

Understanding web services, Part 2: Web Services Description Language (WSDL)

Sharing web services

Skill Level: Intermediate

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)

Backstop Media

07 Jul 2006

The current emphasis on Service-Oriented Architectures (SOA) has put the spotlight on web services, but it's easy to get lost in all the information being bandied about. This series gives you the straight story on all of the major web service specifications, starting with Simple Object Access Protocol (SOAP) and working down to WS Business Process Execution Language (WS-BPEL). In this second installment, you'll learn about Web Services Definition Language, as the *Daily Moon* Classified Department uses WSDL to describe their own web service in such a way that others can easily create clients to access it from any programming language or platform.

Section 1. Before you start

This tutorial is designed to give you an understanding of Web Services Description Language. It is for developers who wish to expose their own services for use by others using WSDL, and also for developers who have the WSDL file for a service they wish to access, and need to create a client.

In order to follow along with this tutorial, you should have a basic understanding of SOAP, which you can achieve by reading Part 1 of this tutorial series, and by extension, a basic understanding of XML. WSDL is programming language agnostic, but the samples at the end of the tutorial use Java and the Apache Axis2 project. The concepts, however, apply to any programming language and environment. Similarly, the tutorial focuses on the more commonly implemented WSDL 1.1, but

the concepts are the same for the upcoming WSDL 2.0, and it covers the basic differences.

About this series

This tutorial series teaches the basic concepts of web services by following the exploits of a fictional newspaper, the *Daily Moon*, as the staff uses web services to create a workflow system to increase productivity in these turbulent times.

Part 1 starts simply, explaining the basic concepts behind web services and showing you how to use SOAP, the specification that underlies most of what is to come, connecting the Classifieds Department with the Content Management System.

In Part 2, the Classifieds Department takes things a step further, using Web Services Description Language (WSDL) to define the messages produced and expected by their own web service, enabling the team to more easily create services and the clients that connect to them.

Part 3 finds the team with a number of services in place and a desire to locate them easily. In response, Universal Description, Discovery and Integration (UDDI) provides a searchable registry of available services as a way to publicize their own services to others.

Parts 4 and 5, WS-Security and WS-Policy, got a goal being securing of the paper's services and the changes the teams need to make in order to access those newly secured services.

Interoperability is the key word in Part 6, as services from several different implementations must be accessed from a single system. Part six covers the requirements and tests involved in WS-I certification.

Finally, Part 7 shows how to use Business Process Execution Language (WS-BPEL) to create complex applications from individual services.

Now let's look at what this tutorial covers in a bit more detail.

About this tutorial

Part 1 of this series, [Understanding web services: SOAP](#), introduced the staff of the fictional Daily Moon newspaper. Specifically, you met the Classifieds Department. In that tutorial, the Classifieds Department built a client to send SOAP messages back and forth to the web service representing the Content Management System. In this tutorial, the Classifieds Department, impressed by what they saw, decides to build their own web service to take and manage ads in the classifieds database. In order

to allow others to use the service, however, they will need to create a Web Services Description Language (WSDL) file. This file gives instructions to those building clients so that they know what messages the service expects and returns.

In this tutorial, you'll learn the following:

- Why WSDL files are important
- What you can do with WSDL files
- The basics of XML schema, which figure into WSDL files
- How to structure a WSDL file
- The basic differences between WSDL 1.1 and WSDL 2.0
- How to automatically generate a WSDL file from a Java class that represents a service
- How to generate a Java class that represents a service from a WSDL file
- How to generate a web service client from a WSDL file

The Classifieds Department will build a service that takes new ads, edits and displays existing ads, and finalizes an issue so that it no longer accepts any more ads. They will use both request/response and one-way messaging.

