CHAPTER 15

Working with web services

15.1 What are web services and what is SOAP? 356
15.2 Creating a SOAP client application with Ant 357
15.3 Creating a SOAP service with Axis and Ant 363
15.4 Adding web services to an existing web application 367
15.5 Writing a client for our SOAP service 371
15.6 What is interoperability, and why is it a problem? 376
15.7 Building a C# client 376
15.8 The rigorous way to build a web service 381
15.9 Reviewing web service development 382
15.10 Calling Ant via SOAP 383
15.11 Summary 384

Web services are an emerging target of software development. Put simply, a web service is a web or enterprise application that provides a way for other programs to call it by using XML as the means of exchanging data. If you can build and deploy a web application, you can build and deploy a web service.

If it's all so easy, why do we have a whole chapter on web services? Because they add new problems to the process: integration and interoperability. Client applications need to be able to call your web service, including applications that are written in different languages or that use different web service toolkits. We need to extend our existing development process to integrate client-side and interoperability tests.

In this chapter, we extend the web application we wrote in chapter 12, adding a SOAP interface to it. We use the Apache Axis library to provide our SOAP interface, rather than the Sun version, because it comes from a sister project to Ant and because we like it. After adding SOAP to our application, we build tests for it, first with a Java client, and then with a C# client running on the .NET platform. As we said, integration and interoperability are the new challenges of a web service development process.

We do not delve deeply into the details of SOAP and web services; we encourage you to read books on the subject (for example, Wesley 2002, and Graham 2001), as well as the SOAP specifications hosted in the web services working group at the W3C (http://www.w3.org/2002/ws/). We do explain the basic concepts behind web services, however, and show how you can use Ant to build, test, and call a web service.

15.1 What are web services and what is SOAP?

Web services use XML as the language of communication to provide computing functionality as a service over the Internet. Web services extend the web, so every service exposes itself as one or more URLs, URLs that provide functionality in response to POST or GET requests. The exact details of the communication are still evolving; SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are the current competing ideologies. REST is a conceptual model of how to expose objects, properties, and methods as URLs (Fielding 2000); to implement a REST service you export URLs for all the objects and attributes you wish callers to have access to; callers send and receive data to and from the URLs in the format they prefer. SOAP has more of a Remote Procedure Call (RPC) flavor. A single URL acts as an *endpoint*; a SOAP endpoint can receive different requests/method calls in the posted request, and return different XML responses for the different methods.

Central to SOAP is WSDL, the Web Services Description Language. This is roughly the SOAP equivalent of an Interface Definition Language (IDL). SOAP 1.1 clients use it to examine a remote API, and that services use to define which API they export, as shown in figure 15.1. Unlike the old RPC world, where writing an IDL file was mandatory, in the new SOAP universe, you can write your classes and let the run time generate the WSDL from it. Many web service practitioners consider this to be a bad thing, as it makes interoperability with other SOAP implementations, and maintenance in general, that much harder. It is, therefore, a shame that WSDL is even harder to work with than classic IDL.

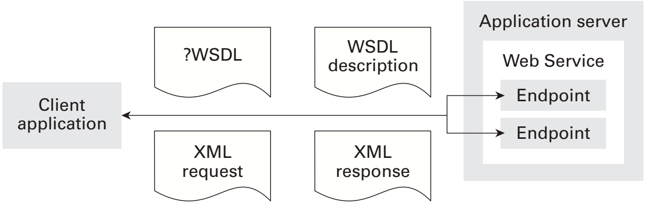

Figure 15.1 SOAP-based web services offer service metadata in their WSDL file and method invocation with XML requests and responses.

SOAP 1.2, still under development, looks to be moving away from an RPC model, in which the caller blocks till the response is received, to a more asynchronous model in which SOAP messages are routed to a destination. Some of the low-level APIs used to support XML messaging in Java can work with this model, specifically the JAXM API for XML messaging.

15.1.1 The SOAP API

The Java API for XML-based RPC (JAX-RPC) is a higher level SOAP API. Both the Apache and Sun toolkits implement JAX-RPC. This API comprises the `javax.xml.soap` and `javax.xml.rpc` packages; the different libraries provide their own implementations of these APIs. Apache Axis has its own API under `org.apache.axis`. Although these APIs are powerful, they are also complex. Although we will use these APIs and their implementation, we will avoid learning the APIs ourselves by handing off the grunge work to Axis and associated tools.

15.1.2 Adding web services to Java

SOAP support in Java is still evolving; initially Sun neglected it—with Java everywhere, there was no need for web services. However, Sun has pulled enough of a U-turn to embrace SOAP, with full support promised in J2EE 1.4. They also provide the Java web services Developer Pack as a download from http://java.sun.com/webservices/. This large download includes many well-known Apache components: Tomcat, Xerces, Xalan, and even Ant. Other vendors provide their own toolkits for supporting web services in Java.

We stick with the Apache Axis library, from the sibling project of Jakarta, because it is nice to be able to step into the code to debug everything. We also know that if there is anything we don't like about the implementation we can get it fixed, even if that means doing it ourselves. You can download Apache Axis from its home page, http://xml.apache.org/axis/; we have used the beta-1, beta-2, and later CVS versions. Like all open source projects, it is continually evolving, so some details may have changed since we wrote this.

Both the Sun and Apache implementations have a moderately complex deployment process: the Sun server library only works with a specially modified version of Tomcat, whereas the Apache implementation prefers that you add your services to their example web application, rather than write your own. We will be patching our existing web service to support Axis, which involves some effort and some more testing.

15.2 *CREATING A SOAP CLIENT APPLICATION WITH ANT*

Before we build our own service, we will pick an existing web service and build a client for it. This lets us explore the client-side experience and build process, before we go deeper into web service development. All that we learn here will apply to our own integration tests.

The Apache Axis library contains two programs for use at build time: WSDL2Java and Java2WSDL. These programs create Java classes from a WSDL description and vice versa. There are also two tasks—`<java2wsdl>` and `<wsdl2java>`—which are Ant wrappers around the programs. Unfortunately, in the beta-2 release, these tasks are not part of the binary distribution; they live in the test package, and you must build them yourself. Because these tasks are not part of the official distribution, and because

they are undocumented, we are not going to cover them. Instead, we will call the documented programs with the `<java>` task. Figure 15.2 shows the overall workflow of our Ant build file.

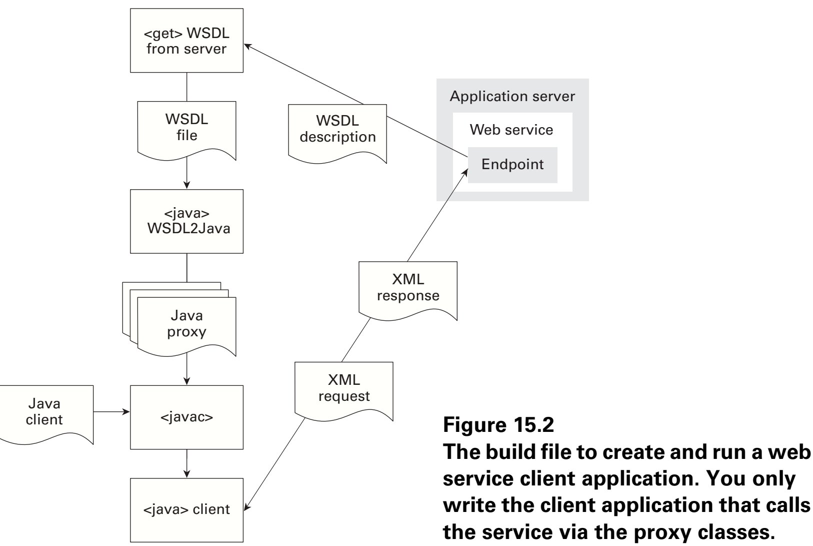

Figure 15.2
The build file to create and run a web service client application. You only write the client application that calls the service via the proxy classes.

