SUNDAY MARCH 12, 2017

Creating a WSDL-first web service with Apache CXF

See my blog article index for other web service tutorials.

This tutorial shows how to create a WSDL-first web service using Apache CXF 3.1.10 running on Tomcat 8 or more recent. We'll be using Apache Maven as our build tool. We'll also create a SOAP client that calls the web service, which takes an integer from the SOAP request and returns the number doubled in value. The finished tutorial source code can be obtained from GitHub by using either the download ZIP button or `git clone -v git://github.com/gmazza/blog-samples.git` command. Before proceeding, Maven will need to be installed on your machine and its `bin` directory placed on your operating system's path.

1. **Create the project file structure.** Following Maven's standard directory layout, create a "web_service_tutorial" folder and from there copy-and-paste the appropriate set of directory creation commands:

```
for Linux:
mkdir -p client/src/main/java/client
mkdir -p service/src/assembly
mkdir -p service/src/main/java/service
mkdir -p service/src/test/java/service
mkdir -p service/src/main/resources
mkdir -p war/src/main/webapp/WEB-INF

for Windows:
mkdir client\src\main\java\client
mkdir service\src\assembly
mkdir service\src\main\java\service
mkdir service\src\test\java\service
mkdir service\src\main\resources
mkdir war\src\main\webapp\WEB-INF
```

2. **Create and configure the Maven pom files.** The following files will need to be added to the specified locations.

   web_service_tutorial/pom.xml: This is the parent pom file declaring common dependencies and plugins used by the submodules (service, war, and client).

```
<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/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>org.gmazza.blog-samples.web-service-tutorial</groupId>
```


CALENDAR

« March 2017

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	**25**
26	27	28	29	30	31	

Today

ABOUT ME

POLITICO Tech Team
Apache CXF and TightBlog committer
Arlington, Virginia USA
gmazza at apache dot org

LINKS

POLITICO
POLITICO - EU
WordPress Blog
Brightspot CMS
Ghost Blog
Jekyll News
Spring Blog
Kohei Nozaki
Rob Winch

LAST 20

Using Apache CXF to access Salesforce's SOAP API
Creating a WSDL-first web service with Apache CXF
Using AppleScript to quickly configure your work environment
TightBlog 2.0 update...

```
    <artifactId>web-service-tutorial</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Tutorial: Basic WSDL-First Web Service</name>
    <packaging>pom</packaging>
    <url>http://web-gmazza.rhcloud.com/blog/entry/web-service-tutorial</url>

    <modules>
        <module>service</module>
        <module>war</module>
        <module>client</module>
    </modules>

    <prerequisites>
        <maven>3.3.9</maven>
    </prerequisites>
```

web_service_tutorial/pom.xml view raw

web_service_tutorial/service/pom.xml: This pom file generates the JAX-WS artifacts using CXF's wsdl2java tool that will be referenced by the web service provider and SOAP client. The Maven Assembly Plugin is used here to create a subset of the full service JAR, containing the JAX-WS artifacts and the WSDL but not the web service implementation, for use by the SOAP client. The `packaging` element is set to `jar`, to be used as the main dependency in the `war` submodule.

JUnit is also included as a test-time dependency in order to run unit tests of the web service provider, as we'll see later in this tutorial.

```
<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/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.gmazza.blog-samples.web-service-tutorial</groupId>
        <artifactId>web-service-tutorial</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <artifactId>web-service-tutorial-service</artifactId>
    <name>-- Web Service Provider</name>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

web_service_tutorial/service/pom.xml view raw

Benefits of Retaining the Electoral College
TightBlog: August 2016 update
TightBlog: July 2016 Update
First release - TightBlog v1.0.0 available!
New user administration functionality in TightBlog
TightBlog: June 2016 Update
TightBlog: May 2016 Update
TightBlog: April 2016 Update
TightBlog: March 2016 Update
TightBlog: February 2016 Update
TightBlog: January 2016 Update
TightBlog: December 2015 update
Working with TightBlog source code
TightBlog: November 2015 update
Pink Slips at Disney. But First, Training Foreign Replacements.
Letter to the Arlington County board via fitness membership fees (Update: With County Response)

RECENT $0.02

- Glen on Using AppleScript to quickly configure your work environment
- Anant Jaynarayan on Using AppleScript to quickly configure your work environment
- Ananth Jayanarayana on TightBlog: August 2016 update
- Glen on First release - TightBlog v1.0.0 available!
- Imby on First release - TightBlog v1.0.0 available!
- Glen on First release - TightBlog v1.0.0 available!
- Glen Mazza on First release - TightBlog v1.0.0 available!
- Imby on First release - TightBlog v1.0.0 available!
- Amit Joshi on First release - TightBlog v1.0.0 available!
- Glen on New user administration functionality in TightBlog

FEEDS

All
Web Services
Blogs & Wikis
Misc Tech
Other
Comments

NAVIGATION

Front Page
Weblog
Login

web_service_tutorial/service/src/assembly/jaxws-jar.xml: The Maven Assembly Plugin uses this file to create a JAR containing the JAX-WS generated artifacts and the WSDL file.

