[How to create a WSDL-first SOAP client in Java with CXF and Maven](http://www.logicsector.com/java/how-to-create-a-wsdl-first-soap-client-in-java-with-cxf-and-maven/" \o "Permanent Link to How to create a WSDL-first SOAP client in Java with CXF and Maven)

On 6 Dec 2011, in [java](http://www.logicsector.com/category/java/), by Justin

*This article was originally written in October 2008 and posted on my old blog at*[*wordpress.com*](http://logicsector.wordpress.com/2008/10/19/how-to-create-a-wsdl-first-soap-client-in-java-with-cxf-and-maven/)*. I’ve retired that blog, so I’ve reposted the original contents here in the hope someone finds it useful. As of December 2011 the code still works great. Note I’m an iOS developer nowadays, not a Java developer, so regretfully*I can’t offer any support nor updates if it doesn’t work for you*. Please feel free to read, comment, and link to the article here at its new home.*

Background

About a week ago I needed to write a SOAP-based client for work. The SOAP framework I’m using is Apache CXF. I’m a total noob when it comes to SOAP services, and so I was a little apprehensive about this at first. My apprehension sprung from hearing horror stories a few years ago from coworkers who were writing Axis SOAP applications, and they were basically tearing their hair out over Axis.

However, word has it that CXF is much easier to use. Well, it took me a while to get it working correctly. In an effort to save other folks the same grief, I’ve posted my code here. If you’re reading this, I’m assuming you’re acquainted with Java and Maven, but fairly new to SOAP, WSDL, etc.

To keep things simple, I decided to write a “Hello World” type of application first to make sure I could get the technology stack working correctly. To keep things *really* simple, I decided to create a trivial Java “main” function that calls a SOAP service, logs the result to the console, and exits (no fancy web interface or anything like that).

The SOAP Service Provider

First I had to select an appropriate web service to test against. There are a bunch of free SOAP-based web services out there, and I chose the CDyne weather service. You can go [read all about it](http://wiki.cdyne.com/wiki/index.php?title=CDYNE_Weather) if you want.

Obtain the Service’s WSDL

When you’re writing a new client in CXF for an existing web service, you start with the WSDL and work from there. This means you need to get a copy of the WSDL from the service provider. The WSDL for the CDyne weather service can be [downloaded from their site](http://ws.cdyne.com/WeatherWS/Weather.asmx?wsdl). You can simply right-click that link and save the WSDL on your hard drive.

Once you have the WSDL in hand, you can build your client around it. Basically, you’ll use a CXF tool called *wsdl2java* to turn the WSDL into Java stub code that you then compile along with your application.

Create the Maven Project

As a recent convert to Maven, I set up a new Maven project. I created a new project directory calledweather-client, which is the *${basedir}*. Also, I put the WSDL file in${basedir}/src/main/wsdl/weather.wsdl.

Yeah, I know Maven has its fancy archetype creator thingie to emit the initial POM file, but like most pragmatic programmers I simply copy and paste a similar POM from somewhere else and modify it to suit my needs. Here’s the project file I came up with.