15.2.1 Preparing our build file

The first step in our build process is to name the endpoint, the URL of the service that we will call. We will use one of the services offered by xmethods (http://xmethods.net), a service that provides a stock quote. First, we set a property to point to the WSDL file of the service:

```
<property name="endpoint"
 value=
 "http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl"
 />
```

We then define the directory locations to store the cached file, and to store any generated Java. As usual, we do not want to place generated source into our source tree, because of the risk of accidental overwriting. We also need an `init` target to set up the build.

```
  <property name="axis.dir"
    location="${lib.dir}/xml-axis" />
  <property name="xercesxalan.dir"
    location="${lib.dir}/xercesxalan" />

  <property name="build.dir" location="build"/>
  <property name="fetched.dir" location="${build.dir}/fetched"/>
  <property name="generated.dir" location="${build.dir}/generated"/>
```

```
  <target name="init">
   <mkdir dir="${fetched.dir}"/>
   <mkdir dir="${generated.dir}"/>
   <mkdir dir="${cached.dir}"/>
   <mkdir dir="${build.classes.dir}"/>
   <condition property="offline">
    <not>
    <http url="${endpoint}"/>
    </not>
   </condition>
</target>
```

Our `init` target probes for the endpoint being reachable, and sets a property if we are offline. When offline, we will not be able to run the service, but we still want to be able to compile the code against a cached copy of the WSDL file.

Retrieving the WSDL from the remote server

The remote server describes the SOAP API from a WSDL file, which it serves alongside the SOAP endpoint. We fetch this WSDL file by using the `<get>` task:

```
<target name="fetch-wsdl" depends="init" unless="offline">
  <get src="${endpoint}" dest="${fetched.dir}/api.wsdl"/>
</target>
```

To run this target behind a firewall, you may need to set the Java proxy properties to get to the remote endpoint; the `<setproxy>` task lets you do this inside your build file:

```
<setproxy proxyhost="web-proxy" proxyport="8080" />.
```

15.2.2 Creating the proxy classes

After retrieving the file, we can create Java proxy stubs from the WSDL-described API. These classes allow us to talk to a SOAP service from our own code, without having to refer to `javax.xml.soap` or `org.apache.axis` in our code. It also adds compile time-type safety into our use of the SOAP service—the generated classes talk to the SOAP API, so we don't have to. To create the stubs, we first set up the Axis classpath, then write a target to create the Java classes from the WSDL we have just fetched.

```
<path id="axis.classpath">
  <fileset dir="${axis.dist.dir}">
    <include name="**/*.jar"/>
  </fileset>
  <fileset dir="${xercesxalan.dist.dir}">
    <include name="*.jar"/>
  </fileset>
</path>

<target name="import-wsdl" depends="fetch-wsdl">
  <java
    classname="org.apache.axis.wsdl.WSDL2Java"
```

```
    fork="true"
    failonerror="true"
    classpathref="axis.classpath"
    >
    <arg file="${fetched.dir}/api.wsdl"/>
    <arg value="--output"/>
    <arg file="${generated.dir}"/>
    <arg value="--verbose"/>
    <arg value="--package"/>
    <arg value="soapapi"/>
  </java>
</target>
```

This target runs the org.apache.axis.wsdl.WSDL2Java program to convert the WSDL interface description into Java source. We do this with `<java>`, taking care to set up the classpath to include all the files in the Axis lib directory. We also had to add an XML parser to the classpath, so we include Xerces in the path. The alternative approach is to add the Ant classpath to the `<java>` call, but we prefer to keep things self-contained. We also run the class in a new JVM, so that if the program returns an error by calling `System.exit()`, we get an exit code instead of the sudden death of Ant.

The parameters to the task tell WSDL2Java to create classes from the downloaded WSDL file, into the package `soapapi`, into the directory build/generated. The result of the build, with many long lines wrapped to make them readable is:

```
import-wsdl:
     [java] Parsing XML file:
         build/fetched/api.wsdl
     [java] Generating portType interface:
         build/generated/soapapi/StockQuoteService.java
     [java] Generating client-side stub:
         build/generated/soapapi/StockQuoteServiceSoapBindingStub.java
     [java] Generating service class:
         build/generated/soapapi/StockQuoteServiceService.java
     [java] Generating service class:
         build/generated/soapapi/StockQuoteServiceServiceLocator.java
     [java] Generating fault class:
         build/generated/soapapi/Exception.java

BUILD SUCCESSFUL
```

The program generated Java proxy classes for the endpoint. Let’s look at them to see how we can use them.

What WSDL2Java creates

The target creates five Java classes that implement the client-side proxy to this service, and one interface listing the methods offered by the remote service:

```
/**
 * StockQuoteService.java
 *
```

```
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package soapapi;

public interface StockQuoteService extends java.rmi.Remote {
    public float getQuote(java.lang.String symbol) throws
                     java.rmi.RemoteException, soapapi.Exception;
}
```

The program also creates a proxy class that implements this interface and redirects it to a remote endpoint, and a locator class that finds the endpoint at run time and binds to it.

15.2.3 Using the SOAP proxy classes

To use these generated classes, simply create a Java file that imports and invokes them:

```
import soapapi.*;

public class SoapClient {

   public static void main(String args[]) throws java.lang.Exception {
       StockQuoteServiceServiceLocator locator;
       locator=new StockQuoteServiceServiceLocator();
       StockQuoteService service;
       service= locator.getStockQuoteService();
       for(int i=0;i<args.length;i++) {
           float quotation=service.getQuote(args[i]);
           System.out.println(args[i]+"="+quotation);
       }
   }
}
```

This service first creates a locator instance to locate the endpoint. In this example, it always returns the same endpoint, but it is conceivable that a locator could use a Universal Description, Discovery, and Integration (UDDI) registry or other directory service to locate a service implementation dynamically.

After creating the locator we bind to the service by asking the locator for a binding; it returns an implementation of the remote interface—the stub class that WSDL2Java created. With this binding, we can make remote calls, here asking for the stock price of every argument supplied to the `main` method, that being the classic simple web service.

15.2.4 Compiling the SOAP client

Before we can run that method, we have to compile the source:

```
<target name="compile" depends="import-wsdl">
  <javac
   srcdir="src;${generated.dir}"
   destdir="${build.classes.dir}"
```

```
    classpathref="axis.classpath"
    debuglevel="lines,vars,source"
    debug="true"
    includeAntRuntime="false"
    />
</target>
```

We supply a path to the source directory to include both our source and the generated files. One irritation of the current SOAP import process is that the Java files are always regenerated, which means they always need recompilation. This makes the build longer than it need be. Unless WSDL2Java adds dependency checking, you should use something similar to `<uptodate>` to bypass the `import-wsdl` target when it is not needed. There is an extra complication here; you need to use a `<filesmatch>` test inside a `<condition>` to verify that the file you just fetched with `<get>` hasn't changed. We omit all this because it is so complex.

