APPENDIX C

Making Java Groovy

Kenneth A. Kousen

FOREWORD BY Guillaume Laforge

MANNING

Making Java Groovy

by Kenneth A. Kousen

Appendix C

Copyright 2014 Manning Publications

brief contents

appendix C
SOAP-based web services

A few years ago, many businesses decided to integrate their internal systems together into a Service-Oriented Architecture using web services. The promise was that automated tools would make the integration straightforward, and by assembling a coherent, reusable set of services, development time for new systems could be considerably reduced.

The idea makes sense when viewed as a way to connect applications written in different languages, running on different operating systems, possibly even deployed in different locations. After all, every language can read and write strings, and what are SOAP-based web services but highly formatted strings transmitted over a network?

Though today RESTful web services are more popular (see chapter 10), many organizations spent considerable resources building a SOAP-based SOA infrastructure. A typical Java developer in the field almost inevitably will have to deal with them sooner or later. The question posed in this appendix, as always, is to ask if Groovy can make that job simpler and easier.

As it turns out, this is another area in which the guiding principle stated in chapter 1 works: Java is good for tools, libraries, and infrastructure, and Groovy is good for everything else. In this appendix, I'll demonstrate how Groovy can be used with the standard Java tools to build SOAP-based clients and services, and see where else Groovy can simplify the overly complex world that is SOAP, WSDL, and lots of XML.

C.1 A quick review of SOAP and WSDL

The web services in this appendix are based on the XML markup languages known as SOAP and WSDL. In this section I'll review the basics of each. After that, I'll use the Java-supplied tools to build web services clients and services, and show how Groovy implementations factor into each.

C.1.1 SOAP

In the original Note for SOAP 1.1 (www.w3.org/TR/2000/NOTE-SOAP-20000508/) prepared by the World Wide Web Consortium (W3C), SOAP was defined as an acronym for Simple Object Access Protocol. In the current version of the specification (1.2), SOAP is now a full technical recommendation of the W3C (www.w3.org/TR/soap12/) and no longer stands for anything.

SOAP is designed to be a simple wrapper around an XML payload (see figure C.1). Because SOAP-based web services are supposed to be independent of transport protocol, the wrapper provides a mechanism for specifying meta information like security and transactions. SOAP 1.2 may be current, but version 1.1 is still the most common version found in practice.

The schema for SOAP defines a root element called `<Envelope>`, which contains an optional `<Header>` and a mandatory `<Body>`. The contents of the header and body are nearly arbitrary, which makes sense because both contain information supplied by the client outside the SOAP specification.

The longest section of the SOAP 1.1 specification discusses how data types are to be *encoded*, or represented, in XML. SOAP-based web services are normally viewed as a way to do remote procedure calls, so a mechanism is needed to define method arguments and return types. This part of the specification is contained in section five, and ever since, the mechanism for converting data types to XML has been known as SOAP Section 5 Encoding. This encoding scheme still shows up in older web services, but in general newer services use the XML Schema specification, which will be discussed further in section C.1.2.

If you take out section five, the SOAP specification really is Simple. The Object Access part is still unclear, but SOAP certainly is a Protocol, so I guess the lost acronym makes some sort of sense.

In principle, you can build SOAP messages programmatically through an API. The relevant Java API is called SAAJ, which stands for SOAP with Attachments API for Java. SAAJ is a thin layer that maps very closely to the SOAP elements, with classes like `SOAPBody`, `SOAPElement`, and `SOAPHeader`. Still, constructing and manipulating XML through the API is tedious at best. Instead, Java provides tools and infrastructure for creating and transmitting SOAP messages automatically. The tools are based on having an XML file full of information describing the service. That file is called a Web Services Definition Language file, or WSDL, and will be discussed next.

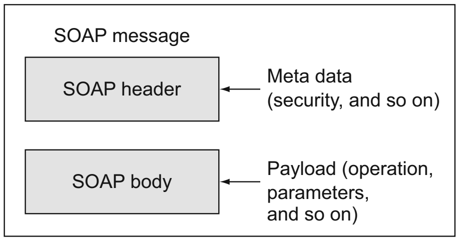

SOAP message
SOAP header
Meta data (security, and so on)
SOAP body
Payload (operation, parameters, and so on)

Figure C.1 A SOAP message containing an optional header and a mandatory body. The header contains header elements, normally processed by intermediaries. The body holds the message, which usually consists of an element representing the operation and child elements for request parameters.

C.1.2 WSDL

The WSDL specification (version 1.1 of which can be found at www.w3.org/TR/wsdl) contains all you need to build a client of a web service. WSDL files contain

- An XML schema, which defines all the elements and attributes (if any) used to map the data types used by the operations
- A `<portType>` element, which contains method signatures for the `<operation>`s
- Descriptions of any `<input>` and `<output>` messages used for request/response operations
- A `<binding>` section, which explains how to create and transmit the messages
- A `<service>` element, which lists where the service resides and links the bindings to the port types so messages can be sent either way

In J2EE 1.4, the web services specification for Java was called the Java API for XML-based Remote Procedure calls, or JAX-RPC. The current specification is the Java API for XML-based Web Services, or JAX-WS. Of particular interest here is that several of the tools implemented by JAX-WS are now part of the Java 1.6 Standard Edition. This means that in this appendix I can demonstrate everything with a regular JDK.

Given a WSDL file, the relevant Java tool is called `wsimport`. That command generates and compiles classes called stubs, which construct SOAP messages from Java method invocations, transmit them to a service, receive the SOAP responses, and translate them back to into Java. The `wsimport` tool (and similar tools from other vendors) is one of the keys to the value of SOAP-based web services. With only the WSDL file available, a fully functional client for a web service can be generated without dealing with XML at all.

The other important tool Java includes is called `wsgen`. It generates the WSDL file from existing Java code. One method for implementing web services is to write the service code first, or at least build its skeleton, and then generate the WSDL file. This is the so-called *bottom-up* approach. Alternatively, the WSDL itself can be written, or generated from other modeling tools, and then the service is implemented based on it. That’s the *top-down* approach. The bottom-up approach is easiest to illustrate, so I’ll use it in this appendix. I’ll use both those tools later in this appendix, first to illustrate how web services work, and then to see how the tools interact with Groovy.

C.2 Building a Groovy web service client

The goal in this section is to build a client for an existing web service. This is very easy, because the Java tools do most of the work. The Java tool in question is the `wsimport` tool described in the previous section, which generates stubs in Java. A Groovy client instantiates the generated stub and calls methods as usual. In this section I’ll pick a web service, review its WSDL file to see what it does, generate the stubs using the Java tool, and build a Groovy client that uses the stubs.

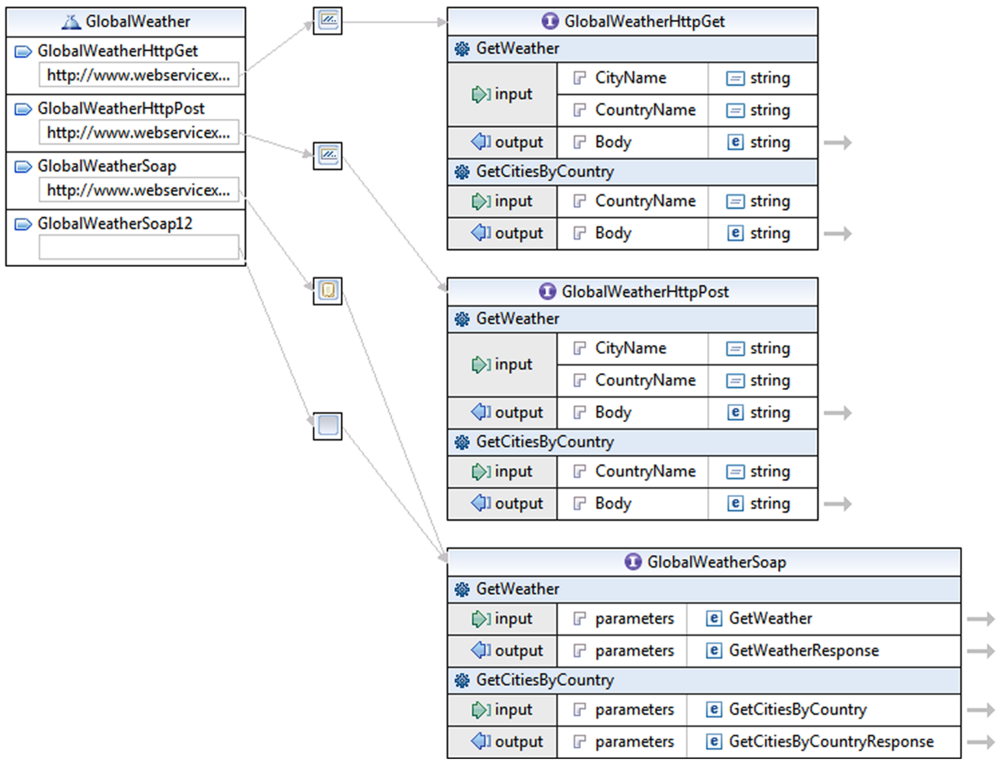


Figure C.2 A graphical view of the WSDL file for Microsoft's Global Weather web service. The service element is on the left, the port types (interfaces) are on the right, and the bindings are in the middle.

