Table of Contents

Java and SOAP By Robert Englander ISBN: 0-596-00175-4 Publisher: O'Reilly
Print Publication Date: 2002/05/01

Chapter 11. JAX-RPC and JAXM

With most software technologies, standardization occurs when the technology has gained a certain amount of momentum. Well, SOAP and web services have gained the requisite momentum, and standards (and their resulting acronyms) are sprouting up. There are a slew of API standards evolving for manipulating XML in Java, all of which fall under the umbrella of the Java APIs for XML (JAX). JAX is not a product — it's an API definition, and as such, it's no different from many of the countless API specifications associated with Java. However, since XML is such a wide-ranging area, JAX is made up of many components. Two of these are of particular interest to us: JAX-RPC and JAXM. JAX-RPC is the Java API for XML-based RPC. This is the API that, over time, I'd expect most significant SOAP RPC implementations to follow. JAXM is the Java API for XML Messaging, and, like JAX-RPC, it will likely become widely accepted.

These two APIs are new, and are not yet ready for the production world. The specifications for JAXM and JAX-RPC are in public review, and therefore are expected to change before they're complete; however, at this stage, the changes should be relatively minor. Nonetheless, Sun has produced a reference implementation that allows you to start working with these technologies. This reference implementation is included in the Java Web Services Developer Pack (Java WSDP) available at http://java.sun.com/webservices/webservicespack.html. You can treat this release as an early beta; as long as you don't use it for a production server and don't expect it to have all the bugs worked out or the features you like, it will help you get up to speed so that you can build a production server as soon as possible when the standards are finalized. The Web Services Developer Pack uses a Tomcat 4 server, so if you've been using Apache SOAP with Tomcat up to this point, you should be pretty comfortable working with JAX. Even if you haven't been using Tomcat, the installation instructions for the pack are straightforward, and you shouldn't have much trouble with it.

11.1. JAX-RPC

Let's start with JAX-RPC, since we've spent most of our time in the RPC world. Like GLUE, JAX-RPC hides the details of the underlying protocol (SOAP) from the programmer. To hide the underlying protocol, JAX-RPC makes use of objects known as stubs and ties. A *stub* is used by a client application to access a remote service. The stub looks like the service interface, but it runs as part of the local client process. The stub in turn uses the underlying framework to generate an appropriate SOAP message and send it over HTTP. So from the

Java and SOAP By Robert Englander ISBN: 0-596-00175-4 Publisher: O'Reilly
Print Publication Date: 2002/05/01

perspective of a client application programmer, if you have the stub you don't need to deal with any of the details of the SOAP message. Figure 11-1 shows the relationship between client code, stubs, and the runtime component of the JAX-RPC client framework.

Figure 11-1. Stubs in the JAX-RPC architecture

11.2. Working Without Ant

The Java Web Services Developer Pack tutorial relies heavily on a package called Ant. All Java source file compilation, directory creation, JAR file creation, etc., are managed by Ant configuration files. Although Ant is quite popular, and using it to build the kind of code in this chapter is a good idea, we're not going to cover its use here. (For more information about Ant, see *Ant: The Definitive Guide* by Jesse Tilly and Eric Burke.) Instead, I'm going to take you through the manual steps for the examples. If you know how to use Ant, you'll be able to gloss over some of the details in this chapter.

The most critical thing to set up is the classpath. The Early Access Release makes use of 10 JAR files that need to be on your classpath. These files are located in the *common\lib* directory underneath the root of your Developer Pack installation, and it's up to you to make sure they are on your classpath using whatever technique you prefer. Here are the files:

11.3. Creating a JAX-RPC Service

Back in Chapter 9 we talked about building proxy services, which sit between client applications and other services. In that chapter, the back-end service with which the proxies communicated was located at *http://mindstrm.com:8004/glue/urn:CorpDataServices*. Let's do the same thing again, only this time we'll build our proxy service using the JAX-RPC reference implementation.[1] This example allows us to see how JAX-RPC interoperates with GLUE, and gives us a chance to build both a service and a client using JAX-RPC.

[1] I'm assuming you've downloaded and installed the Java Web Services Developer Pack already, and verified that it is working properly.