15.2.5 Running the SOAP service

With compilation complete, there is one more target to write:

```
<target name="run" depends="compile" unless="offline">
  <java
    classname="SoapClient"
    fork="true"
    failonerror="true"
    >
    <arg value="SUNW"/>
    <arg value="MSFT"/>
    <classpath>
      <path refid="axis.classpath"/>
      <pathelement location="${build.classes.dir}"/>
    </classpath>
  </java>
</target>
```

This target runs the stock quote client, fetching the stock price of Sun and Microsoft, producing a result that we cannot interpret as good news for either of them, at least during May 2002:

```
run:
     [java] SUNW=6.67
     [java] MSFT=52.12
```

If we were ambitious, we could save the output of the run to a properties file, and then load the output as Ant properties and somehow act on them.[1]

[1] Having applications output their results in the properties file format makes it very easy to integrate their results into Ant or, indeed, into any other Java application. We have used this trick for Win32/Java communications. XML is more powerful, but harder to work with.

15.2.6 Reviewing SOAP client creation

As we have shown, Ant can create a build file that goes from a remote WSDL description to Java code and then it can compile and run this code to make remote SOAP RPC calls.

More SOAP services could provide extra functionality for the build, such as returning information from a remote database, information that could populate a file used in the build or one of the build's redistributables.

Alternative implementations to the Apache Axis SOAP libraries have different processes for creating stub code from WSDL services. We have not looked at the process for using alternate implementations in any detail, but they should be amenable to a similar build process. If multiple Java SOAP implementations do become popular, we may eventually see Ant adding a task to import WSDL that supports different implementations, just as `<javac>` supports many Java compilers.

15.3 CREATING A SOAP SERVICE WITH AXIS AND ANT

Apache Axis enables you to develop SOAP services in three ways: the simple way, the rigorous way, and the hard way. The simple method is to save your Java files with the .jws extension and then save them under the Axis web application; when you fetch these files in a web browser Axis compiles them and exports them as web services. The rigorous method is to write the WSDL for services, create the bindings and web service deployment descriptors, copy these to the Axis servlet, and register these deployment descriptors with the servlet.

The hard way is to retrofit an existing application with SOAP support and then use either of the previous two approaches. It is a lot easier to develop under Axis than it is to add SOAP to an existing web application.

Installing Axis on a web server

Axis is a web application. It is redistributed in the expanded form, rather than as a WAR file. To install it, you copy everything under webapp/axis to the directory webapp/axis in Tomcat. Because many versions of Tomcat 4 do not allow libraries in WEB-APP/lib to implement `java.*` or `javax.*` packages, you also need to copy jaxrpc.jar and saa.jar to CATALINA_HOME/common/lib. Make sure you have the right lib directory—CATALINA_HOME/server/lib and CATALINA_ HOME/lib are the wrong places. If you are trying to install Axis on other application servers and you are using Java1.4, then you may need to configure the server so that the system property `java.endorsed.dirs` includes the directory containing the jaxrpc.jar file.

If these URLs load, you know that you have a successful installation, assuming that your copy of Tomcat is running on port 8080:

```
http://localhost:8080/axis/servlet/AxisServlet
http://localhost:8080/axis/StockQuoteService.jws?wsdl
```

The first of these verifies that all the libraries are in place and all is well; the second forces the Axis servlet to compile the sample web service and run it. The service is, of course, a version of the infamous stock option service. If either URL is unreachable, you may not be running Tomcat on that port; alter the URL to point to your server. If you receive the 500 error code (internal server error), it is probably because the libraries are not in the right place. Failing that, check the installation guide in the Axis documents and FAQ at the Axis web site for more advice.

15.3.1 The simple way to build a web service

Now that Axis is in place and working, let us write our simple web service, which will export the indexed search as a web service, enabling calling applications to send a search term and then get the result back as a list of URLs. We will also tack in a management call to tell us the last search string submitted.

The easiest way to write a web service in Axis is to implement the service API as a Java class, save it with the extension .jws, and copy it into the Axis web application anywhere outside the WEB-INF directory. We could do that, but it is too reminiscent of the JSP problem: if the server compiles the code, bugs only show up after deployment. Given that .jws files are really .java files, why can't we compile them in advance, just to make sure they work? We can, and that's what we are going to do.

There are two ways to do this. We could work with the .jws files and then copy them to files with a .java extension to test compile them. However, if we do that and we find a bug, clicking on the error string in the IDE will bring up the copied file, not the .jws original. Java IDEs don't know that .jws files are really Java source, so we would lose out on the method completion, refactoring, and reformatting that we expect from a modern IDE. Clearly, the second approach—save as .java files and copy to .jws during deployment—is the only sensible one. Of course, we have to differentiate the service files from normal Java files, which we do by keeping them out of the src directory tree, placing them into a directory called soap instead. In our build process, we have to give these files a .jws extension before copying them to a web server or into a WAR file. Figure 15.3 shows the build process.

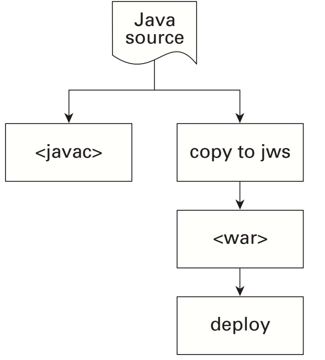

Figure 15.3
Our simple web service build process. The <javac> step is only there to validate files before deployment.

This is what our simple service looks like, with a stub implementation of our `search` call:

```
public class SearchService {

    private static String lastSearch="";
    private static final String[] emptyArray=new String[0];

    public String[] search(String keywords) {
        setLastSearch(keywords);
        return emptyArray;
    }

    public String getLastSearchTerm() {
        return lastSearch;
    }

    private void setLastSearch(String keywords) {
        lastSearch=keywords;
    }
}
```

The methods in bold are our service methods: one for end users, the search method that always returns the same empty string array, and the other a management call to see what is going on. A real system should split the management API into a separate endpoint, so that access could be restricted, but we aren't going to worry about that here.

To compile this code in Java, we have a short build file; it is so short, we include it in its entirety as listing 15.1

Listing 15.1 A build file to add a web service to an existing Axis installation

```
<?xml version="1.0"?>
 <project name="soapserver" default="default"
  basedir="." >

  <property name="endpoint"
    value="http://localhost:8080/axis/SearchService.jws"/>

  <property environment="env"/>
  <property name="build.dir" location="build"/>
  <property name="build.classes.dir" location="build/classes"/>

  <target name="default" depends="test"
    description="create a web service" >
  </target>

  <target name="init">
   <mkdir dir="${build.classes.dir}"/>
   <fail unless="env.CATALINA_HOME">Tomcat not found</fail>
  </target>

  <target name="clean">
   <delete dir="${build.dir}"/>
  </target>
```

```
  <target name="compile" depends="init">                                ❶
    <javac
     srcdir="soap"
     destdir="${build.classes.dir}"
     debuglevel="lines,vars,source"
     debug="true"
     includeAntRuntime="false"
     >
     </javac>
  </target>

  <target name="deploy" depends="compile">
    <copy                                                               ❷
      todir="${env.CATALINA_HOME}/webapps/axis/">
      <fileset dir="soap" includes="**/*.java"/>
      <mapper type="glob" from="*.java" to="*.jws"/>
    </copy>
  </target>

  <target name="test" depends="deploy">
    <waitfor timeoutproperty="deployment.failed"                        ❸
      maxwait="30"
      maxwaitunit="second">
      <http url="${endpoint}?wsdl" />
    </waitfor>
    <fail if="deployment.failed"
      message="application not found at ${verify.url}" />
    <echo>service is live on ${endpoint}</echo>
  </target>
</project>
```

The first few targets are the traditional `init`, `clean`, and `compile` targets; the only difference is the source directory for the compiler is now `"soap"` ❶. We do not need to add any of the Axis libraries to the classpath, because we do not reference them. All we need to do is declare public methods in our class and they become methods in a SOAP web service. We do have to be careful about our choice of datatypes if we want real interoperability; by restricting ourselves to integers, strings, and arrays of simple datatypes, we are confident that other SOAP implementations can call us.