C.2.1 The Global Weather web service

To demonstrate this process I need an existing, publically available, preferably simple web service. To make sure there's no Java bias anywhere in this system, I'll pick one that I know to be implemented in .NET, called the Global Weather service.

Microsoft makes available a set of SOAP-based web services at www.webservicex.net. One of these is called the Global Weather service, which provides weather conditions for cities around the world. The WSDL file for this service can be accessed at www.webservicex.net/globalweather.asmx?WSDL.

That's a standard pattern, by the way. You access the WSDL file associated with a web service at the service endpoint (the URL without the query string) and append ?WSDL to the end. The fact that the endpoint address in this case ends with asmx is another clue that the service is implemented using .NET.

Figure C.2 shows a graphical view of the WSDL file from Eclipse.

Microsoft web services, by default, embed the name of the transport protocol (SOAP, HTTP GET, or HTTP POST) into the name of the interface. Because I'm only interested in the SOAP 1.1 service, I'll focus on the port type (shown with an "I" for "interface") called `GlobalWeatherSoap`. The XML defining the port type is shown next:

```
<wsdl:portType name="GlobalWeatherSoap">
    <wsdl:operation name="GetWeather">
       <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get
     weather report for all major cities around the world.</
     wsdl:documentation>
       <wsdl:input message="tns:GetWeatherSoapIn" />
       <wsdl:output message="tns:GetWeatherSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="GetCitiesByCountry">
       <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get
     all major cities by country name(full / part).</wsdl:documentation>
       <wsdl:input message="tns:GetCitiesByCountrySoapIn" />
       <wsdl:output message="tns:GetCitiesByCountrySoapOut" />
    </wsdl:operation>
</wsdl:portType>
```

The GlobalWeatherSoap interface contains two operations, called GetWeather and GetCitiesByCountry. Each is a request/response service, because they each have a single input message and a single output message.

To call the methods I need to know what to supply for parameters and what I'm going to get back. The included schema shows that for the GetWeather operation the arguments are a CityName and a CountryName, both of which are strings:

```
<s:element name="GetWeather">
    <s:complexType>
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
                  name="CityName" type="s:string" />
            <s:element minOccurs="0" maxOccurs="1"
                  name="CountryName" type="s:string" />
        </s:sequence>
    </s:complexType>
</s:element>
```

Both the CityName and CountryName elements have a minOccurs attribute of 0, implying they are optional, but presumably in a normal request both would be set. The response to the operation is called, naturally enough, a GetWeatherResponse, which contains a string:

```
<s:element name="GetWeatherResponse">
    <s:complexType>
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
                name="GetWeatherResult" type="s:string" />
        </s:sequence>
    </s:complexType>
</s:element>
```

Similarly, in reading through the WSDL I can see that the GetCitiesByCountry operation takes a string argument called CountryName and returns a string called GetCitiesByCountryResult.

The name of the `<service>` element is `GlobalWeather`. The service contains three *ports*, each of which has the name of the service with the name of the transport protocol appended. Again, all I care about is the `GlobalWeatherSoap` port. It contains an `<address>` element that gives the endpoint address, or the URL where the service can be accessed. Here's the XML for the service element in question:

```
<wsdl:service name="GlobalWeather">
    <wsdl:port name="GlobalWeatherSoap" binding="tns:GlobalWeatherSoap">
        <soap:address location=
            "http://www.webservicex.net/globalweather.asmx" />
    </wsdl:port>
    ...
</wsdl:service>
```

Finally, the `<binding>` element contains lots of XML, but it all boils down to saying that the service is in the "document/literal wrapped" style and transmitted as SOAP over HTTP. This is typical of most web services developed in the past few years, and fortunately matches what the tools expect as well.

C.2.2 Generating the stubs

Now it's time to use the tool. The `wsimport` script is a command-line tool that flags the WSDL location as well as where to put the generated stubs. In my system, I created a typical Eclipse project, which contained src and bin directories for the source code and compiled files, respectively. That means I used the tool by opening a command prompt in the root of the Eclipse project and typing the following string:

```
c:\> wsimport -d bin -s src -keep \
         http://www.webservicex.net/globalweather.asmx?WSDL
```

The command arguments say to put the compiled files in the bin directory and any generated source in the src directory, keep the source (otherwise only the compiled files would remain), and use the specified URL for the location of the WSDL file. Because no package name was specified for the generated code, a package based on the target namespace (www.webservicex.net) was used by default. The resulting classes are shown in figure C.3.

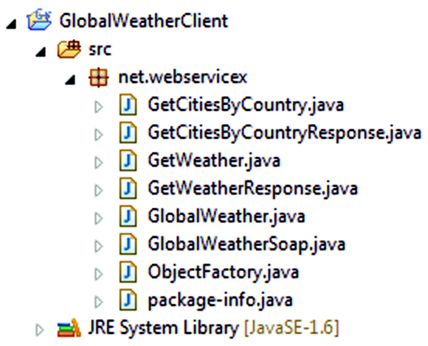

GlobalWeatherClient
src
net.webservicex
GetCitiesByCountry.java
GetCitiesByCountryResponse.java
GetWeather.java
GetWeatherResponse.java
GlobalWeather.java
GlobalWeatherSoap.java
ObjectFactory.java
package-info.java
JRE System Library [JavaSE-1.6]

Figure C.3 The net.webservicex package contains the classes generated by the `wsimport` tool. There's a class for each operation and a class for each request and response type for the operations.

C.2.3 Building the Groovy client

All the generated files are, of course, implemented in Java. To access the web service, a client instantiates the service stub and invokes the operations. According to the JAX-WS specification, the client stub is created according to the following pattern:

```
<portType> stub = new <service_name>().get<port_name>Port()
```

A simple client for the Global Weather service looks like

```
GlobalWeatherSoap stub = new GlobalWeather().getGlobalWeatherSoapPort()
```

Then I use the stub to make method calls as though the service is local. Note that the client can be written in either Java or Groovy, and until this point the distinction doesn't matter. As it happens, though, a Groovy client is definitely preferred here, because unlike most web services that return objects, this particular service returns a chunk of XML.

The operation `GetCitiesByCountry` returns the list of valid cities in a given country. A bit of experimentation shows that although the return type of the `GetCities-ByCountry` operation is `String`, the value is formatted XML that looks similar to the following listing.

Listing C.1 The `String` output for the `GetCitiesByCountry` operation for Canada

```
<NewDataSet>
  <Table>
    <Country>Canada</Country>
    <City>Nitinat Lake Meteorological Aero</City>
  </Table>
  <Table>
    <Country>Canada</Country>
    <City>Ile Rouge Meteorological Aeronau</City>
  </Table>
  <Table>
    <Country>Canada</Country>
    <City>La Scie, Nfld.</City>
  </Table>
...
  <Table>
    <Country>Canada</Country>
    <City>Belle River</City>
  </Table>
</NewDataSet>
```

Processing this in Java is as ugly as processing any other block of raw XML. You need either a SAX parser or a DOM parser, and then you need to search for the desired elements and extract their contents, keeping in mind that the character data of an element isn't the value of the element, but rather is the value of the first child of the element.

Groovy, however, has no problem parsing XML and walking the resulting tree to find what you want. Dealing with XML is one of Groovy's sweet spots, as shown in chapter 4 on integration.

For example, the next listing shows a Groovy client that prints the first 10 city values in the response for Canada.