```
<assembly>
   <id>jaxws</id>
   <formats>
      <format>jar</format>
   </formats>
   <includeBaseDirectory>false</includeBaseDirectory>
   <fileSets>
      <fileSet>
         <directory>target/classes</directory>
         <outputDirectory>/</outputDirectory>
         <includes>
            <include>DoubleIt.wsdl</include>
            <include>org/**</include>
         </includes>
      </fileSet>
   </fileSets>
</assembly>
```

web_service_tutorial/war/pom.xml: This pom creates the WAR file that will host the web service on Tomcat.

```
<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/maven-v4_0_0.xsd">

   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>org.gmazza.blog-samples.web-service-tutorial</groupId>
      <artifactId>web-service-tutorial</artifactId>
      <version>1.0-SNAPSHOT</version>
   </parent>
   <artifactId>web-service-tutorial-war</artifactId>
   <name>-- Service WAR file</name>
   <packaging>war</packaging>

   <dependencies>
      <dependency>
          <groupId>org.gmazza.blog-samples.web-service-tutorial</groupId>
          <artifactId>web-service-tutorial-service</artifactId>
          <version>1.0-SNAPSHOT</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
```

web_service_tutorial/war/pom.xml view raw

web_service_tutorial/client/pom.xml: This pom file includes as a dependency the JAX-WS artifact jar created above and uses the Maven Exec Plugin to activate the SOAP client.

```
            <configuration>
               <executable>java</executable>
               <arguments>
                  <argument>-classpath</argument>
                  <classpath/>
                  <!-- Uncomment below for debug output with CXF.  Will need to configure
                          client-side interceptors and supply a java.util.logging properties file:
                          http://cxf.apache.org/docs/debugging-and-logging.html
                          Place the logging file in the same directory as this pom file.
                   -->

                  <!--argument>
                     -Djava.util.logging.config.file=mylogging.properties
                  </argument-->
                  <argument>
                     client.WSClient
                  </argument>
               </arguments>
            </configuration>
         </plugin>
      </plugins>
   </build>
</project>
```

web_service_tutorial/client/pom.xml view raw

3. **Create the web service WSDL and generate the JAX-WS artifacts.** Place the below DoubleIt.wsdl file into the `service/src/main/resources` directory. The endpoint used in defined in the soap:address location in the wsdl:service section, see the notes section at the bottom for information on how to use a different endpoint URL for your own projects.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="DoubleIt"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:di="http://www.example.org/schema/DoubleIt"
   xmlns:tns="http://www.example.org/contract/DoubleIt"
   targetNamespace="http://www.example.org/contract/DoubleIt">
   <wsdl:types>
      <xsd:schema targetNamespace="http://www.example.org/schema/DoubleIt">
         <xsd:element name="DoubleIt">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="numberToDouble" type="xsd:int"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="DoubleItResponse">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="doubledNumber" type="xsd:int" />
```

```
</xsd:sequence>
```

web_service_tutorial/service/src/main/resources/DoubleIt.wsdl view raw

4. **Create the servlet deployment descriptor (web.xml) file.** Place the file in the `war/src/main/webapp/WEB-INF` directory of the project.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <display-name>Sample web service provider</display-name>
    <listener>
        <listener-class>
             org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
              classpath:META-INF/cxf/cxf.xml
        </param-value>
    </context-param>
    <servlet>
        <servlet-name>WebServicePort</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
```

web_service_tutorial/war/src/main/webapp/WEB-INF/web.xml view raw

Note for the CXF configuration above, the `META-INF/cxf/cxf.xml` initialization file referenced is not part of the tutorial code but actually included in the CXF framework. Next, to verify the pom.xml files and generate the JAX-WS and JAXB artifacts, enter `mvn clean install` from the root web_service_tutorial directory.

5. **Create the web service configuration file.** The CXF configuration file works in conjunction with (and takes precedence over) the @WebService annotation on the web service provider. CXF uses a Spring application context for this. Copy the following cxf-servlet.xml file to the `war/src/main/webapp/WEB-INF` directory.

(CXF only) cxf-servlet.xml:

```
 <?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 statement optional but often required if you're declaring
          other CXF beans in this file such as CXF's JMX MBeans -->
```

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

    <jaxws:endpoint
        id="doubleit"
        implementor="service.DoubleItPortTypeImpl"
        wsdlLocation="WEB-INF/wsdl/DoubleIt.wsdl"
        address="/doubleit">
```

6. **Create the Web Service Provider.** This class is known as either the SEI (Service Endpoint Interface) implementation or Service Implementation Bean (SIB). The SEI is the DoubleItPortType class that was generated from the WSDL earlier. The methods in the SEI map to the operations defined in the portType section of the WSDL.