The actual deployment task is a simple copy ❷ of all the Java files under the web directory into the CATALINA_HOME/webapps/axis/ directory tree. If we were deploying remotely, we would use FTP instead. That's it. No configuration files; no need to restart the server. If only all deployments were so simple.

Simple deployment or not, we need to verify that the deployment worked. Our `test` target tries to retrieve the WSDL description of the service for 30 seconds ❸; if it is successful, it reports that the service is live and the build succeeds.

15.4 ADDING WEB SERVICES TO AN EXISTING WEB APPLICATION

Now that we have shown the basics of web services and how to configure Tomcat to work with Axis, it is time to retrofit a SOAP endpoint to our existing web application. To do this we have add the appropriate libraries to the WEB-INF/lib directory, and then configure Axis to work. We need to make some changes to the web.xml configuration file to achieve that, but there we can use XDoclet.

15.4.1 Configuring the web application

Recall that in section 12.3.2, we configured the template files used by `<webdoclet>` to include servlets conditionally. We now need to add the Axis configuration details to the same template files. The first step is to extract the settings from the Axis/WEB-INF/web.xml file, so we open it in an editor and find the `<servlet>` and `<servlet-mapping>` tags. The servlet settings we insert into the servlets.xml file that `<webdoclet>` uses to build our web application's web.xml file is as follows:

```
<servlet>
  <servlet-name>AxisServlet</servlet-name>
  <display-name>Apache-Axis Servlet</display-name>
  <servlet-class>
    org.apache.axis.transport.http.AxisServlet
  </servlet-class>
</servlet>

<servlet>
  <servlet-name>AdminServlet</servlet-name>
  <display-name>Axis Admin Servlet</display-name>
  <servlet-class>
      org.apache.axis.transport.http.AdminServlet
  </servlet-class>
  <load-on-startup>100</load-on-startup>
</servlet>
```

The servlet mappings file sets up the bindings of these servlets to URL patterns beneath the server, one for the Axis admin servlet, the others providing the SOAP endpoints for the service clients. We find the values further down the Axis web.xml file and paste them into our templates/servlet-mappings.xml file:

```
<servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>*.jws</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
  <url-pattern>/servlet/AxisServlet</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>AxisServlet</servlet-name>
```

```
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>AdminServlet</servlet-name>
  <url-pattern>/servlet/AdminServlet</url-pattern>
</servlet-mapping>
```

The configuration data you need to paste may well vary from the beta-2 release, the latest version at the time of writing, and the final version, so follow the process we have described to get the most up-to-date settings.

15.4.2 Adding the libraries

The servlet configuration is not enough—we need to add the Axis libraries. We do this by bringing all the JAR files from the xml-axis/lib directory into our WAR file. Rather than do this naively, we first filter out any files that we don't need, such as log4j-core.jar and crimson.jar. We move these into a subdirectory called not_to_server. This lets us use the pattern lib/**/*.jar to pull in all the Axis jar files needed client side, while lib/*.jar works as the server-side pattern.

15.4.3 Including SOAP services in the build

Everything is ready; it is time to extend our web application with the SOAP entry point. We do this with three property declarations and one new target:

```
<property name="soap.src.dir"
  location="soap"/>

<property name="soap.classes.dir"
  location="${build.dir}/soap/classes"/>

<property name="soap.jws.dir"
  location="${build.dir}/soap/jws"/>

<target name="make-soap-api"
  depends="init">
  <mkdir dir="${soap.classes.dir}"/>
  <javac
    srcdir="${soap.src.dir}"
    destdir="${soap.classes.dir}"
    includeAntRuntime="false"
    >
    <classpath>
      <path refid="compile.classpath"/>
      <pathelement location="${build.classes.dir}"/>
    </classpath>
  </javac>
  <copy todir="${soap.jws.dir}">
    <fileset dir="${soap.src.dir}"
      includes="**/*.java"/>
    <mapper type="glob" from="*.java" to="*.jws"/>
```

```
  </copy>
</target>
```

This target compiles the service source using the full classpath of the project, including any Java files we have compiled. If that succeeds, it copies the Java files into a staging directory, renaming the files in the process. We then need to add two lines to our existing `<war>` task declaration, in the target `make-war`, and declare that this target depends upon our new `make-soap-api` target. The result is that the Axis libraries and our SOAP endpoint are now in our web application.

15.4.4 Testing the server for needed classes

We need to make sure the changes we have made to build process works, which means writing tests. Ant already runs the HttpUnit tests we wrote in chapter 12 immediately after deployment. We now need to add a test to fetch our endpoint's WSDL description to verify that Axis is working.

Because Axis configuration and deployment is more complex—needing someone to deploy the jax-rpc.jar outside the WAR file—this test is inadequate. If something goes wrong with the configuration, then the test will fail, but it won't provide clues as to a solution. It may provide an error trace starting with a `ClassNotFoundException`, but those errors mean nothing to the operations people who often install and configure production web servers. To avoid them calling in the engineering staff (us!) to diagnose the problem, we have to write tests with simpler diagnostics.

Our solution is to extend our JSP `<happy>` tag with a `classMustExist` attribute, which if set, triggers an attempt to instantiate the class with `Class.forName()` and throws a JSP exception if that attempt failed for any reason. Then we add a new attribute, `errorText`, which, if set, overrides the text of the `JspException` thrown when a test fails and provides more useful error messages. We had to do a bit of refactoring to do this cleanly. The result of these changes is that we can write a test file, happyaxis.jsp, containing tests for classes found in the different Axis libraries:

```
<%@ taglib uri="/WEB-INF/antbook.tld" prefix="happy" %>
<html><head><title>happy</title></head>
<body>
<happy:happy
  classMustExist="javax.xml.soap.SOAPMessage"
  errorText="saaj needs to be installed correctly"/>
<happy:happy
  classMustExist="javax.xml.rpc.Service"
  errorText="jax-rpc needs to be installed correctly"/>
<happy:happy
  classMustExist="org.apache.axis.transport.http.AxisServlet"
  errorText="axis.jar not found"/>
<p>Axis libraries are present</p>
</body>
</html>
```

This is a server-side mirror of the technique of using `<available>` and `<fail>` in a build file to validate build-time requirements; now we can test for classes existing on the server. This is a useful technique for any project with a complex deployment process.

15.4.5 Implementing the SOAP endpoint

With Axis integrated into our web application, and the deployment tests written, all that remains is to generate client side JUnit tests, write the real client application, and to bind the SOAP endpoint to our search code. We are going to tackle these in reverse order, implementing the API first. Writing the basic code to implement the searching from the SOAP endpoint turns out to be easy, although we are only returning the local path to the documents, not a network accessible URL.

```
public String[] search(String keywords) {
    try {
        setLastSearch(keywords);
        String[] results;
        Document[] docs = SearchUtil.findDocuments(keywords);
        results=new String[docs.length];
        for (int i=0; i < docs.length; i++) {
            results[i]=docs[i].getField("path");
        }
        return results;
    }
    catch (SearchException e) {
        return emptyAarray;
    }
}
```

Some more work needs to done to return the documents—perhaps a new method that returns the files as MIME attachments or a JSP page that serves it up in response to a GET. We could also turn any local exceptions into AxisFault exceptions; Axis will send these back over the network to the caller. We can evolve these features over time.