Listing C.2 Parsing the XML string returned by `GetCitiesByCountry`

```
import net.webservicex.GlobalWeather;
import net.webservicex.GlobalWeatherSoap;

GlobalWeatherSoap stub = new GlobalWeather().globalWeatherSoap
def dataSet = new XmlSlurper().parseText(stub.getCitiesByCountry("Canada"))
def cities = dataSet.Table.City

println "There are ${cities.size()} cities"
cities[0..10].each { println it }
```

In three lines of Groovy I instantiate the stub, call the `getCitiesByCountry` method, parse the resulting XML, and extract the values of all the `City` elements. I then print out the total number of cities and list the first 10:

```
There are 561 cities
Nitinat Lake Meteorological Aero
Ile Rouge Meteorological Aeronau
La Scie, Nfld.
Amherst, N. S.
Erieau Meteorological Aeronautic
Amphitrite Point
Coronach Spc
Argentia, Nfld
Pam Rocks
Sundre
St. Anthony, Nfld.
```

The web service returns 561 Canadian cities. The listing of the first 10 shows that apparently there's a length limit in the web service for the name of the associated weather station, but there aren't any other surprises.

The other operation exposed by this web service is called `GetWeather`. I appended the following lines to the previous listing:

```
def city = 'Ottawa'
def country = 'Canada'
println "The weather report for $city, $country is"
println stub.getWeather(city, country)
```

The resulting output once again is in XML form, via the returned string:

```
The weather report for Ottawa, Canada is
<?xml version="1.0" encoding="utf-16"?>
<CurrentWeather>
  <Location>Ottawa Int'L. Ont., Canada (CYOW) 45-19N 075-40W 114M</Location>
  <Time>Aug 31, 2010 - 04:00 PM EDT / 2010.08.31 2000 UTC</Time>
  <Wind> from the SW (230 degrees) at 13 MPH (11 KT):0</Wind>
  <Visibility> 15 mile(s):0</Visibility>
  <SkyConditions> mostly cloudy</SkyConditions>
  <Temperature> 89 F (32 C)</Temperature>
  <DewPoint> 71 F (22 C)</DewPoint>
  <RelativeHumidity> 55%</RelativeHumidity>
  <Pressure> 30.01 in. Hg (1016 hPa)</Pressure>
  <Status>Success</Status>
</CurrentWeather>
```

Once again, to access the individual details it's easy enough to use the `parseText` method from `XmlSlurper` and get the child values:

```
GlobalWeatherSoap stub = new GlobalWeather().globalWeatherSoap
def xml = stub.getWeather('Ottawa', 'Canada')
def root = new XmlSlurper().parseText(xml)
println "The temperature is ${root.Temperature}"
```

The script then prints

```
The temperature is  -4 F (-20 C)
```

Using Groovy on the Global Weather web service is helpful because the designers of the service restricted their outputs to simple strings, even though the results were XML complex types. The stubs, though, were Java. Groovy is able to instantiate and invoke methods on the stubs without any changes.

In most cases, especially when the web service is generated from existing code, the returned values won't be just blocks of XML. Instead, for Java, the Java API for XML Binding will create classes to map to the complex XML types in the schema. I'll deal with this situation in the next section when I create a web service.

C.3 Building a web service in Java and Groovy

As the previous section demonstrated, building a web service client starts with applying the `wsimport` tool to the service's WSDL file. The tool produced Java stubs that the Groovy client used as usual. In this section I'll look at the opposite problem: implementing the web service itself. I'll take the bottom-up approach, writing the implementation code first and then using the `wsgen` tool to generate the WSDL file. Again, the goal is to see how the standard Java tools work in general, and see what happens when I use Groovy for the service implementations rather than Java. Groovy implementations tend to be shorter, simpler, and more powerful, so if I can use Groovy where I would normally use Java and the tools still work, the development tasks will be easier.

When you learn about web services, most of the available tutorials use as their "Hello, World" application what I like to call the World's Slowest Calculator. The service consists of add, subtract, multiply, and divide methods in a class, and everything is generated from that. The advantage to this example is that the implementation is extremely simple, so it doesn't distract from the tools and concepts. Just try not to think about the fact that I'm generating SOAP messages and transmitting them over the network just to add a couple of numbers.

Two terms are relevant here. First is the Service Endpoint Interface (SEI), which is the interface containing the methods exposed by the web service. The other is the Service Implementation Bean (SIB), which implements the interface. I'll use a Java SIB first and then write a Groovy one to illustrate the tool differences, if any. Then I'll use the tools to implement a web service in Java and see how it changes when I move to Groovy.

C.3.1 A Java SIB for the Calculator

A bottom-up web service starts with a POJO. I sprinkle in JAX-WS annotations,[1] which are used by the tool when it generates the WSDL file. The following listing shows the Calculator SIB, implemented as a POJO with annotations.

Listing C.3 A Java SIB with web service annotations

```
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

@WebService
public class Calculator {

    @WebResult(name="sum")
    public double add(
            @WebParam(name="x") double x,
            @WebParam(name="y") double y) {
        return x+y;
    }

    @WebResult(name="difference")
    public double subtract(
            @WebParam(name="x") double x,
            @WebParam(name="y") double y) {
        return x-y;
    }

    @WebResult(name="product")
    public double multiply(
            @WebParam(name="x") double x,
            @WebParam(name="y") double y) {
        return x*y;
    }

    @WebResult(name="quotient")
    public double divide(
            @WebParam(name="x") double x,
            @WebParam(name="y") double y) {
        return x/y;
    }
}
```

Expose as web service

Return parameter in WSDL

Operation arguments in WSDL

Return parameter in WSDL

Operation arguments in WSDL

All that's needed to turn a POJO into a web service is the `@WebService` annotation. This annotation has various optional parameters, but here I'm relying on the defaults. Because the WSDL is generated using reflection, the operation arguments and return types all wind up with generic names, like `arg0`, `arg1`, and `return` in the

[1] Strictly speaking, the annotations are from a different specification than JAX-WS. For simplicity, however, I'll refer to everything related to JAX-WS as being from that spec.

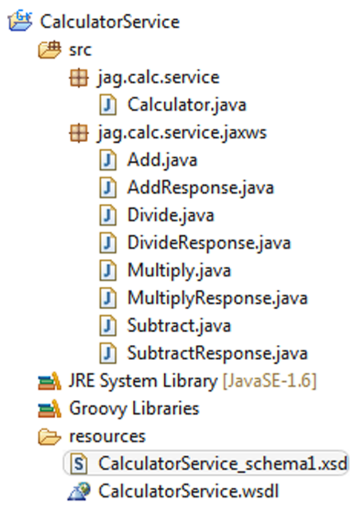
WSDL file. I used the `@WebResult` and `@WebParam` annotations to set the names in the WSDL file. Because the WSDL file is the contract between the clients and the service, and the WSDL is exposed to the outside world, it seems prudent to use realistic names in it.

This service avoids a lot of issues because all the arguments and return types are doubles. Therefore, no annotations from the Java API for XML Binding (JAXB) specification are needed yet, because Java doubles map naturally to XML schema doubles. Later I'll show a more complex example in which extra work will be needed for the argument and return type mappings.

I created an Eclipse project containing this class, as in the previous section. This time I want to run the `wsgen` tool, with the arguments shown:

```
C:\> wsgen -d bin -s src -r resources -wsdl -cp bin calc.service.Calculator
```

The command-line arguments say to store any generated classes in the bin directory, save any generated source code in src, store the generated WSDL file in the resources directory (which has to be created ahead of time), and generate everything from the compiled `Calculator` class, which resides in the bin directory. After running the `wsgen` command, the updated project is as shown in figure C.4.



```
CalculatorService
  src
    jag.calc.service
      Calculator.java
    jag.calc.service.jaxws
      Add.java
      AddResponse.java
      Divide.java
      DivideResponse.java
      Multiply.java
      MultiplyResponse.java
      Subtract.java
      SubtractResponse.java
  JRE System Library [JavaSE-1.6]
  Groovy Libraries
  resources
    CalculatorService_schema1.xsd
    CalculatorService.wsdl
```

Figure C.4 The updated Eclipse project, showing the generated classes in the `jag.calc.service.jaxws` package and the generated WSDL and schema in the resources directory

The tool generated two classes for each operation, one for the request and one for the response. It also generated a WSDL file, which imports an XML schema. Rather than include both listings here, I'll present the representative sections.

First, the `portType` element from the WSDL file contains all the operations. Here's the listing for the `add` operation:

```
  <portType name="Calculator">
    <operation name="add">
      <input message="tns:add"/>
      <output message="tns:addResponse"/>
    </operation>
...
  </portType>
```

The other operations are similar. Following the normal conventions, the input and output messages delegate to `message` elements, each of which has a `<part>` child with an attribute named `element`. Those then lead to the `add` and `addResponse` elements in the schema, which are shown next:

```
<xs:element name="add" type="tns:add"/>

<xs:element name="addResponse" type="tns:addResponse"/>
...
```

```
<xs:complexType name="add">
    <xs:sequence>
        <xs:element name="x" type="xs:double"/>
        <xs:element name="y" type="xs:double"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="addResponse">
    <xs:sequence>
        <xs:element name="sum" type="xs:double"/>
    </xs:sequence>
</xs:complexType>
```

The `@WebResult` and `@WebParam` annotations set the names of the associated elements to `x`, `y`, and `sum`, as shown.