weather-client/**pom.xml**

|  |  |
| --- | --- |
| 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  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109 | <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>com.logicsector</groupId>      <artifactId>weather-client</artifactId>      <version>1.0</version>      <name>SOAP weather client</name>      <packaging>jar</packaging>        <dependencies>          <dependency>              <groupId>org.apache.cxf</groupId>              <artifactId>cxf-rt-frontend-jaxws</artifactId>              <version>2.1.2</version>          </dependency>          <dependency>              <groupId>org.apache.cxf</groupId>              <artifactId>cxf-rt-transports-http</artifactId>              <version>2.1.2</version>          </dependency>          <dependency>              <groupId>org.slf4j</groupId>              <artifactId>slf4j-api</artifactId>              <version>1.5.2</version>          </dependency>          <dependency>              <groupId>org.slf4j</groupId>              <artifactId>slf4j-log4j12</artifactId>              <version>1.5.2</version>          </dependency>      </dependencies>        <build>          <finalName>weather-client</finalName>          <plugins>              <!-- Generate Java classes from WSDL during build -->              <plugin>                  <groupId>org.apache.cxf</groupId>                  <artifactId>cxf-codegen-plugin</artifactId>                  <version>2.1.2</version>                  <executions>                      <execution>                          <id>generate-sources</id>                          <phase>generate-sources</phase>                          <configuration>                              <sourceRoot>${basedir}/target/generated/src/main/java</sourceRoot>                              <wsdlOptions>                                  <wsdlOption>                                      <wsdl>${basedir}/src/main/wsdl/weather.wsdl</wsdl>                                      <extraargs>                                          <extraarg>-client</extraarg>                                      </extraargs>                                  </wsdlOption>                              </wsdlOptions>                          </configuration>                          <goals>                              <goal>wsdl2java</goal>                          </goals>                      </execution>                  </executions>              </plugin>              <!-- Add generated sources - avoids having to copy generated sources to build location -->              <plugin>                  <groupId>org.codehaus.mojo</groupId>                  <artifactId>build-helper-maven-plugin</artifactId>                  <executions>                      <execution>                          <id>add-source</id>                          <phase>generate-sources</phase>                          <goals>                              <goal>add-source</goal>                          </goals>                          <configuration>                              <sources>                                  <source>${basedir}/target/generated/src/main/java</source>                              </sources>                          </configuration>                      </execution>                  </executions>              </plugin>              <!-- Build the JAR with dependencies -->              <plugin>                  <artifactId>maven-assembly-plugin</artifactId>                  <configuration>                      <descriptorRefs>                          <descriptorRef>jar-with-dependencies</descriptorRef>                      </descriptorRefs>                  </configuration>              </plugin>          </plugins>          <!-- Build with Java 1.5 -->          <pluginManagement>              <plugins>                  <plugin>                      <groupId>org.apache.maven.plugins</groupId>                      <artifactId>maven-compiler-plugin</artifactId>                      <configuration>                          <source>1.5</source>                          <target>1.5</target>                      </configuration>                  </plugin>              </plugins>          </pluginManagement>      </build>    </project> |

The only items of interest in the POM are:

* We depend on the CXF v2.1.2 client libraries and the (most excellent) SLF4J logging system
* We invoke the **cxf-codegen-plugin** to run *wsdl2java* to generate our Java stub code into${basedir}/target/generated/src/main/java
* We use the **build-helper-maven-plugin** so that Maven can compile from two source directories (normally Maven just compiles what’s in ${basedir}/src/main and not${basedir}/target/generated/src, so we tell Maven to go compile the generated stub code too)
* We use the **maven-assembly-plugin** to create a final JAR containing all necessary dependencies, which Maven will create as weather-client-jar-with-dependencies.jar when we perform a mvn assembly:assembly

At this point, even though I had no code in the project, I ran mvn assembly:assembly to build the Java stubs from the WSDL file. The output is in the generated source directory mentioned earlier, in case you want to go poke at it.

The Client Code

Once we have the autogenerated stubs we can use them in a real Java program. Before you can use the stubs, you have to identify what the actual *service object* is. You can find out by looking at the generated stub code and see which Java class extends Service. That will be the service interface that you call in your client. In this case, the service is called simply “Weather”.

Without further ado, here’s the code I wrote to invoke the SOAP service:

weather-client/src/main/java/com/logicsector/soapclient/**SoapClient.java**

|  |  |
| --- | --- |
| 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  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | package com.logicsector.soapclient;    import java.text.SimpleDateFormat;  import java.util.Date;  import java.util.List;    import org.slf4j.Logger;  import org.slf4j.LoggerFactory;    import com.cdyne.ws.weatherws.Forecast;  import com.cdyne.ws.weatherws.ForecastReturn;  import com.cdyne.ws.weatherws.POP;  import com.cdyne.ws.weatherws.Temp;  import com.cdyne.ws.weatherws.Weather;  import com.cdyne.ws.weatherws.WeatherSoap;    public class SoapClient {      private static final Logger           LOGGER      = LoggerFactory.getLogger(SoapClient.class);      private static final SimpleDateFormat DATE\_FORMAT = new SimpleDateFormat("EEEE, MMMM d yyyy");        public static void main(String[] args) {          try {              LOGGER.debug("Creating weather service instance (Note: Weather = Service subclass)...");              long start = new Date().getTime();              // Get a reference to the SOAP service interface.              Weather weatherService = new Weather();              WeatherSoap weatherSoap = weatherService.getWeatherSoap();              // An alternate way to get the SOAP service interface; includes logging interceptors.              // JaxWsProxyFactoryBean factory = new org.apache.cxf.jaxws.JaxWsProxyFactoryBean();              // factory.setServiceClass(WeatherSoap.class);              // factory.setAddress("<http://ws.cdyne.com/WeatherWS/Weather.asmx>");              // factory.getInInterceptors().add(new org.apache.cxf.interceptor.LoggingInInterceptor());              // factory.getOutInterceptors().add(new org.apache.cxf.interceptor.LoggingOutInterceptor());              // WeatherSoap weatherSoap = (WeatherSoap) factory.create();              long end = new Date().getTime();              LOGGER.debug("...Done! weatherService instance: {}", weatherService);              LOGGER.debug("Time required to initialize weather service interface: {} seconds", (end - start) / 1000f);                // Send a SOAP weather request for zip code 94025 (Menlo Park, CA, USA).              LOGGER.debug("weatherSoap instance: {}", weatherSoap);              start = new Date().getTime();              ForecastReturn forecastReturn = weatherSoap.getCityForecastByZIP("94025");              end = new Date().getTime();              LOGGER.debug("Time required to invoke 'getCityForecastByZIP': {} seconds", (end - start) / 1000f);              LOGGER.debug("forecastReturn: {}", forecastReturn);              LOGGER.debug("forecastReturn city: {}", forecastReturn.getCity());              LOGGER.debug("forecastReturn state: {}", forecastReturn.getState());              LOGGER.debug("forecastReturn result: {}", forecastReturn.getForecastResult());              LOGGER.debug("forecastReturn response text: {}", forecastReturn.getResponseText());              LOGGER.debug("");              List<Forecast> forecasts = forecastReturn.getForecastResult().getForecast();              for (Forecast forecast : forecasts) {                  LOGGER.debug("  forecast date: {}", DATE\_FORMAT.format(forecast.getDate().toGregorianCalendar().getTime()));                  LOGGER.debug("  forecast description: {}", forecast.getDesciption());                  Temp temps = forecast.getTemperatures();                  LOGGER.debug("  forecast temperature high: {}", temps.getDaytimeHigh());                  LOGGER.debug("  forecast temperature low: {}", temps.getMorningLow());                  POP pop = forecast.getProbabilityOfPrecipiation();                  LOGGER.debug("  forecast precipitation day: {}%", pop.getDaytime());                  LOGGER.debug("  forecast precipitation night: {}%", pop.getNighttime());                  LOGGER.debug("");              }              LOGGER.debug("Program complete, exiting");          }          catch (Exception e) {              LOGGER.error("An exception occurred, exiting", e);          }      }    } |

Note that we’re importing the stubs as **import com.cdyne.ws.weatherws.Forecast**, etc, within the client program. The client is also hard-coded to get the weather report from the 94025 zip code, although you could easily alter the client to take the zip code as a command-line argument.

The All-Important CXF Client Configuration File

This is the part of the development process that threw me for a loop. I didn’t see any CXF documentation that indicated a cxf.xml file needs to be in the classpath of the client, so I hadn’t included one in the project. My client program kept failing with a (very cryptic, very unhelpful) CXF*BusException* (the complete message was **org.apache.cxf.BusException: No binding factory for namespace http://schemas.xmlsoap.org/soap/ registered**, which I’m mentioning here in case anyone else is Googling with the same problem).

Sure, there are plenty of CXF tutorials on the Internet, but they mostly seem to build a client and a service in the same project (sharing a cxf.xml file) and I had assumed the configuration file was for configuring only the *server*. Silly me.

It took me several *days* of Googling, trying different JAR dependencies, Googling again, testing various source code modifications, Googling some more, asking for help on the cxf-user mail list — all to no avail.

Eventually, while reading the solution to an unrelated problem, I finally discovered the cause. On start-up for a server OR A CLIENT, the CXF system looks for a cxf.xml file, and fails without it. Just for the record, the BusException message is incredibly unhelpful. Grrr!! I think it should read something like **org.apache.cxf.BusException: No binding factory for namespace http://schemas.xmlsoap.org/soap/ registered *(did you include a cxf.xml file somewhere in the classpath?)***, or some such.