15.4.6 Deploying our web service

How do we deploy this web service? We already do this, because all we are doing is adding new libraries, files, and configuration data to our existing web application. The command `ant deploy` in our webapp directory is all we need to update our application.

After deploying, there are two web pages that we can fetch to test. First is our happy page:

```
http://localhost:8080/antbook/happyaxis.jsp
```

This should return the message "Axis libraries are present." We have added this page to our list of pages to fetch using HttpUnit, so our deployment target triggers an automatic probe of this page. The next page we fetch is the Axis-generated WSDL file for the service:

```
http://localhost:8080/antbook/SearchService.jws?wsdl
```

If the result of this fetch is an XML file, then everything is working. Axis has compiled the file and generated a WSDL description from its methods. Again, we modify our HttpUnit test to fetch this file during deployment, so that we automatically verify that Axis is working and that our SOAP endpoint is present whenever we deploy.

15.5 WRITING A CLIENT FOR OUR SOAP SERVICE

As usual, we want some unit tests, too, but this time we don't need to write them—we are going to make WSDL2Java do the work. The Axis utility will even generate JUnit test cases, one for each method, simply by setting the `--testcase` option. We do all this in a build file that is separate from the server and in a separate directory. We need nothing of the server other than its endpoint and the WSDL we can get from it. We will make Ant create the proxy classes and basic JUnit tests for us, and then we will write and execute the real tests and Java client. Figure 15.4 shows the process we will be implementing.

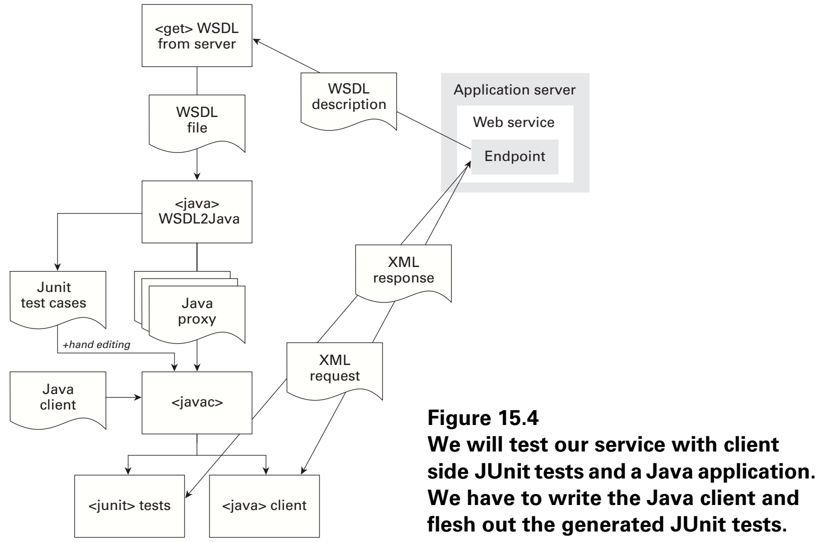

Figure 15.4
We will test our service with client side JUnit tests and a Java application. We have to write the Java client and flesh out the generated JUnit tests.

15.5.1 Importing the WSDL

To import the WSDL from our service, we just reuse our `<get>` target from 15.2.1. This time we bypass all the connectivity tests, because we are binding to a local server:

```
<property name="endpoint"
  value="http://localhost:8080/antbook/SearchService.jws?wsdl" />

<target name="fetch-wsdl" depends="init">
  <get src="${endpoint}" dest="${local.wsdl}"/>
</target>
```

```
<target name="import-wsdl" depends="fetch-wsdl">
  <java
    classname="org.apache.axis.wsdl.WSDL2Java"
    fork="true"
    failonerror="true"
    classpathref="axis.classpath"
    >
    <arg file="${local.wsdl}"/>
    <arg value="--output"/>
    <arg file="${generated.dir}"/>
    <arg value="--verbose"/>
    <arg value="--package"/>
    <arg value="soapapi"/>
    <arg value="--testCase"/>
 </java>
</target>
```

When we run the `import-wsdl` target, the WSDLToJava program creates the following files in the directory named in `${generated.dir}/soapapi`:

```
SearchService.java
SearchServiceService.java
SearchServiceServiceLocator.java
SearchServiceServiceTestCase.java
SearchServiceSoapBindingStub.java
```

These classes comprise the locator and proxy for the web service, as we saw in section 15.2, and a new class containing an automatically generated JUnit test case. We can use this generated test case as the framework for our test.

15.5.2 Implementing the tests

The generated JUnit test cases only test a call to the endpoint's methods with some parameters. For example, the test generated for the `search` method sends an empty string as the search term and does nothing with the return value:

```
package soapapi;

public class SearchServiceServiceTestCase extends junit.framework.TestCase
{

    public SearchServiceServiceTestCase(String name) {
        super(name);
    }

    public void test1SearchServiceSearch() {
        soapapi.SearchService binding;
        try {
            binding = new soapapi.SearchServiceServiceLocator().
                getSearchService();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            throw new junit.framework.
           AssertionFailedError("JAX-RPC ServiceException caught: " + jre);
        }
```

```java
        assertTrue("binding is null", binding != null);

        try {
            java.lang.String[] value = null;
            value = binding.search(new java.lang.String());
        }
        catch (java.rmi.RemoteException re) {
            throw new junit.framework.
                AssertionFailedError("Remote Exception caught: " + re);
        }
    }
```

Notice that this is a ready-to-compile JUnit test case; it subclasses `junit.framework.TestCase` and provides a valid constructor. Even without writing another line of code, we can test basic operation of our SOAP endpoint, and we can edit the test cases to test the service properly. There are three things that we must do to create real test cases. First, we must copy the generated file into our source tree, where it will not be overwritten, and move it outside the `soapapi` package, so that if we compile our source, and the generated directories, then no source file overwrites the other .class file. Next, we edit the test methods to send valid data to the SOAP service, and check for valid data coming back. For the test case above, we send a real search term and require a nonempty array back:

```java
try {
    java.lang.String[] value = null;
    value = binding.search("test");
    assertTrue("should have got an array back",
        value!=null && value.length>0);
}
catch (java.rmi.RemoteException re) {
    throw new junit.framework.
        AssertionFailedError("Remote Exception caught: " + re);
}
```

For the test of the `getLastSearchTerm` method, we search on a string and then verify that the service returns this string if we immediately call `getLastSearchTerm`. Doing so introduces a small race condition on a laden system, but we ignore it:

```java
public void test2SearchServiceGetLastSearchTerm() {
    soapapi.SearchService binding;
    try {
        binding = new soapapi.SearchServiceServiceLocator()
            .getSearchService();
    }
    catch (javax.xml.rpc.ServiceException jre) {
        throw new junit.framework.
            AssertionFailedError("JAX-RPC ServiceException caught: "
            + jre);
    }
    assertTrue("binding is null", binding != null);
```

```
    try {
        java.lang.String value = null;
        String searchTerm="test2";
        binding.search(searchTerm);
        value = binding.getLastSearchTerm();
        assertEquals(searchTerm,value);
    }
    catch (java.rmi.RemoteException re) {
        throw new junit.framework.
            AssertionFailedError("Remote Exception caught: " + re);
    }
}
```