Incidentally, the tool doesn't know where I plan to deploy the service, so the `service` element in the WSDL file only has a placeholder in it:

```
<service name="CalculatorService">
    <port name="CalculatorPort" binding="tns:CalculatorPortBinding">
        <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
</service>
```

One additional observation before I move on—for some strange reason, the `wsgen` tool is designed to work with the SIB, not the SEI. Every other distributed application specification, from JAX-RPC to RMI to even CORBA, expects a clean separation between the endpoint interface and the implementation bean. JAX-WS doesn't require this, and it's simpler if I don't have it. Life gets more complicated if I do what's normally considered good design and separate the interface from the implementation, but I'll have to come back to that later when I show a more sophisticated service.

The next step is to deploy the service, because the client stubs are generated using `wsimport` aimed at the deployed WSDL file. Normally that would require creating a web application with a special deployment descriptor called webservices.xml, but the JAX-WS API affords me a simplification for testing purposes. Instead of a web application, I can use the `javax.xml.ws.Endpoint` class from the standard library. Publishing the service can be done as a one-liner in a regular Java application, as shown in the next listing.

Listing C.4 Publishing the `Calculator` service for testing

```
import javax.xml.ws.Endpoint;

public class CalculatorServer {
    public static void main(String[] args) {
        Endpoint.publish("http://localhost:1234/calc", new Calculator());
        System.out.println(
            "Calculator ready to receive requests on port 1234...");
    }
}
```

The static `Endpoint.publish` method exposes the web service at the specified URL using a single-threaded HTTP server. That's not sufficient for production, but it's good enough for testing purposes. To prove that everything is working, I'll write a test based on `GroovyTestCase`. That means I need a client, which is generated the same way the Global Weather client was generated previously:

```
c:\> wsimport -d bin -s src -keep http://localhost:1234/calc?WSDL
```

This generates the stubs that do the plumbing necessary to generate a SOAP message, transmit it to the service, receive a response, and convert it back to a double.

Looking at the WSDL file, note that the `portType` is called `Calculator`, the service is called `CalculatorService`, and the contained port is called `CalculatorPort`. The following listing shows a `GroovyTestCase` to test the calculator.

Listing C.5 A `GroovyTestCase` to evaluate the calculator web service

```
import jag.calc.service.*
import groovy.util.GroovyTestCase;

class CalculatorTest extends GroovyTestCase {
    Calculator calc
                                                    Runs before
    void setUp() {                              <-- each test
        super.setUp();                                            Initializes
        calc = new CalculatorService().calculatorPort       <--   the stub
    }

    void testAdd() { assertEquals 5, calc.add(2,3), 0.0001 }
    void testSubtract() { assertEquals 2, calc.subtract(5, 3), 0.0001 }
    void testMultiply() { assertEquals 6, calc.multiply(2, 3), 0.0001 }
    void testDivide() { assertEquals 0.25, calc.divide(1, 4), 0.0001 }
}
```

The `setUp` method runs before each test. The code in the `setUp` method reinstantiates the stub each time to ensure that the test cases do not conflict with each other. Each test uses the `assertEquals` method with three arguments: the first is the right answer, the second is the test, and the third is a precision, needed because I'm comparing doubles. All of the tests pass, as expected.

This section showed how the `wsgen` tool is used with POJO annotations to generate a WSDL file, and the `wsimport` tool uses the resulting WSDL to generate client stubs. The service implementation is in Java, as are the client stubs, but at least the test case is in Groovy. The question now is, what happens if the SIB is implemented in Groovy? Does that affect the `wsgen` tool at all?

C.3.2 *Implementing the SIB in Groovy*

The Groovy SIB, shown in the next listing, is somewhat simpler. First, I don't have to assign data types to the method parameters. Second, in Groovy a closure returns the last computed value, so I don't need any formal return statements, either.

Listing C.6 A Groovy implementation of the calculator web service

```
import javax.jws.WebService;

@WebService
class GroovyCalculator {
    double add(x,y) { x + y }
    double subtract(x,y) { x - y }
    double multiply(x,y) { x*y }
    double divide(x,y) { x/y }
}
```

I could add `@WebResult` and `@WebParam` annotations as before, but I want to make a point first. Running `wsgen` on the compiled version of `GroovyCalculator` already leads to problems:

```
C:\> wsgen -d bin -s src -keep -r resources \
    -cp bin jag.calc.service.GroovyCalculator
Exception in thread main:
    java.lang.NoClassDefFoundError: groovy/lang/GroovyObject
...
```

The `wsgen` tool is thrown by the lack of a Groovy `library` class. That's probably not a big surprise. Groovy classes extend `groovy.lang.GroovyObject`, not `java.lang.Object` as Java classes do. The first attempt to solve the problem is to add the groovy-all JAR file (from the embeddable subdirectory of my Groovy installation) to the classpath and try again.

Rather than show the somewhat dazzling exception stack trace that results, let me explain the next problem. The `@WebService` annotation tells `wsgen` to expose every public method as a web service. Unfortunately, when the tool is applied to a Groovy SIB, one of the methods it finds is `getMetaClass`, and it has no idea what to do with that.

Every Groovy class has an associated metaclass, which is the key to Groovy metaprogramming. Metaprogramming is how you create Groovy builders, domain-specific languages, and lots more. Here, though, it's an inconvenience. I certainly don't want the `getMetaClass` method exposed as a web service, so how do I stop that from happening?

The answer is to do what I wanted to do all along, which is to separate the interface from the implementation. Rather than let the tool generate an SEI, I'll provide one and tell the tool about it via the `@WebService` annotation. The tool then knows that only the methods defined in the SEI are supposed to be exposed as web services, which solves the problem.

The new SEI is shown here:

```
@WebService
public interface Calculator {
    @WebResult(name="sum")
    double add(
        @WebParam(name="x") double x,
        @WebParam(name="y") double y);
    ...
}
```

To play nicely with the tool, the SEI is written in Java, but if you try it (which I'll do later) you'll find that the SEI could just as easily have been written in Groovy. Because I'm using Java this time, though, I have to supply data types for the method arguments. All of the annotations that were previously on the SIB are now on the SEI.

With all those annotations now on the SEI, the Groovy SIB becomes very simple:

```
@WebService(endpointInterface="jag.calc.service.Calculator")
class GroovyCalculator implements Calculator {
    double add(double x, double y) { x + y }
    double subtract(double x, double y) { x - y }
    double multiply(double x, double y) { x*y }
    double divide(double x, double y) { x/y }
}
```

The `@WebService` annotation is now there to tell `wsgen` that this SIB implements the listed `Calculator` endpoint interface.

This version of the calculator, with its Java SEI and Groovy SIB, works the same as the pure Java version, but there are a few changes. First, here's a Groovy script to run the new calculator:

```
import javax.xml.ws.Endpoint

Endpoint.publish 'http://localhost:1234/calc', new GroovyCalculator()
println 'Calculator ready to receive requests on port 1234...'
```

There's nothing unusual there, other than instantiating the Groovy SIB rather than the Java one. The analogous test case is basically the same, too, but there's one annoying change that will become apparent:

```
class GroovyCalculatorTest extends GroovyTestCase {
    Calculator calc

    protected void setUp() throws Exception {
        super.setUp()
        calc = new GroovyCalculatorService().groovyCalculatorPort
    }
...
}
```

I may have introduced an interface, but the name of the SIB is embedded in the name of the service and the port. Part of the appeal of separating the interface from the implementation is that I avoid exposing details of the implementation to the outside client, and the name of the implementation class is certainly one of those details. Although there are multiple ways to fix that problem, a particularly interesting one is to use the `@Delegate` AST transformation discussed in chapter 2.

C.3.3 Splitting the interface from the implementation

As the saying goes, every problem in computer science is solved by adding a layer of indirection. If my problem is that the name of the SIB becomes part of the service and port elements, I'll make an acceptable name for the SIB and delegate its implementation to another class.

That's a perfect use case for Groovy's @Delegate AST transformation. I annotate a field inside a wrapper class with @Delegate, and all methods in the field's class are exposed by the wrapper.