   Place this DoubleItPortTypeImpl.java file in the `service/src/main/java/service` directory:

```
package service;

import javax.jws.WebService;
import org.example.contract.doubleit.DoubleItPortType;

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

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

   While not required, we should also create unit tests of our web service implementation in which we check before deploying the SIB that its methods are properly implemented (e.g., doubleIt is not erroneously tripling incoming numbers). Sample test cases for this SIB are available in the software download and placed in the same Java package as the class we're testing albeit in a different folder location (per Maven convention, `service/src/test/...` instead of `service/src/main/...`). Doing it this way reduces the need for Java import statements in the test cases of what we're testing while still keeping test code out of deployment JARs. (You may later wish to check this blog article for more advanced integration testing via actual SOAP calls against a deployed web service.)

   Now that we have the SIB and its unit test cases available run `mvn clean install` from either the project root or service directory. The JUnit tests will be automatically detected and run before any JARs are created. If there's any failure in the test cases the build will halt, requiring you to fix the SIB prior to re-running the build process. If failures occur, check the `service/target/surefire-reports` folder that will be created for detailed test results.

7. **Build and deploy the WAR file.** This step is different depending on whether you're deploying to Tomcat embedded (in-memory) or Tomcat standalone:

   Tomcat embedded deployment: No special configuration or manual download of Tomcat needed. The Maven `tomcat7:run-war` command will download and run an in-memory Tomcat instance and host the web service WAR on it. This is a very simple and quick way to test the web service works without needing to bother with Tomcat installation or configuration. It is configured to run on localhost:8080 so make sure you don't have any other servers presently occupying that address.

Tomcat standalone deployment: If you haven't worked with Tomcat before, here are some basic instructions for installing and configuring this servlet container. The Tomcat user's guide and mailing list can provide more help if needed.

1. Download the latest release version of Tomcat and extract the zip or tar.gz file into a new directory.
2. Have an environment variable $CATALINA_HOME point to your expanded Tomcat application directory, e.g. for Linux (in your ~/.bashrc file): `export CATALINA_HOME=/usr/myapps/tomcat-8.x.x`
3. In the CATALINA_HOME/conf/tomcat-users.xml file, create a user with the role of `manager-script` or give the "tomcat" user (which should already be in the file) the `manager-script` role as shown below. This role allows for deploying web applications using scripting tools such as the Tomcat Maven Plugin we're using in this tutorial. (Although not covered in this tutorial, if you would like to do WAR deployment using the Tomcat Manager browser application, grant the `manager-gui` role instead.)

```
<tomcat-users>
  ...other entries...
  <role rolename="manager-script"/>
  <role rolename="manager-gui"/>
  <user username="tomcat" password="????" roles="tomcat,manager-script"/>
</tomcat-users>
```

4. Start Tomcat from a console window: `{prompt}% $CATALINA_HOME/bin/startup.sh`
5. If you granted the user the `manager-gui` role, confirm that you can log into the manager webapp at `http://localhost:8080/manager/html` using the username and password of the manager account.

For Maven to be able to deploy and undeploy webapps on standalone Tomcat, you'll need to add a server entry to your $MAVEN_HOME/conf/settings.xml file. Add the username and password for the manager account as follows:

```
<servers>
   ...
   <server>
         <id>myTomcat</id>
         <username>tomcatuser</username>
         <password>password</password>
   </server>
</servers>
```

The war submodule's pom.xml file uses the "id" element to indicate the app server it will be deploying to.

Next, to deploy the web service onto Tomcat:

- For embedded Tomcat, simply navigate to the `web_service_tutorial` or `web_service_tutorial/war` directory and enter **`mvn clean install tomcat7:run-war`**. (I recommend `run-war` over `run` due to some liberties the latter takes in determining the WAR's classpath.)
- For standalone Tomcat, navigate to the `web_service_tutorial` or `web_service_tutorial/war` directory and enter **`mvn clean install tomcat7:redeploy`**. (See the list of other goals available such as `tomcat7:deploy` and `tomcat7:undeploy`.) This single command string does all of the following:
  - Deletes the service/target output folder if it already exists (same as `mvn clean` alone)
  - Recreates the folder and generates and compiles the Java classes into it (`mvn package`)
  - Creates a WAR file for the web service provider and JAR file for the JAX-WS artifacts (`mvn package`)
  - Installs the JAR and WAR into your local Maven repository (`mvn install`)
  - Undeploys the previous WAR file if it previously existed (`tomcat7:redeploy`)
  - Deploys the new WAR file onto Tomcat. (`tomcat7:redeploy`)

After deployment, make sure you can access the web service's wsdl at `http://localhost:8080/doubleit/services/doubleit?wsdl` before proceeding further. If the browser reports an error message instead, check the output from Tomcat's console window and/or server logs (in $CATALINA_HOME/logs) for troubleshooting clues.

If you're having any difficulty deploying the web service, StackOverflow of course and the Apache CXF mailing list are good sources for help, CXF also has a chatroom, #apache-cxf@irc.freenode.net.

8. **Create the web service client.** The following client tests the DoubleIt operation declared in the WSDL. Place the below WSClient.java file in the `client/src/main/java/client` directory.

```
package client;

import org.example.contract.doubleit.DoubleItPortType;
import org.example.contract.doubleit.DoubleItService;

public class WSClient {

    public WSClient() {
    }

    public static void main (String[] args) {
        DoubleItService service = new DoubleItService();
        DoubleItPortType port = service.getDoubleItPort();
        makeCalls(port);
    }

    private static void makeCalls(DoubleItPortType port) {
        doubleIt(port, 10);
        doubleIt(port, 0);
```

9. **Run the client code.** First run `mvn clean install` from the client (or parent) submodule to compile the above class. Then run `mvn exec:exec` from the client submodule.

   You should see the following results:

```
[INFO] The number 10 doubled is 20
[INFO] The number 0 doubled is 0
[INFO] The number -10 doubled is -20
```

   To activate client-side logging, the client/pom.xml file provides options that can be activated.

Additional notes:

1. For naming the elements in the WSDL, I followed the conventions used by GlassFish Metro's AddNumbers.wsdl in its "fromwsdl" sample. In particular, given the stem of the WSDL name (DoubleIt), this is what I used:

   - name of wsdl:portType: DoubleIt**PortType**
   - name of wsdl:binding: DoubleIt**Binding**
   - name of wsdl:service: DoubleIt**Service**
   - name of port under wsdl:service: DoubleIt**Port**

JAX-WS WSDL-to-Java tools use these names when generating the JAX-WS classes, so following this convention helps keep the classes clearly named and understandable.

2. The web service endpoint URL specified in the WSDL (in the wsdl:service section) is used only by the SOAP client, not the web service provider. The DoubleItService.java class used by the client has a default reference to the local WSDL from where the endpoint URL is read. For the provider, the web service stack overrides this URL with a value appropriate for the WAR's configuration. What follows is the algorithm a CXF web service provider uses to determine the endpoint URL for a web service.

```
http://localhost:8080/doubleit/services/doubleit?wsdl
          A              B        C          D
```

   - A is the host and port of the servlet container.
   - B is the name of the war file.
   - C comes from the url-pattern element in the web.xml file.
   - D comes from the address attribute in the cxf-servlet.xml file.

   Note: If you're hosting the web service behind an HTTP server (that will internally forward the SOAP call to the web service provider hosted elsewhere), CXF's wsdl2java tool offers the publishedEndpointUrl setting to specify the web service URL that will appear in the WSDL's soap:address when clients make the WSDL-requesting "?wsdl" HTTP call. Note this does not change the actual endpoint URL where the SOAP service is hosted, but just gives a different address (to the HTTP server) for clients to use.

3. The optional `wsdllocation` element configured for CXF's wsdl-to-java tool in the service/pom.xml results in the subsequently generated DoubleItService.java class using a classpath reference to the WSDL instead of one hardcoded to the local file system. As the WSDL gets packaged in the JAR files created, this allows you to run the web service and client from machines different from the one where you created the application.

4. For developer convenience, the signatures of the generated service methods can sometimes be changed between "wrapper style" and "non-wrapper style" by setting a JAX-WS binding property as explained here.

Posted at 07:00AM Mar 12, 2017 by Glen Mazza in Web Services | Comments [0]

COMMENTS:

POST A COMMENT:

Name:

E-Mail:

URL:

Notify me by email of new comments

Remember Information?

Your Comment:

HTML Syntax: ***NOT allowed***

Please answer this simple math question

3 + 38 =

Preview Post

Valid HTML! Valid CSS!