To run the tests we have to compile the tests and proxy classes, now with a classpath containing Axis, Xerces, and JUnit, then use `<junit>`, to call the tests. We configure the task to search for `**/*TestCase.java`, rather than the usual `**/*Test.java`, and do this over the source and generated directories:

```
<target name="test" depends="compile"
        description="Execute unit tests">
  <junit printsummary="yes"
         errorProperty="test.failed"
         failureProperty="test.failed"
         fork="true">
    <classpath>
      <path refid="axis.classpath"/>
      <pathelement location="${build.classes.dir}"/>
    </classpath>
    <batchtest>
      <fileset dir="${client.src.dir}"
               includes="**/*TestCase.java"/>
    </batchtest>
  </junit>
</target>
```

The first time we ran these tests it failed in `test2SearchServiceGetLastSearchTerm`—we weren't getting back the last term we searched for. It turned out that Axis was creating a new object instance for each request, but we had expected a servlet style reentrant invocation. Until we made the `lastSearchTerm` field in our `SearchService` class static, it was being reset every invocation, causing the test to fail. This is, of course, exactly what functional tests are for: to validate your assumptions.[2]

---

[2] The web service deployment descriptor can enable sessions, but it also has to be set up on the client side. With JWS drop-in services, you do not get the option to specify this.

15.5.3 Writing the Java client

After the tests pass, we can write the real Java client:

```
import soapapi.*;

public class SearchClient {

   public static void main(String args[]) throws Exception {

        SearchServiceServiceLocator locator;
        locator=new SearchServiceServiceLocator();
        soapapi.SearchService service=locator.getSearchService();       ❶

        String lastTerm=service.getLastSearchTerm();
        System.out.println("last search = "+lastTerm);                   ❷

        String[] results=service.search(args[0]);
        for(int i=0;i<results.length;i++) {
            System.out.println(results[i]);
        }                                                                ❸
   }
}
```

This client has three stages.

❶ It finds and binds to the service.

❷ It retrieves and displays the previous search term, for curiosity .

❸ It sends the first argument of our application to the web service as a search term, and prints the results.

We have to run the program, of course, so let's write a target to invoke it with `<java>`:

```
<target name="run" depends="test">
  <java
    classname="SearchClient"
    fork="true"
    failonerror="true"
    >
    <arg value="deployment"/>
    <classpath>
      <path refid="axis.classpath"/>
      <pathelement location="${build.classes.dir}"/>
    </classpath>
  </java>
</target>
```

What happens when we run this? Well, we run the search and get a list of Ant documents that contain the word "deployment":

```
[java] last search = deployment
[java] /home/ant/docs/manual/OptionalTasks/ejb.html
[java] /home/ant/docs/manual/OptionalTasks/serverdeploy.html
              ...
[java] /home/ant/docs/manual/Integration/VAJAntTool.html
```

This means we have completed our example web service, from integration with our application all the way to our client, including both configuration checks to probe for Axis, and client-side functional tests to verify that the service does what we expect. We are now very close to being able to declare the service ready for production. One missing item is the extra server-side functionality to retrieve the indexed files; we will leave this until version 2.0. What we do have to do for version 1.0 is verify that our service is interoperable.

15.6 WHAT IS INTEROPERABILITY, AND WHY IS IT A PROBLEM?

Interoperability, or interop, as it is often called, is an ongoing issue with SOAP. The developers of SOAP toolkits, the SOAPBuilders, all work on interoperability tests to verify that foundational datatypes such as strings, integers, Booleans, arrays, and base64 encoded binary data can all be exchanged between clients and servers.

This is all very well, but it is not enough; complex types are not yet standardized. Consider the `HashTable` class: Java implements `java.util.HashTable` and .NET has its own implementation in `System.Collections.HashTable`. You can return one of these from a service you implement in your language of choice:

```
public HashTable getEmptyHashTable() {
    return new HashTable();
}
```

A client written to use the same toolkit as the service will be able to invoke this SOAP method and get a hashtable back. A client written in another toolkit, or in a different language, will not be able to handle this. If we were writing our server API by coding a WSDL file first and then by writing entry points that implemented this WSDL, we would probably notice that there is no easy way to describe a hashtable; consequently, we would define a clean name-value pair schema to represent it. Because we are developing web services the lazy way, by writing the methods and letting the run time do the WSDL generation, we do suffer from the hashtable problem. There is no warning at build time that the datatypes we are using in our service are not usable by other SOAP libraries, which means that we may only find out that we have an interop problem some time after we have deployed our service. We need to rectify this.

15.7 BUILDING A C# CLIENT

To detect interoperability problems early, we need to create a client with a different SOAP toolkit and then verify that it can call our service.

Although we could use the Sun web services toolkit, we chose, instead, to make life seemingly more complex by creating a C# client. It is a little known fact that there is a task in Ant to compile C# programs, the `<csc>` task, and that Ant 1.5 added the `<wsdltodotnet>` task to go alongside `<csc>`, purely to make C#-based interoperability testing inside Ant possible and easy. Because these tasks call down to programs

in the .NET framework SDK, they only work on a Windows PC with the SDK installed. You do not need the commercial Visual Studio.Net, just the downloadable SDK. The jEdit editor has a C# mode for editing, and we will build with Ant. The Ant .NET tasks have not been tested with either the Rotor public source version of .NET for FreeBSD or with the Ximian team's Mono implementation.

Building a .NET client for our service is nearly identical to building a Java version: we run a program to generate the stub classes, add an entry point class, build them, and then run the program with our chosen arguments. See figure 15.5.

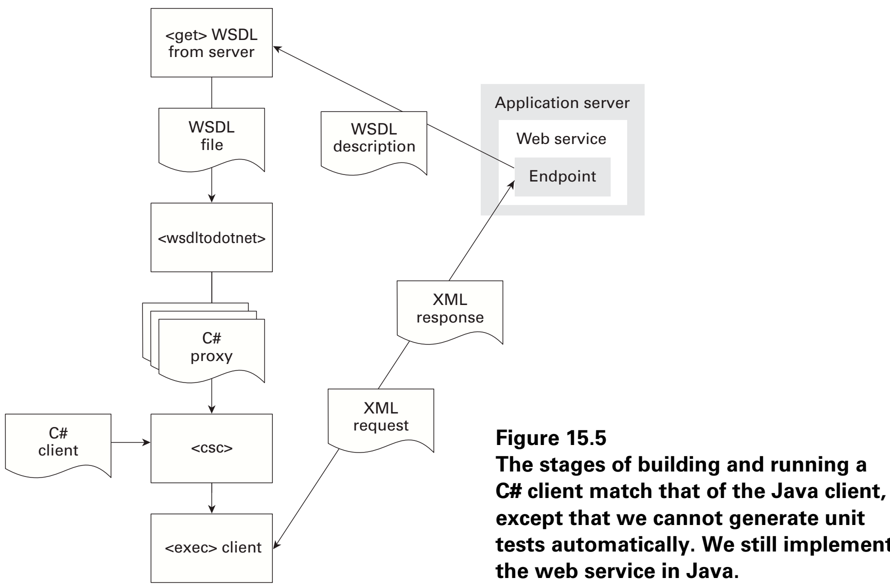

Figure 15.5
The stages of building and running a C# client match that of the Java client, except that we cannot generate unit tests automatically. We still implement the web service in Java.