Anyhoo, here’s the CXF configuration file I used. Not much to it. It’s basically just a trivial Spring configuration with three lines of imports.

weather-client/src/main/resources/**cxf.xml**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <beans xmlns="<http://www.springframework.org/schema/beans>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"      xmlns:http-conf="<http://cxf.apache.org/transports/http/configuration>"      xsi:schemaLocation="<http://cxf.apache.org/transports/http/configuration>    <http://cxf.apache.org/schemas/configuration/http-conf.xsd>    <http://www.springframework.org/schema/beans>    <http://www.springframework.org/schema/beans/spring-beans.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"/>    </beans> |

Good thing this is easier than Axis.

Logging Configuration

For completeness, I’ve included the logging file I used. Since we’re using LOG4J as the logging layer under SLF4J, we need to supply a log4j.properties file.

weather-client/src/main/resources/**log4j.properties**

|  |  |
| --- | --- |
| 1  2  3  4  5 | log4j.rootCategory=WARN, CONSOLE  log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender  log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout  log4j.appender.CONSOLE.layout.ConversionPattern=[%d{ABSOLUTE} %-5p %c{1}]: %m%n  log4j.logger.com.logicsector=DEBUG |

Building and Running the Client

Once you have all the code in order, it’s time to build it. From within the weather-client directory, just use the previously-mentioned Maven build command to create a JAR file with dependencies:

mvn assembly:assembly

Now we can run the client. As mentioned previously, the client is hard-coded to get the weather report for the 94025 zip code (Menlo Park, California). From within the weather-client directory, the following command will start the client and invoke the service:

java -cp target/weather-client-jar-with-dependencies.jar com.logicsector.soapclient.SoapClient

The command above should all be on a single line. Note you may need to change your path separator if you’re not on a Unix-based system. If everything went smoothly you should see output something like this:

|  |  |
| --- | --- |
| 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  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | [17:31:06,278 DEBUG SoapClient]: Creating weather service instance (Note: Weather = Service subclass)...  Oct 18, 2008 5:31:08 PM org.apache.cxf.service.factory.ReflectionServiceFactoryBean buildServiceFromWSDL  INFO: Creating Service {http://ws.cdyne.com/WeatherWS/}Weather from WSDL: file:/c:/Projects/weather-client/src/main/wsdl/weat  her.wsdl  [17:31:08,325 DEBUG SoapClient]: ...Done! weatherService instance: com.cdyne.ws.weatherws.Weather@754fc  [17:31:08,325 DEBUG SoapClient]: Time required to initialize weather service interface: 2.047 seconds  [17:31:08,325 DEBUG SoapClient]: weatherSoap instance: org.apache.cxf.jaxws.JaxWsClientProxy@6458a6  [17:31:08,825 DEBUG SoapClient]: Time required to invoke 'getCityForecastByZIP': 0.5 seconds  [17:31:08,825 DEBUG SoapClient]: forecastReturn: com.cdyne.ws.weatherws.ForecastReturn@aea710  [17:31:08,825 DEBUG SoapClient]: forecastReturn city: Menlo Park  [17:31:08,825 DEBUG SoapClient]: forecastReturn state: CA  [17:31:08,825 DEBUG SoapClient]: forecastReturn result: com.cdyne.ws.weatherws.ArrayOfForecast@5a2eaa  [17:31:08,825 DEBUG SoapClient]: forecastReturn response text: City Found  [17:31:08,825 DEBUG SoapClient]:  [17:31:08,825 DEBUG SoapClient]:   forecast date: Friday, October 17 2008  [17:31:08,825 DEBUG SoapClient]:   forecast description: Sunny  [17:31:08,825 DEBUG SoapClient]:   forecast temperature high: 82  [17:31:08,825 DEBUG SoapClient]:   forecast temperature low: 58  [17:31:08,825 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,825 DEBUG SoapClient]:   forecast precipitation night: 00%  [17:31:08,825 DEBUG SoapClient]:  [17:31:08,825 DEBUG SoapClient]:   forecast date: Saturday, October 18 2008  [17:31:08,825 DEBUG SoapClient]:   forecast description: Sunny  [17:31:08,841 DEBUG SoapClient]:   forecast temperature high: 73  [17:31:08,841 DEBUG SoapClient]:   forecast temperature low: 55  [17:31:08,841 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,841 DEBUG SoapClient]:   forecast precipitation night: 00%  [17:31:08,841 DEBUG SoapClient]:  [17:31:08,841 DEBUG SoapClient]:   forecast date: Sunday, October 19 2008  [17:31:08,841 DEBUG SoapClient]:   forecast description: Partly Cloudy  [17:31:08,841 DEBUG SoapClient]:   forecast temperature high: 70  [17:31:08,841 DEBUG SoapClient]:   forecast temperature low: 55  [17:31:08,841 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,841 DEBUG SoapClient]:   forecast precipitation night: 00%  [17:31:08,841 DEBUG SoapClient]:  [17:31:08,841 DEBUG SoapClient]:   forecast date: Monday, October 20 2008  [17:31:08,841 DEBUG SoapClient]:   forecast description: Partly Cloudy  [17:31:08,841 DEBUG SoapClient]:   forecast temperature high: 70  [17:31:08,841 DEBUG SoapClient]:   forecast temperature low: 53  [17:31:08,841 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,841 DEBUG SoapClient]:   forecast precipitation night: 00%  [17:31:08,841 DEBUG SoapClient]:  [17:31:08,841 DEBUG SoapClient]:   forecast date: Tuesday, October 21 2008  [17:31:08,856 DEBUG SoapClient]:   forecast description: Sunny  [17:31:08,856 DEBUG SoapClient]:   forecast temperature high: 73  [17:31:08,856 DEBUG SoapClient]:   forecast temperature low: 54  [17:31:08,856 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,856 DEBUG SoapClient]:   forecast precipitation night: 10%  [17:31:08,856 DEBUG SoapClient]:  [17:31:08,856 DEBUG SoapClient]:   forecast date: Wednesday, October 22 2008  [17:31:08,856 DEBUG SoapClient]:   forecast description: Sunny  [17:31:08,856 DEBUG SoapClient]:   forecast temperature high: 74  [17:31:08,856 DEBUG SoapClient]:   forecast temperature low: 55  [17:31:08,856 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,856 DEBUG SoapClient]:   forecast precipitation night: 00%  [17:31:08,856 DEBUG SoapClient]:  [17:31:08,856 DEBUG SoapClient]:   forecast date: Thursday, October 23 2008  [17:31:08,856 DEBUG SoapClient]:   forecast description: Sunny  [17:31:08,856 DEBUG SoapClient]:   forecast temperature high: 73  [17:31:08,856 DEBUG SoapClient]:   forecast temperature low: 55  [17:31:08,856 DEBUG SoapClient]:   forecast precipitation day: 00%  [17:31:08,856 DEBUG SoapClient]:   forecast precipitation night: 00%  [17:31:08,856 DEBUG SoapClient]:  [17:31:08,856 DEBUG SoapClient]: Program complete, exiting |

Interestingly, it takes 2 seconds on my machine to initialize the interface, which seems like a really long time. CXF is probably doing a lot of stuff under the covers, but still, 2 seconds is *forever* in computer time.

The call to the weather service interface, once initialized, takes about half a second every time, which includes marshalling a SOAP request, sending it over the internet, receiving the response, and unmarshalling its contents. Not too bad I guess.

Concluding Thoughts

Hopefully this example will form the basis of your next SOAP client. It really is pretty easy once you see a complete and working example.

If you were ambitious, parts of this code could easily be incorporated into a web application that provides a weather report for the user. You’d simply create a servlet that takes the zip code as a parameter, invokes the SOAP service, and shows the weather report in the response. In other words, the technique of calling the SOAP service would be the same even if this was a web application.

Well, that’s the end of my post about creating a CXF client with Maven. I’d love to read your comments if you found this post helpful.

*EDIT:* Download the Source!

All the code necessary to build the project is listed above. However, to save time, you can simply[download the code](http://www.logicsector.com/wp-content/uploads/2011/12/weather-client.zip) from my web site. Happy coding!