because if you check the XML Schema you have :

<xs:complexType name="hello">

<xs:sequence>

<xs:element minOccurs="0" name="arg0" type="xs:string"/>

</xs:sequence>

</xs:complexType>

**Conclusion**

In this article we have **generated with CXF Eclipse wizard a WebService with JAX-WS.** This wizard initializes CXF by creating beans.xml wich declare Java classes which must be published and CXF Servlet. You can notice that HelloServiceImpl is not linked to CXF and could be used with another JAX-WS implementation.

In next article [step3] we will **create and generate a client with JAX-WS** (and CXF) which will consume our WebService.

**How to add WS Security:**

* 1. Add spring configuration in the configuration file
  2. Add one ServerPasswordCallback class

Server callbacks verify passwords by supplying the password with which the incoming password will be compared.

import java.io.IOException;

import javax.security.auth.callback.Callback;

import javax.security.auth.callback.CallbackHandler;

import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class ServerPasswordCallback implements CallbackHandler

{

public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {

WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

if (pc.getIdentifer().equals("joe")) {

// set the password on the callback. This will be compared to the

// password which was sent from the client.

pc.setPassword("password");

}

}

}

USERNAME\_TOKEN

Obtain the password for UsernameToken credentials. This usage code is used both on the client side (to obtain a password to send to the server) and on the server side (to obtain a password in order to compare it with the password received from the client).

On the server side, this code is set in the following cases:

* *Digest password*—if the UsernameToken contains a digest password, the callback must return the corresponding password for the given user name (given by WSPasswordCallback.getIdentifier()). Verification of the password (by comparing with the digest password) is done by the WSS4J runtime.
* *Plaintext password*—implemented the same way as the digest password case (since Fuse Services Framework 2.4.0).
* *Custom password type*—if getHandleCustomPasswordTypes() is true on org.apache.ws.security.WSSConfig, this case is implemented the same way as the digest password case (since Fuse Services Framework 2.4.0). Otherwise, an exception is thrown.

If no Password element is included in a received UsernameToken on the server side, the callback handler is not called (since Fuse Services Framework 2.4.0).

<http://localhost:8080/WebServicesPOC/services>

<http://localhost:8080/WebServicesPOC/services/UserServicePort?wsdl>