For the calculator, let's assume I have the same Calculator interface as before, but instead the implementation bean looks like the following:

```
package jag.calc.service

import groovy.lang.Delegate;
import javax.jws.WebService;

@WebService(endpointInterface="jag.calc.service.Calculator")
class GenericCalculator {
    @Delegate Calculator calc
}
```

All the Calculator methods are implemented by the contained delegate, here called calc. Now it isn't even necessary to make GenericCalculator implement the Calculator interface, though it does do so here because it helps my IDE understand what's going on. In many IDEs, if you say that GenericCalculator implements Calculator, the IDE will not realize that the @Delegate annotation passes all the methods through. The GenericCalculator uses duck typing to invoke the delegated methods, so it doesn't need the delegate to implement Calculator.

If I now implement Calculator in two ways, once as a Java class that I call JavaCalculator, and once as a Groovy class I call GroovyCalculator, then I can choose which implementation to use for the delegate at runtime:

```
import javax.xml.ws.Endpoint;

Calculator calc = new GenericCalculator(calc:new JavaCalculator())
Endpoint.publish "http://localhost:1234/calc", calc
println 'Calculator ready to receive requests on port 1234...'
```

In this Groovy deployment I need to assign the calc parameter inside the GenericCalculator to either the Java implementation or the Groovy implementation. Here I chose the Java implementation. Either way, though, (1) as far as the tool is concerned the actual SIB is a Groovy class called GenericCalculator, and (2) the client has no idea which actual implementation class is being used. The corresponding GroovyTestCase is the same as the previous one, with the only change being the instantiation of the stub:

```
calc = new GenericCalculatorService().genericCalculatorPort
```

Note that now it has the name `GenericCalculator` embedded in it, rather than either actual implementation class. The delegate provides the needed layer of indirection.

The World's Slowest Calculator is a useful example because it illustrates the concepts and the simple tools, without getting into complex type-mapping issues. To be honest, though, if a framework can't handle this example, it can't handle anything. Now I'll show what happens if I have object-based arguments and/or return types in the operations.

C.3.4 A nontrivial domain model

The next web service I'll build will use complex types as arguments and return values from the operations. To illustrate the differences, I'll start by using POJOs and then introduce POGOs.

The new service is called Potter's Potions. If you send Potter's Potions a cauldron full of ingredients, the service's wizards will brew the potion for you, guaranteeing a certain level of effectiveness.[2]

As before, I'll build the web service from the bottom up, first developing the classes and then generating everything else. The previous section showed that if Groovy is involved, the `wsgen` tool works best when there's a clean separation between the SEI and the SIB. With that in mind, here's the `Wizard` interface, this time implemented in Groovy:

```
@WebService
interface Wizard {
    @WebResult(name="potion")
    Potion brewPotion(@WebParam(name="cauldron") Cauldron c);
}
```

The interface has only a single method, `brewPotion`, which takes a `Cauldron` as an argument and returns a `Potion`. The `@WebService`, `@WebResult`, and `@WebParam` annotations have been added as usual. The `Cauldron` class shows that the domain model also includes an `Ingredient` class, as shown in the next listing.

Listing C.7 The `Cauldron` POJO, which wraps the list of ingredients

```
public class Cauldron {
    @XmlElementWrapper(name="ingredients")
    @XmlElement(name="ingredient")
    private List<Ingredient> ingredients = new ArrayList<Ingredient>();

    public void addIngredient(Ingredient i) {
        ingredients.add(i);
    }

    public void removeIngredient(Ingredient i) {
        ingredients.remove(i);
    }
```

JAXB annotations for wrapper element

[2] Harry isn't directly involved. He provided financial support and lent his name for marketing purposes.

```
    public Cauldron leftShift(Ingredient i) {
        addIngredient(i);
        return this;
    }
}
```

Used by << operator in Groovy scripts

The XmlElementWrapper and XmlElement annotations make the SOAP messages more intuitive. They imply that an instance of the Cauldron class represented in XML will contain a root element called <ingredients>, which in turn will have a set of <ingredient> children.

The other interesting aspect of the Cauldron POJO is that by implementing the leftShift method, operator overloading works in the Java class. The << operator can be applied to the Cauldron in a Groovy script to add new Ingredients. For example, given this POJO I can write (in Groovy)

```
Cauldron c = new Cauldron()
c << new Ingredient(...)
c << new Ingredient(...)
```

Actually, because the method returns the this reference, I can chain those calls, as I'll show in a test class later. Speaking of the Ingredient class, it also is a simple POJO, as shown in the following listing.

Listing C.8 The Ingredient POJO, used in the Cauldron

```
public class Ingredient {
    private String name;
    private double amount;
    private String units;

    public Ingredient() {}

    public Ingredient(String name, double amount, String units) {
        this.name = name;
        this.amount = amount;
        this.units = units;
    }
...
}
```

The rest of the Ingredient class consists of getters and setters for the name, amount, and units fields. The other POJO in the system is for the Potion itself, which is shown in the next listing.

Listing C.9 The Potion POJO

```
public class Potion {
    private String name;
    private String effect;
    private Date expiration;
    private double probability;

    private List<Ingredient> ingredients;
    public Potion() {}

    public Potion(String name, String effect, Date expiration) {
        this.name = name;
        this.effect = effect;
        this.expiration = expiration;
    }
...
}
```

Once again, the rest of the class is getters and setters for the fields shown. The idea behind the `Potion` class is that each one has a `name` and an `effect`, as well as an `expiration` date and a `probability` that the potion will work. Better wizards will produce potions with a higher likelihood of being effective. Better wizards cost more, of course.

The `Calculator` example also demonstrated how to use the `@Delegate` annotation to keep the name of the implementation class out of the WSDL file. Therefore, here's the generic Hogwarts wizard (in Groovy):

```
import groovy.lang.Delegate;
import javax.jws.WebService;

@WebService(endpointInterface="jag.pp.service.Wizard")
class HogwartsWizard {
    @Delegate Wizard wizard
}
```

The `HogwartsWizard` class includes the `endpointInterface` attribute on the `Web-Service` annotation, which is necessary for the `wsgen` tool to work, but doesn't explicitly have an `implements` clause on the class, to avoid IDE issues. The methods are available, but because they're supplied by the AST transformation, I'll use duck typing to take advantage of them.

The following listing shows a Groovy `Wizard` implementation.

Listing C.10 The `Granger` implementation, which is a high-quality `Wizard`

```
import jag.pp.entities.Cauldron;
import jag.pp.entities.Potion;

class Granger implements Wizard {

    Potion brewPotion(Cauldron c) {
        Potion p = new Potion(name:'test',effect:'none',
            expiration:new Date() + 1, probability:0.9);
        p.ingredients = c.ingredients;
        return p
    }
}
```

Just for testing, the brewed potion is called `test`, expires tomorrow, and has no effect, though there's a 90-percent likelihood of that. The `Granger` wizard is implemented in Groovy.

Starting the server can now be done with the following script:

```
Wizard wiz = new HogwartsWizard(wizard:new Granger())
Endpoint e = Endpoint.publish("http://localhost:1234/wizard", wiz)
println "Wizard available to receive requests..."
```

The next listing shows a client to access this service. This time, to show another possibility, the client uses a dynamic proxy rather than the static stubs. The implementation uses the `URL`, `Service`, and `QName` classes from Java web service libraries. Only the `Wizard` interface itself is referenced, and no other generated classes.

Listing C.11 Client

```
Service service = Service.create(
    new URL('http://localhost:1234/wizard?WSDL'),
    new QName('http://service.pp.jag/','HogwartsWizardService'))
Wizard w = service.getPort(
    new QName('http://service.pp.jag/','HogwartsWizardPort'),
    Wizard.class)

Cauldron c = new Cauldron()
c << new Ingredient(name:'sopophorous bean',amount:1,units:'bean')
c << new Ingredient(name:'gillyleaf',amount:1,units:'handful')
println c
Potion p = w.brewPotion(c)
println p
```

The left-shift operator is used to add `Ingredient` instances to the `Cauldron`. It's time to show test cases to prove everything works. Then I'll replace the POJOs with POGOs and show what happens.

SPOCK TESTS FOR POTTER'S POTIONS

In chapter 6, I discussed the Spock testing framework. Spock tests are written in Groovy but can be used to test purely Java or mixed Java and Groovy systems. Here I'll show a test for the `Cauldron` implementation, which will verify that the operator overloading works, and a test for the web service itself.

First, the next listing shows the `Cauldron` tests.