The first step in building the service is to write a Java interface for it, which I'll call `javasoap.book.ch11.services.IStockServiceProxy`. This interface must implement the `java.rmi.Remote` interface because JAX-RPC makes use of the Java RMI package. This strategy is fairly common because Java RMI provides a good distributed computing abstraction. The methods in the interface should be defined to throw

Java and SOAP By Robert Englander ISBN: 0-596-00175-4 Publisher: O'Reilly
Print Publication Date: 2002/05/01

`RemoteException`, which also comes from the `java.rmi` package. We'll define one method, `getStockQuote( )`, which takes a string parameter for the stock symbol and returns an instance of `javasoap.book.ch9.services.ProxyQuote`. There's no need to create a new class for returning a quote; we've already done it in Chapter 9. Here's what the code looks like for the `IStockServiceProxy` interface:

11.4. Creating a JAX-RPC Client

Before we start writing a client application, let make sure to copy all the class files generated earlier into their proper locations on the local filesystem. Remember that we have them in a WAR file, and we also have them under the *gen\WEB-INF\classes* directory. But neither of these locations is on my local classpath, so now is a good time to put them where they belong so the client application can access them.

Writing a client application for this service is pretty simple. All you need to do is get an instance of the stub that was generated earlier and call the desired service method on that object. Here's the code for a sample application class called `javasoap.book.ch11.client.StockQuoteApp`:

11.5. Generating Stubs from WSDL

Generating client code from WSDL is pretty simple. First we'll modify the *CorpDataServices.xml* file that we used earlier as input to `xrpcc`. We have to remove the entire `rmi` section and replace it with a `wsdl` section. Here's what the modified file should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <wsdl name="CorpDataServicesProxy"
        location="http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl"
        packageName="javasoap.book.ch11.glueclient">
  </wsdl>
</configuration>
```

11.6. Dynamic Invocation Interface

At times, you want more control over the way your application interfaces with a service. One reason for wanting more control is that your application does not know the service methods and parameters in advance, making it impossible to generate stubs. It's also reasonably likely that the service you want to invoke does not provide a WSDL document describing it. The Dynamic Invocation Interface (DII) is an API that allows you to invoke services at a level much closer to the JAX-RPC runtime. We're not going to spend any time with DII, but you should be aware of it. We saw a glimpse of its structure in Chapter 9 when we developed a simple Axis example. It may not be identical, and these APIs are still evolving, but the approach is the same.

Java and SOAP By Robert Englander ISBN: 0-596-00175-4 Publisher: O'Reilly
Print Publication Date: 2002/05/01

I see DII as a last-resort approach to writing code to communicate with SOAP web services. Letting a tool generate the low-level code based on a WSDL description is really the best choice, since WSDL is language and implementation independent, and is an accepted standard. If your SOAP tools can work with a WSDL file, that's your best bet.

11.7. JAXM, in Less Than a Nutshell

The APIs that JAXM provides for writing messaging clients and services are at a low level, similar to what we saw in Chapter 8. They require you to make a connection to the server (or an intermediary) and build up the XML that constitutes the message. Although this is a powerful technique, it gives you a lot of rope with which to hang yourself. I like the flexible nature of working directly with XML, but I still want the structure of RPC-style interaction. If you need to design your services to accept XML documents as parameters, you might find that literal encoding is sufficient within the context of RPC. If that's not enough, then you'll have to walk the messaging (or proprietary) path.

I suggest you take a look at the examples that come with the Java Web Services Developer Pack tutorial. Even though we don't develop a JAXM example here, you're armed with enough knowledge to understand what's happening. And something tells me that a future edition of this book will delve deep into JAXM, JAX-RPC, and the rest of the JAX Pack.

11.8. What Next?

You're now more than ready to start developing web services in Java. Don't be put off by the pace of change. Keep track of the latest technologies, including new releases of existing implementations. New versions of GLUE, Axis (Apache SOAP), .NET, and many others will make the development of services in Java easier, better, and more powerful. Finally, keep an eye on standardized APIs, security, service registries, and IDE integration; the list is growing as fast as the bandwidth requirements! Jump in.

Java and SOAP By Robert Englander ISBN: 0-596-00175-4 Publisher: O'Reilly
Print Publication Date: 2002/05/01