Tools and Prerequisites

The bulk of this tutorial is conceptual, but in order to follow along with the code towards the end of this tutorial, you will need to have the following software available and installed:

Apache Geronimo or another application server -- The team creates a new web service in the course of this tutorial, and you will need an application on which to run it. Because web services are, of course, supposed to be interoperable, it doesn't really matter which one you use. This tutorial demonstrates the use of Apache Geronimo, which is also the basis for IBM's WebSphere Community Edition. You can also use other application servers such as WebSphere application server. You can download Apache Geronimo from <http://geronimo.apache.org/downloads.html>.

For more information on installing Geronimo, see the first tutorial in this series, "[Understanding web services specifications, Part 1: SOAP](#)."

Apache Axis2 version 0.95 or higher -- You can create SOAP messages by hand, and you can interpret them by hand, but it's much easier to have an implementation handy. This tutorial demonstrates the use of Apache Axis2, which contains implementations of various SOAP-related APIs to make your life significantly easier. You can download Apache Axis2 at <http://ws.apache.org/axis2/download.cgi>. This tutorial uses version 0.95, but later versions should work. (Note that Part 1 of this series used version 0.94, but it has not been tested with the code in this part.)

Java 2 Standard Edition version 1.4 or higher -- All of these tools are Java-based, as are the services and clients you'll build in this tutorial. You can download the J2SE SDK from <http://java.sun.com/j2se/1.5.0/download.jsp>.

You'll also need a Web browser and a text editor, but I'm sure you already have those. If you like, you can also use an IDE such as Eclipse, but because we're focusing on the technologies rather than tools, we'll just be using a text editor and the command line to edit our files and compile them.

Section 2. Overview

Before getting into the details, let's get a look at the big picture.

Web service refresher

The main advantage of using web services rather than traditional programming methods is interoperability. You can create a distributed system, in which computers that are physically and geographically separated can communicate, but in most cases, you're dealing with proprietary middleware that limits your flexibility. In other words, both the sender and receiver need to be using the same software.

Web services provides a text based (actually, XML-based) means for transferring messages back and forth, which means that your applications are not just machine independent, but also operating system and programming language independent. As long as both parties follow the web services standards, it doesn't matter what software is running on either side.

Several standards for transferring this information exist, but this tutorial series focuses on SOAP because of its flexibility and use with more advanced standards.

SOAP refresher

A SOAP message consists of three main parts: a Header, the Body, and the payload, which is enclosed in the body. Consider this example:

Listing 1. A sample SOAP message

```
>SOAPenv:Envelope
  xmlns:SOAPenv="http://schemas.xmlsoap.org/SOAP/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"<
>SOAPenv:Body<
  >req:getNumberOfArticles xmlns:req="http://daily-moon.com/CMS/"<
    >req:category<classifieds>/req:category<
  >/req:getNumberOfArticles<
>/SOAPenv:Body<
>/SOAPenv:Envelope<
```

The overall message is called the Envelope, the contents of which consist of the Header and the Body. The Header contains information about the message itself, such as routing information, or information intended to be processed by "SOAP intermediaries", or services between the sender and the ultimate receiver, who may process the message. The Body of the message includes the "payload", which includes the actual data to be passed to the web service.

In this case, the payload is the `getNumberOfArticles` element and its contents.

What WSDL is for

When you create a service, you typically do it because you want other people to use it. In order for them to do that, they need to know what information to send to the service, what information the service is going to send back, and where to find the service in the first place. You could, of course, put this into a word processing document, but it is much more helpful to have a standard, preferably human- and machine-readable, format for this information.

WSDL provides this standard format. The main advantage, aside from a lack of ambiguity, is the fact that because it is the de facto standard, and it is in XML, it is machine-readable, to the point where you can create clients (and even the skeleton of a service) automatically. The Classifieds Department is going to create a service that accepts and manages classified ads, and in order to allow others such as job aggregation Web sites to more easily use the service, they will also describe it using a WSDL file.

What this tutorial will accomplish

During the course of this tutorial, you'll learn about WSDL by following the staff of the Daily Moon Classifieds Department as they create their own service and expose it for others to use.