Listing C.12 Spock specification for the `Cauldron`

```
class CauldronTests extends Specification {
    Cauldron c

    def setup() { c = new Cauldron() }

    def "cauldron starts off empty"() {
        expect: c.ingredients.size() == 0
    }

    def "add ingredient increases size by 1"() {
        when: c.addIngredient(new Ingredient())
        then: c.ingredients.size() == old(c.ingredients.size()) + 1
    }
    def "remove ingredient decreases size by 1"() {
        given:
        Ingredient i = new Ingredient()
        c.addIngredient(i)
        c.addIngredient(new Ingredient())

        when: c.removeIngredient(i)
        then: c.ingredients.size() == old(c.ingredients.size()) - 1
    }

    def "left shift operator adds ingredient"() {
        when: c << new Ingredient()
        then: c.ingredients.size() == old(c.ingredients.size()) + 1
    }

    def "left shift operator can be chained"() {
        when: c << new Ingredient(name:'a') << new Ingredient(name:'b')
        then: c.ingredients*.name.containsAll 'a','b'
    }
}
```

The `addIngredient` and `removeIngredient` methods work as expected. It's interesting to see that using the `<<` operator in a Groovy test automatically invokes the left-shift operation in the Java class, and, as implemented, allows chaining so it can be invoked repeatedly.

For the test of the web service itself, see the following listing.

Listing C.13 Spock specification for the Potter's Potions web service

```
class HogwartsWizardTest extends Specification {
    @Shared Endpoint e
    Wizard w
    Cauldron c

    def setupSpec() {
        e = Endpoint.publish("http://localhost:1234/wizard",
            new HogwartsWizard(wizard:new Granger()))
    }

    def setup() {
        Service service = Service.create(
            new URL('http://localhost:1234/wizard?WSDL'),
            new QName('http://service.pp.jag/','HogwartsWizardService'))
        w = service.getPort(
            new QName('http://service.pp.jag/','HogwartsWizardPort'),
            Wizard.class)
        c = new Cauldron()
        c << new Ingredient(
           name:'sopophorous bean',amount:1,units:'bean') <<
           new Ingredient(name:'gillyleaf',amount:1,units:'handful')
    }

    def "wizard exposed as a web service"() {
        expect: w
    }
```

Shared endpoint and its initialization

Dynamic client

Potential trap avoided!

```
    def "brew a potion"() {
        when: Potion p = w.brewPotion(c)
        then:
        p != null
        p.ingredients*.name.containsAll 'sopophorous bean','gillyleaf'
    }

    def cleanupSpec() {
        e?.stop()
    }
}
```

Collect ingredient names for validation

Shut down web service

In this case I use the `@Shared` annotation, which publishes the endpoint and makes it available to all the tests. By keeping a shared reference to the `Endpoint` itself, I can later shut it down when all the tests are finished. The service is published in a `setupSpec` method, analogous to the `@BeforeClass` annotation in JUnit 4, which means the `setupSpec` method runs only once for the entire test. Its match is the `cleanupSpec` method (like the `@AfterClass` annotation in JUnit 4), which shuts down the service. The `setup` method, here used to initialize the cauldron, is run before each test. Note that when I stop the service I use the wicked cool[3] safe dereference operator `?.` in case the start-up was unsuccessful.

By the way, I avoided a potential trap by putting the `<<` operator at the end of the current line, rather than at the beginning of the next line. Although I know from the previous test that the operator can be chained, that's not the issue here. Java uses semicolons to terminate statements. In Groovy those are optional, so I need to make sure that Groovy understands there's more information coming on the next line. Without the operator, Groovy sees the line as a complete statement and then doesn't know what to do with the following line. By placing the `<<` operator at the end of one line, Groovy knows the statement isn't over yet and reads on.

HELP THE GROOVY PARSER Always make it clear to the Groovy parser when a statement is not finished.

The test accesses the list of ingredients in the resulting potion, collects their names into another list, and then uses the GDK method `containsAll` to check that the supplied values appear somewhere in the list.

Using POGOs in the domain model

The Potter's Potions web service works using POJOs. At the moment, the only Groovy classes are the SIB wrapper (`HogwartsWizard`, which uses the `@Delegate` AST) and the `Wizard` implementation itself, `Granger`. What if I replace one of the POJOs with a POGO?

Here's a Groovy version of the `Ingredient` class. It has the same functionality as the associated POJO but lacks semicolons, getter and setter methods, and public and private modifiers:

[3] Sorry for the colloquialism, but sometimes the safe dereference operator in Groovy impresses Java developers as much as the most elegant metaprogramming example.

```
class Ingredient {
    String name
    double amount
    String units
}
```

If I run the wsgen tool on this, I immediately run into a problem, and it's similar to the problem I had earlier with the Groovy service. The wsgen task asks JAXB to figure out how to serialize instances of this class. That causes the JAXB processor to search for all the properties in Ingredient. Because Ingredient is written in Groovy, it contains a method called getMetaClass, implying that there's a metaClass property JAXB needs to manage. JAXB has no idea what to do with that, so the process fails.

With the Groovy service, I found that if I enforce a clean separation between the interface and the implementation I can still use the same tools. Here the solution is even easier. Following is the updated class, which works just fine:

```
import javax.xml.bind.annotation.XmlAccessType
import javax.xml.bind.annotation.XmlAccessorType

@XmlAccessorType(XmlAccessType.FIELD)          <-- Go directly to
class Ingredient {                                  the private fields
    String name
    double amount
    String units
}
```

The annotations tell JAXB to use the private fields for the properties, rather than going through the getter methods. Therefore, it only looks at the fields I've exposed, and the getMetaClass method no longer enters into it. Now everything works.

FIELD ACCESS To use POGOs in SOAP-based web services, go directly to the fields.

Using Groovy in SOAP-based web services

To use Groovy classes in a SOAP-based web service

- Define an SEI to go with the SIB, using Groovy for either or both.
- Annotate all domain classes to require field access for the properties.

The choice of whether or not to use Groovy in each case is dependent on the actual problem. For example, the Global Weather service returned an XML block, so Groovy was a natural for the client. On the server side, if the service itself is simpler when implemented in Groovy (and almost every service will be), then go right ahead, as long as these two rules are followed.

One of the features of SOAP and WSDL is that the tools hide the actual XML. At no point in the examples presented so far did I need to parse or generate any actual XML messages. There's one situation, however, in which XML is involved, even using the tools, and that's when pre- or postprocessing of the messages is necessary. That's the subject of the next section.

C.4 Processing XML messages with web service handlers

One of the main benefits of using the JAX-WS infrastructure is that neither the client nor the service ever has to deal directly with the SOAP messages. When the client called the potions service, no XML processing was exposed on either side. Using JAX-WS virtually eliminates the need to deal with the Java API for XML Processing (JAXP) or the SOAP with Attachments API for Java (as in parsley, SAAJ, rosemary, and thyme).

Still, there are times when I have to deal with the actual XML. For those times, the JAX-WS infrastructure provides handlers, which process any messages passing through them.

The web services architecture presumes that a SOAP message travels through a series of intermediaries on its way from the client to the service and back. These intermediaries serve a variety of roles, from routing to encryption to logging to application-specific tasks.

In JAX-WS, handlers come in two forms: logical handlers and protocol handlers. Logical handlers have access only to the body of SOAP messages, whereas protocol handlers work with the entire message. Multiple handlers can be added to a web service, forming a chain of responsibility that processes messages on their way to the service and back. Figure C.5 illustrates a handler chain on a service.

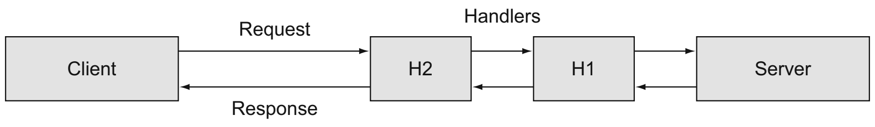


Figure C.5 Handlers H1 and H2 intercept requests and responses from the client to the server and can modify the message both on the way in and on the way out again.

As usual, I'm interested more in the sweet spots where Groovy can help Java, rather than looking for ways to do everything in Groovy. In the case of handlers, it turns out to be easiest to define the handler in Java but let it delegate any XML processing to Groovy. Defining a handler requires implementing an interface based on generic types, and although that can be done in Groovy, sometimes it's easier to do so in Java because the Java tools prefer Java interfaces.

The following example adds a handler to the Potter's Potions web service. The idea is to check the incoming `Cauldron` for ingredients that might be used to practice Dark Arts and, if any are found, to add a disclaimer message to the outgoing message. The process is as follows:

1. Write a Java class that implements the `SOAPHandler` interface using the `SOAPMessageContext` generic type.
2. In the `handleMessage` method, determine whether the message is incoming or outgoing.
3. Pass the message to a Groovy class for further processing.