15.7.1 Probing for the classes

Because we are only supporting the Windows implementation of .NET, we can only build the .NET client on Windows, and then only those versions with the .NET SDK installed and on the PATH. How do we restrict this? With a few moderately complex conditions:

```
<target name="probe_for_dotnet_apps" >
 <condition property="wsdl.found">
    <or>
      <available file="wsdl"     filepath="${env.PATH}" />
      <available file="wsdl.exe" filepath="${env.PATH}" />
      <available file="wsdl.exe" filepath="${env.Path}" />
    </or>
  </condition>
 <echo> wsdl.found=${wsdl.found}</echo>
 <condition property="csc.found">
    <or>
```

```
      <available file="csc"     filepath="${env.PATH}" />
      <available file="csc.exe" filepath="${env.PATH}" />
      <available file="csc.exe" filepath="${env.Path}" />
    </or>
  </condition>
 <echo> csc.found=${csc.found}</echo>
 <condition property="dotnetapps.found">
    <and>
      <isset property="csc.found"/>
      <isset property="wsdl.found"/>
    </and>
  </condition>
 <echo> dotnetapps.found=${dotnetapps.found}</echo>
</target>
```

These conditions ultimately set the `dotnetapps.found` property if we can find the programs wsdl and csc on the PATH; we don't tie ourselves to Windows explicitly, so if new platforms add the programs, we will try and use them.

15.7.2 Importing the WSDL in C#

The first step in creating the client is to generate C# source from the WSDL. We use the `<wsdltodotnet>` task to do this, feeding it the file we downloaded in section 15.5.2 in the `fetch-wsdl` target:

```
<property name="out.csc" location="${generated.net.dir}/soapapi.cs"/>

<target name="import-dotnet" depends="probe_for_dotnet_apps,fetch-wsdl"
  if="dotnetapps.found">
  <wsdltodotnet destFile="${out.csc}"
    srcFile="${local.wsdl}"
    />
</target>
```

This target creates a single file that contains the web service proxy class. Here is a fragment of the file:

```
[System.Web.Services.WebServiceBindingAttribute(
    Name="SearchServiceSoapBinding",
    Namespace="http://localhost:8080/antbook/SearchService.jws")]

public class SearchServiceService :
    System.Web.Services.Protocols.SoapHttpClientProtocol {

...

    /// <remarks/>
    [System.Web.Services.Protocols.SoapRpcMethodAttribute("",
       RequestNamespace="http://localhost:8080/antbook/SearchService.jws",
       ResponseNamespace="http://localhost:8080/antbook/SearchService.jws")]

    [return: System.Xml.Serialization.SoapElementAttribute("return")]

    public string getLastSearchTerm() {
```

```
        object[] results = this.Invoke("getLastSearchTerm", new object[0]);
        return ((string)(results[0]));
    }
...
}
```

We don't need to understand the details of this code, any more than we need to understand the proxy code that Axis generates. Note, however, that all the declarations in front of class and methods are *attributes*; these are like XDoclet tags except that you are really declaring constructors for objects that get serialized into the binary files. At run time, you can introspect the code to see what attributes are associated with the program, the class, the methods, or member variables. In our code, the web service support code in the .NET framework uses our declarations to bind properly to our service at run time.

15.7.3 Writing the C# client class

We can now write our C# client:

```
using System;

public class DotNetSearchClient {

    public static void Main(String[] args)  {

    SearchServiceService service=new SearchServiceService();

    String lastTerm=service.getLastSearchTerm();
    Console.WriteLine("last search = "+lastTerm);

    String[] results=service.search(args[0]);
    for(int i=0;i<results.Length;i++) {
        Console.WriteLine(results[i]);
        }
    }
}
```

By comparing this to the Java client in section 15.5.3, you will see that there is almost no difference between the Java and the C# client; indeed, we used cut-and-paste to create the C# client.

15.7.4 Building the C# client

Let's compile this code by using the <csc> task:

```
<property name="out.app" location="${build.net.dir}/netclient.exe"/>

<target name="build-dotnet" depends="import-dotnet"
  if="dotnetapps.found">
  <copy toDir="${generated.net.dir}">
    <fileset dir="${src.net.dir}" includes="**/*.cs" />
  </copy>
```

```
  <csc
    srcDir="${generated.net.dir}"
    destFile="${out.app}"
    targetType="exe"
    >
    </csc>
</target>
```

The <csc> task will compile all C# files in and below the srcDir directory, just as the <javac> task compiles Java source. Unlike <javac>, the output is not a directory tree full of object files. The task creates the executable, library, or DLL straight from the source files. The task does check dependencies, and rebuilds the target file if any of the input files have changed.

One little irritant of the task is that you can only specify one source directory. This prevents us from building our handwritten source together with the generated source. To fix this, we have to copy our handwritten source to the generated directory , before running the build. A consequence of this is that when we click on any error line in an Ant hosting IDE, the IDE brings up the duplicate file, the one being compiled, not the master copy. We have to be very careful which file we are editing. We may enhance this task to support multiple source directories; as usual, check the documentation.

15.7.5 Running the C# client

With the code compiled, it is time to run it, this time with <exec>:

```
<target name="dotnet" depends="build-dotnet" if="dotnetapps.found">
  <exec
    executable="${out.app}"
    failonerror="true"
    >
    <arg value="deployment"/>
  </exec>
</target>
```

What is the result of this? Well, we get nearly the same results as before—because we are running against a local server on a Windows system, the file paths we get back are all Windows based:

```
[exec] last search = deployment
[exec] C:\jakarta-ant\docs\manual\OptionalTasks\ejb.html
[exec] C:\jakarta-ant\docs\manual\OptionalTasks\serverdeploy.html
  ...
[exec] C:\jakarta-ant\docs\manual\Integration\VAJAntTool.html
```

This is exactly what we wanted—to call our Java web service from a C# client. Now that we have this dual-language client import-and-build process working, we can keep using it as we extend the classes.

15.7.6 Review of the C# client build process

As you can see, it is not that much more complicated to build a C# program in Ant than it is to build a Java application: the fact that Ant is the ubiquitous Java build tool does not mean that it can only build Java programs, that is merely what it is best at. In chapter 17, we will go one step farther and build native C++ code.

The reason we are compiling C# code here is not because we have a big C# project, but because we need to verify that our Java-based web service is interoperable with the other SOAP implementations. The process for doing so is the same for all target languages: import the WSDL, write an entry point or other test case, and run them. Were we writing our web service in another language such as C# or Perl, we would be able to use our build file to create an Axis/Java client to test the service, complete with generated JUnit test cases.

Often the act of running the WSDL importers is a good initial test of interoperability, extending the entry point even better. It's a pity that the Microsoft toolkit doesn't generate NUnit tests for us to use alongside the JUnit tests; we have to do these by hand. If we did start developing a complex .NET client, we might find ourselves taking a closer look at NAnt, a .NET version of Ant, found at SourceForge (http://nant.sourceforge.net), and maybe `<exec>`, the NAnt build from our Ant task. Alternatively, we might write an `<nunit>` task for Ant.

Finally, we need to state that the hashtable problem is a fundamental problem with web services: it is too easy to write a web service whose methods can only be called by clients that use the same language and toolkit implementation as the service. This belies the whole notion of using XML-based web services as a way to communicate across languages. Something needs to be done to address this.

15.8 THE RIGOROUS WAY TO BUILD A WEB SERVICE

The most rigorous approach to building a web service is to create a WSDL specification of the interface, and perhaps an XSD description of all the datatypes. SOAP has its own syntax for declaring simple datatypes, but because XSD is more standardized, we encourage you to follow the XSD path.

The other aspect of rigorous service development is to implement the service in a Java file, and not as a JWS page, which lets you bypass the copy-based renaming of Java source to JWS pages. The Java files just live in the same source tree as the rest of the web application, and are validated by the build-time `<javac>` compile of the main source tree.