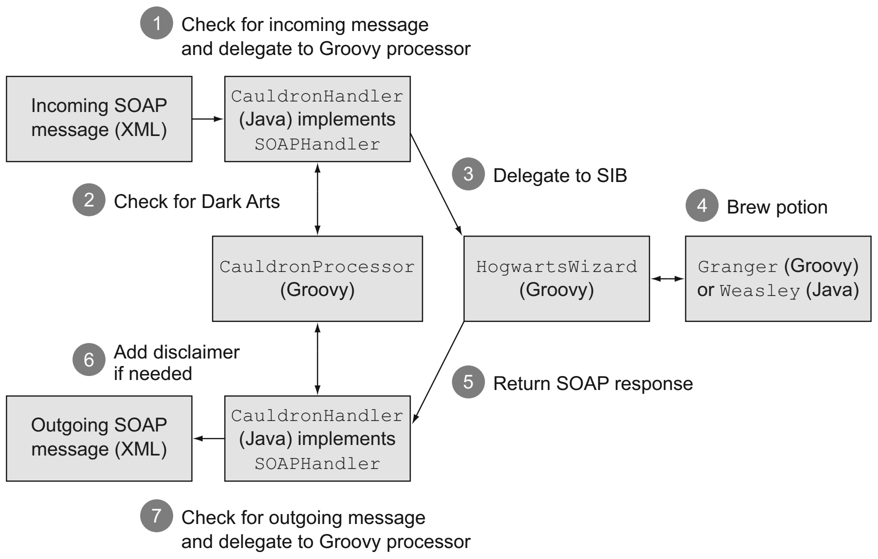

1 Check for incoming message and delegate to Groovy processor
Incoming SOAP message (XML)
CauldronHandler (Java) implements SOAPHandler
2 Check for Dark Arts
3 Delegate to SIB
4 Brew potion
CauldronProcessor (Groovy)
HogwartsWizard (Groovy)
Granger (Groovy) or Weasley (Java)
6 Add disclaimer if needed
5 Return SOAP response
Outgoing SOAP message (XML)
CauldronHandler (Java) implements SOAPHandler
7 Check for outgoing message and delegate to Groovy processor

Figure C.6 Handling XML is best done by delegating to Groovy, but the `SOAPHandler` interface is most easily implemented in Java. Implement the Handler in Java, but delegate the actual XML processing to Groovy to get the best of both worlds.

4 In the Groovy processor, if the message is on the way in, scan the body for potentially dangerous ingredients. If it's on the way out, and evil has been found, add an appropriate disclaimer.

Each step is demonstrated in the subsections that follow. The overall process is illustrated in figure C.6.

C.4.1 SOAP handler in Java

The following listing shows a Java handler for the Potter's Potions service, called `CauldronHandler`. The real work is done in the `handleMessage` method.

Listing C.14 A Java handler to check for evil

```
public class CauldronHandler implements SOAPHandler<SOAPMessageContext> {

    private CauldronProcessor cp = new CauldronProcessor();
    private boolean darkArtsDetected;

    @Override
    public boolean handleMessage(SOAPMessageContext context) {
        boolean response = (Boolean) context.get(
                MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        try {
            cp.setBody(context.getMessage().getSOAPBody());
            if (!response) {
                darkArtsDetected = cp.detectDarkArts();
                cp.printInfo();
```

Groovy delegate for XML processing

Shared between request and response

Block for incoming requests

```
                } else {                                    Block for outgoing
                    if (darkArtsDetected) {                 responses
                        cp.addDisclaimer();
                    }
                }
            } catch (SOAPException e) {
                e.printStackTrace();
            }
            return true;
        }

        public boolean handleFault(SOAPMessageContext context) {
            return false;
        }

        public void close(MessageContext context) {}
        public Set<QName> getHeaders() { return null; }
    }
```

This is typical Java, meaning it has a lot of ceremony surrounding the essence. The class implements the `SOAPHandler` interface, which takes the generic type `SOAPMessage-Context`. The interface contains four methods: `handleMessage`, which is invoked on each message; `handleFault`, which is called if something goes wrong; `getHeaders`, which retrieves any header blocks that can be processed by this handler; and `close`, which is self-explanatory.

DISTINGUISHING INCOMING FROM OUTGOING MESSAGES

The `handleMessage` method is called once with the inbound request and once with the outbound response. On the way in, I check for ingredients that can be used for Dark Arts potions, and if I find any I set the Boolean variable `darkArtsDetected` to `true`.[4] The company can't be associated with anything like that, so if the detector found anything suspicious, a disclaimer message is added to the response. The standard idiom for handlers is to check the `MESSAGE_OUTBOUND_PROPERTY` to determine which way the message is traveling. The actual message is in XML, so I use a Groovy class to work with it.

C.4.2 Using Groovy for XML manipulation

`SOAPMessage` is part of the SAAJ API, which is easier to work with than a standard DOM tree but still tedious. I need some way to get the XML data out of the SOAP message and into something that Groovy can process easily. Normally when I work with XML-based formats in Groovy I use an `XmlParser` or an `XmlSlurper`. In this particular case, however, something different is available. The SOAP message is composed of a `SOAP-Part`, which in turn is made up of a `SOAPHeader` and a `SOAPBody`. The key observation is that `SOAPPart` implements `org.w3c.dom.Node`, whereas `SOAPHeader` and `SOAPBody` implement both that interface as well as `org.w3c.dom.Element`.

[4] I'm assuming here that a new handler is created for each request, so the Boolean property will have a single value for the request going in and the response coming out. If the app server generates a single handler for all requests, this approach will have to be changed.

As part of its metaprogramming capabilities, Groovy includes a construct known as a *category*. A category is a special type of class, composed of all static methods, that's used to add new methods to an existing class. Rather than add the methods to a class's metaclass, however, with a category the additional methods are only available inside a `use` block. The result is a set of additional methods that are restricted to a controlled environment.

CATEGORIES Groovy categories add methods to existing classes inside a `use` block only.

One of the categories built into the Groovy API is `groovy.xml.dom.DOMCategory`. This category adds methods to both the `Element` and `Node` interfaces described previously. Because the `SOAPBody` class implements `Element` and `Node`, I can use these additional methods to make processing easier.

In the `CauldronHandler` class, I extract the SOAP body from the incoming message and put it into the (Groovy) `CauldronProcessor` class:

```
cp.setBody(context.getMessage().getSOAPBody());
```

Then I use the `CauldronProcessor` to get the job done. The following listing shows the `CauldronProcessor`.

Listing C.15 A Groovy class for working with the body of a SOAP message

```
class CauldronProcessor {
    Element body

    def printInfo() {
        use (DOMCategory) {
            println "There are ${body.'**'.size()} nodes"
            println body.list()
        }
    }

    boolean detectDarkArts() {
        use (DOMCategory) {
            return body.'**'.any { it =~ /.*unicorn.*/ }
        }
    }

    def addDisclaimer() {
        use (DOMCategory) {
            def e = body.'**'.find { it.name() == 'effect' }
            e.replaceNode {
                effect "No warranty expressed or implied"
            }
        }
    }
}
```

Element to hold SOAP body

Use block for DOMCategory

Search using regular expression

Modify tree

In the Groovy class, a reference of type `Element` holds the body of the SOAP message. In the Java class, calling the `setBody` method puts the body into this object. Like all

Groovy attributes, the setBody method is generated automatically by declaring the body property.

The DOMCategory adds a list method to Element. The list method prints the complete tree in a nicely formatted output. Although normally I wouldn't print to the console when running a web service, it's useful for demonstration purposes, and I can always use a logger later instead.

I also use the syntax body.'**' in each of the body processing methods. The ** symbol is shorthand for a depth-first search among all the descendent nodes and returns all of them in a collection. In the printInfo method, I invoke the size method to see how many there are in all. In the detectDarkArts method, I search the resulting list to see if any of the contained elements (which include all the supplied ingredients in the cauldron) have the word unicorn in them. Nothing good can come of that, so if that's found, the method returns true and the darkArtsDetected Boolean in the Java class is set to true.

If Dark Arts are detected, I want to change the properties of the returned response. I could add a new element with a warning, but the response message is eventually going to be converted into a Potion, one of the domain classes. Therefore, it's easy to modify the effect child of the potion, replacing its normal value with a disclaimer.