We don't go into detail on this more rigorous server-side development process. We could probably write a whole new book on how to build, test, and deploy web services with Ant, and get into much more detail into how SOAP and Axis work. What we can do is provide some directions for you to follow, if you want to explore this problem. One of the best starting points is actually the test server classes you can find in the Axis CVS tree; these are the most up-to-date examples of service generation.

To turn a Java class into a SOAP endpoint, you need to provide a Web Service Deployment Descriptor (WSDD) that tells the Axis run time what the attributes of the service are. In the descriptor, you must name the service and the class that implements it, and which class methods are to be accessible via SOAP. You can also register handlers for SOAP headers. These are the SOAP equivalent of headers in an HTTP request: little fragments of information that the SOAP endpoint or other server-side code can use to implement features such as security and sessions. You could use HTTP headers instead, but the SOAP header model integrates better with an XML-based communication system, and works when you use alternative transports such as email.[3] If you want to do complex SOAP handling, a deployment descriptor file is mandatory; this means that you must use Java and not JWS files to implement your service.

After deploying your application, you have to register your WSDD files with the Axis administration servlet. Unless you change this server to be accessible remotely, you need to run code server side to register each deployment descriptor, and you need to make a list of all the WSDD files to register. You can call the administration program from a build file via `<java>`, so registering local builds is easy.

Based on our past examples of generating XML descriptor files from Java source, readers no doubt expect a new XDoclet task at that point. Unfortunately, we can't provide one because XDoclet does not support Axis at the time of writing. We expect this to be fixed eventually; the XDoclet team has promised us that they will be writing tags for the Sun toolkit, so a matching Axis set makes sense.

When you are being fully rigorous, you write the XSD and then the WSDL files before you implement your service class. Writing these files can be problematic; the CapeClear editor (http://www.capeclear.com/) is the best there is for this purpose. After writing the WSDL file, call WSDL2Java with the `-server` attribute, and the program generates the server-side stubs for your service You can take these generated classes and implement your web service behind them.

15.9 REVIEWING WEB SERVICE DEVELOPMENT

We have just set up an advanced build process to add SOAP support to our application. Adding the Axis libraries and configuration settings to our existing web application was relatively simple, but it forced us to add new deployment tests for missing classes, implemented through our existing `<happy>` JSP page. With the libraries and configuration all working, we can create web services simply by saving Java source files with a .jws extension in the main web application directory.

Writing the service is half the problem; testing it, the remainder. The Axis client-side utilities come into play here, creating Java proxy classes from our services' WSDL description. The WSDL2Java class can even generate basic JUnit test cases, which can act as a foundation for hand-coded unit tests.

---

[3] There is still one good reason for using cookies: hardware load balancers can direct requests to specific servers based on cookie values.

Web services are an area of heated development. Axis will evolve, Sun is coming out with its own web service package, and, inevitably, Ant will acquire wrapper tasks to simplify the stages of the build using Apache, Sun, and other toolkits.

Ultimately, web services are distributed applications scaled up. If you are writing one, you are writing an application to work across the Internet, interoperating with systems written in other languages, communicating over a protocol (HTTP) that is chosen because it can get through firewalls, not because it is the best protocol for such things (it isn't). This is a major undertaking. Ant alone is not adequate. What Ant gives you is the means to build, deploy, and test your code, including automated generation of client-side stub classes and test cases. It is not a silver bullet. It is, however, along with JUnit, an essential tool for this kind of project.

15.10 CALLING ANT VIA SOAP

If calling SOAP services from a build file lets your program use remote services from the build file, what is a good service to use? How about Ant itself?

Rant, Remote Ant, is a project under way at SourceForge (http://sourceforge.net/projects/remoteant/). This project contains a web application that gives you remote Ant access via a SOAP interface. You can submit a request from a remote system, naming a build file and a target in the file to execute. The servlet executes the build, returning success or failure information.

This is a nice model for implementing a distributed build process, in which different machines in a cluster take on different parts of a big build. It could also be useful for a build process in which a single central machine was the reference build system; developers could use Rant to trigger a new build on this system from their own machine. If the build process is sufficiently complex, especially if it integrates with native compilers or a local database, a centralized build does start to make sense, even if a replicable build environment were preferable. To trigger a remote build you simply invoke it via an Ant task:

```
<taskdef name="rant"    <-- Declares the task
  classname="com.einnovation.rant.RantTaskDef">
  <classpath>
    <fileset dir="lib">
      <include name="*.jar"/>
    </fileset>
  </classpath>
</taskdef>

<property name="endpoint"
  value="http://127.0.0.1:8080/rant/servlet/rpcrouter" />
<property name="target.file" location="../soap/soap.xml" />

<target name="default" >
  <rant buildFile="${target.file}"
      soapURL="${endpoint}"          Calls the
      target="default"/>             remote build
</target>
```

That SOAP is the marshaling layer is irrelevant, except that it lets you trigger remote Ant builds from any language that has a compatible SOAP library: Perl, Python, maybe even the Microsoft.NET framework.

You should not place the Rant service up on a generally accessible web server. Allowing any caller to invoke any Ant file in the system is a significant security issue. Even worse, if the server supported anonymous FTP, a malicious person could upload the build file before referring to it.

Neither of the authors uses this tool in any serious manner, but we like the idea. If we did use it, we would change the API so that you could only select from a limited number of build files, which would significantly lessen the security implications. The other major issue that needs fixing in the current release, version 0.1, is that the service does not return the output of the remote build. All you get now is a success message or the failure exception; it needs to return the log as XML for postprocessing. There is also the issue that Rant uses the original Apache SOAP product, not Axis; Axis has better interoperability.

To use Rant, you need to install its web application on your server. After the application server expands the application, you may need to update rant/WEB-INF/lib with later Ant versions, and any libraries you need for optional tasks. This is because it contains its own version of Ant in the web application's lib directory.

Because the Rant tool is still in its infancy, we would expect it to address issues such as these in future versions. It could become an essential and useful part of every complex build process, replacing those deployment processes in which the client build file uses the `<telnet>` task to connect to a remote server and run Ant remotely.

15.11 SUMMARY

We explored some aspects of integrating with SOAP-based web services. We demonstrated how to fetch a WSDL description of a web service, and how to use Axis to generate local proxy classes that you can integrate with your own source to create a working web service client. As web services become more common and the SOAP implementations more stable, an increasing number of people will use Ant to build web service servers or clients. What we covered here is a foundation.

The easy way to add a web service to your existing web application is to give a Java file the .jws extension and place it in the web application alongside HTML or JSP pages. Axis, if you have installed and configured it correctly, will compile the file and export it as a SOAP endpoint.

After exposing the endpoint, comes the other half of the problem: the client side. We covered how to build Java and C# clients in Ant, both of which follow a similar process. You fetch the WSDL description of the service, generate proxy classes, and then compile and run these classes against hand-coded client applications. Because interoperability is such an issue with SOAP, you need to continually import and build client applications in as many languages and frameworks as you can manage.

If you only build clients in the same language and SOAP toolkit as that of the server, you may not discover that your service suffers from the hashtable problem until the service goes live, which is never a good time to find out that you have a fundamental design problem.

If you are working in the web services area, you should read some of our other work, which will give you more insight into how to design, develop, and deploy these systems (Loughran 2002-1, Loughran 2002-2). Working with web services can be fun, but there are many challenges to address. Ant can certainly make the process more tractable.