Putting the handler into the system requires two additional steps. First, I need to prepare an XML file declaring the handler, as shown next:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<handler-chains
    xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-class>jag.pp.handlers.CauldronHandler</handler-class>
        </handler>
    </handler-chain>
</handler-chains>
```

This file, handler-chain.xml, declares that I have a chain of handlers consisting of a single handler. The current handler is the Java class that implements the SOAP-Handler interface.

Now that I have the configuration file, I also need to tell the web service to use it. Fortunately, that can be done with a single annotation applied to the SIB, which is reproduced here in its entirety:

```
@WebService(endpointInterface="jag.pp.service.Wizard")
@HandlerChain(file="handler-chain.xml")
class HogwartsWizard {
    @Delegate Wizard wizard
}
```

The HandlerChain annotation tells the JAX-WS processor where to find the handler configuration file for this service implementation bean. Now when I run the service all requests will go through the handler on the way to the service, and all responses will pass through it on the way back to the client.

Back in listing C.13, I presented a Spock specification for testing the web service. To test the handler I add a single new test, as shown next:

```
def "check for dark arts disclaimer"() {
    setup:
    c << new Ingredient(name:'unicorn blood',amount:2,units:'smidges')

    when: Potion p = w.brewPotion(c)
    then: p.effect == 'No warranty expressed or implied'
}
```

If I add a smidge of unicorn blood to the ingredients in the cauldron, the Dark Arts detector sees it and changes the value of the potion's `effect` property to the disclaimer.

One of Groovy's greatest advantages over Java is its ease of working with XML. In this section, a Java handler delegated SOAP message processing to a Groovy class, which used a category to easily process the message. The Java class connected the Groovy processor to the existing infrastructure. The Groovy class made the XML manipulation easy. The combination makes writing web service handlers a simple task.

What about GroovyWS?

The Groovy ecosystem has spawned a large number of projects based on the language, with many more coming all the time. One project, called GroovyWS, is aimed at the same problems discussed in this appendix.

GroovyWS is an elegant attempt to remove all Java from the system, though it does depend on an underlying JAX-WS implementation. You can download a complete GroovyWS implementation, which comes with Apache CXF as its Java web services engine.

For details about GroovyWS, including an installation guide and some examples, look for the GroovyWS link on the main Groovy page. I've heard good things about the project from some dedicated users. Still, in a book about Java and Groovy integration it feels a bit outside the book's scope. To be honest, I've never needed it to accomplish my goals.

C.5 Gradle tasks for wsimport and wsgen

The previous sections discussed how to use a combination of Groovy and Java to both access and implement web services. Because the goal was to highlight how the technologies worked, I didn't spend time showing the details of the build process. In each case, the `wsimport` and `wsgen` tools were run from the command line. Part of their appeal is that they're included in the standard JDK 1.6 installation. I can include running the tasks as part of a Gradle build, however, if I declare the proper dependencies. This also gives me an opportunity to illustrate a conditional task in a Gradle build file.

Both the `wsimport` and `wsgen` commands have corresponding Ant tasks available. The key here is to use those tasks, making sure they run at the proper stage of the build and that they only run when necessary. In my case, I have a separate project for the client and for the server, so there are two similar build files involved.

For `wsimport`, I want to run the command before the `compileJava` task, because I'm generating Java stubs that are relied upon by the client. The JAR dependencies containing the necessary Ant tasks are found in repositories other than Maven central, so I have to declare them as well. Here's the new `repositories` section of the build files:

```
repositories {
    mavenCentral()
    mavenRepo urls:["http://download.java.net/maven/1,
        "http://download.java.net/maven/2"]
}
```

Now Gradle will check the standard Maven repository first and then search the listed Maven 1 and Maven 2 download areas for any unresolved dependencies.

Because the Ant tasks are external to my projects, I declare a `configurations` element, here called `jaxws`:

```
configurations {
    jaxws
}
```

The `jaxws` element gives a name that can be used both in the tasks, in order to put the required JARs in the classpath, and in the `dependencies` section. Here's the updated `dependencies` section:

```
dependencies {
    compile group:'org.codehaus.groovy', name:'groovy-all', version:'2.1.6'
    jaxws 'com.sun.xml.ws:jaxws-tools:2.1.4'

    testCompile "org.spockframework:spock-core:$spockVersion"
}
```

The `jaxws` label identifies the `tools` dependency for both the `wsimport` and `wsgen` Ant tasks. Next is the `wsgen` task, along with lines specifying when it should run, as shown in the next listing.

Listing C.16 The `wsgen` task in Gradle, based on its Ant task

```
task wsgen(dependsOn: compileGroovy)  {          <-- Compile SIB first
    doLast{
        ant {
            taskdef(name:'wsgen',
                classname:'com.sun.tools.ws.ant.WsGen',
                classpath:configurations.jaxws.asPath)      <-- Put JARs in classpath
            wsgen(keep:true,                          <-- wsgen command-line args
                destdir: 'bin',
                sourcedestdir:'src',
                resourcedestdir:'resources',
                genwsdl:'true',
                classpath:sourceSets.main.classes.asPath,
                sei:'mjg.pp.service.HogwartsWizard')
        }
    }
}
wsgen.onlyIf { !(new File('src/mjg/pp/service/jaxws')).exists() }   <-- Conditional execution
classes.dependsOn wsgen
```

The wsgen task depends on the compileGroovy task, because the wsgen tool operates on the compiled SIB. The rest of the parameters are analogous to the same command-line parameters used earlier, including the destination directories for the generated WSDL and XML schema files. Also note how the classpath of the wsgen task refers to JARs from the configuration element. Finally, the Gradle onlyIf method of the Gradle task takes a closure and only runs if the closure returns true. In this case, I check to see if the task has already been run by looking for the expected package on the disk. If it isn't there, Gradle runs the task.

The Gradle task for wsimport is quite similar. Because the Potter's Potions application uses dynamic proxies, the code in the next listing is for the Global Weather client instead.

Listing C.17 The Gradle task for `wsimport`, again based on the Ant task

```
task wsimport(dependsOn: processResources)  {          <-- When to run
    doLast{
        ant {
            taskdef(name:'wsimport',
                classname:'com.sun.tools.ws.ant.WsImport',
                classpath:configurations.jaxws.asPath)     <-- Classpath for Ant tasks
            wsimport(keep:true,                             <-- Command-line args
                destdir: sourceSets.main.classesDir,
                sourcedestdir:'src',
            wsdl:'http://www.webservicex.net/GlobalWeather.asmx?wsdl')
        }
    }
}
wsimport.onlyIf { !(new File('src/net/webservicex')).exists() }   <-- Conditional execution
compileJava.dependsOn(wsimport)
```

As with the wsgen task, I choose when to run the task, use the jaxws configuration to set the classpath, supply the command-line arguments, and only run if the stubs have not already been generated. Once again, Gradle makes customizing the build straightforward—a great advantage over more traditional build tools like Ant and Maven.

JAVA

Making Java Groovy

Kenneth A. Kousen

You don't need the full force of Java when you're writing a build script, a simple system utility, or a lightweight web app—but that's where Groovy shines brightest. This elegant JVM-based dynamic language extends and simplifies Java so you can concentrate on the task at hand instead of managing minute details and unnecessary complexity.

Making Java Groovy is a practical guide for developers who want to benefit from Groovy in their work with Java. It starts by introducing the key differences between Java and Groovy and how to use them to your advantage. Then, you'll focus on the situations you face every day, like consuming and creating RESTful web services, working with databases, and using the Spring framework. You'll also explore the great Groovy tools for build processes, testing, and deployment and learn how to write Groovy-based domain-specific languages that simplify Java development.

What's Inside

- Easier Java
- Closures, builders, and metaprogramming
- Gradle for builds, Spock for testing
- Groovy frameworks like Grails and Griffon

Written for developers familiar with Java. No Groovy experience required.

Ken Kousen is an independent consultant and trainer specializing in Spring, Hibernate, Groovy, and Grails.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/MakingJavaGroovy

Free eBook
SEE INSERT

"Focuses on the tasks that Java developers must tackle every day."
—From the Foreword by Guillaume Laforge Groovy Project Manager

"Thoroughly researched, highly informative, and mightily entertaining."
—Michael Smolyak Next Century Corporation

"A comprehensive tour through the Groovy development ecosystem."
—Sean Reilly Equal Experts in the UK

"I measured this book's ROI in Revelations per Minute."
—Tim Vold, Minnesota State Colleges and Universities

ISBN 13: 978-1-935182-94-8
ISBN 10: 1-935182-94-3
54499
9 781935 182948

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]