First, Christine takes on XML schemas, which defines the data that can appear in the SOAP message. Next, Larry puts together the structure of an actual WSDL file to get a feel for what information they will be passing back and forth. Later, Gene takes the WSDL file and uses it as the inspiration for a Java class that represents the service, and then just to be sure, he uses Axis2 to generate a WSDL file from the Java class to see the differences. From there, Francis takes the WSDL file and uses it to generate both the basic web service, and a client to access that web service.

Let's get started.

Section 3. A quick primer on XML Schema

The Daily Moon is a pretty small newspaper, and as such the Classifieds Department can't afford a dedicated DBA. So Christine has always considered the database "her baby". After looking at the specifications for WSDL, she decides to take on the task of defining the data that can go into the SOAP messages.

Validation and options

The staff is already familiar with XML after their initial foray into web services, so the next logical step is for Christine to learn about validation.

XML validation is the process of making sure that both the structure and the content of an XML document conforms to any predefined requirements. Originally, this meant conforming to the Document Type Definition, or DTD, a structure left over from XML's roots as an SGML language. DTD's weren't especially difficult, but they did have major drawbacks in terms of lack of flexibility, lack of support for XML namespaces, and other problems, so the XML community moved on to other types of XML schemas.

Notice that the word "schemas" is not capitalized in that last paragraph. Now, in this case, I'm talking about the general use of the term, meaning the specific structure for a document, as opposed to any language for documenting that structure. The most commonly supported of these XML schemas is W3C XML Schema language, which is frequently referred to simply as XML Schema. (Note the capitalization.)

Christine sees that they will be using XML Schema to build their WSDL 1.1

document, but notes that WSDL 2.0 specifically supports other schemas, such as RELAX NG and Schematron, which can be used in its place.

The instance document

Christine starts with a sample document that approximates the data she expects to flow in and out of the system. She pulls a couple of existing ads and creates the XML:

Listing 2. The instance document

```
Listing 2: the
instance document
<?xml
version="1.0"?>
<ClassifiedList>

  <ClassifiedAd
adId="1138">
<content>Vintage
1963 T-Bird.
Less than 300
miles.
Driven by my
daughter until I
took it away.
Serious inquires
only. 555-3264
after 7
PM.</content>
<endDate>4/15/2007</endDate>
<startDate>4/1/2007</startDate>
</ClassifiedAd>

  <ClassifiedAd
adId="2187">
  <content>30
ft ladder, only
used once.
Willing to let
go for half its
worth. Has slight
dent near the
middle.
Harder than a
human head. $150
OBO.</content>
<endDate>4/30/2007</endDate>
<startDate>4/10/2007</startDate>
</ClassifiedAd>

</ClassifiedList>
```

This document is known as an "instance document", it represents the data to be defined by the XML schema. In this case, it consists of two `ClassifiedAd` elements as children of the `ClassifiedList` element. Each `ClassifiedAd` includes an `adId` attribute that corresponds to the primary key within the database, the content of the ad, and the start and end dates for the ad.

She names this document `classifieds.xml`.

The basic schema

Christie's next step is to create the basic schema documents so she can test the system. The actual schema itself is also an XML document, so she creates the basic file, `classifieds.xsd`, and a single schema element, as seen in Listing 3:

Listing 3. Schema element

```
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

</xsd:schema>
```

This element will act as the parent for all of the definitions Christine creates.

Validating the document

The next step in testing the system is to put together a simple Java application that will validate the instance document against the schema. All the class has to do is specify validation and parse the document, as shown in Listing 4.

Listing 4. The `ValidateWithSchema` class

```
import
java.io.File;
import
java.io.IOException;
import
javax.xml.parsers.DocumentBuilder;
import
javax.xml.parsers.DocumentBuilderFactory;
import
javax.xml.parsers.ParserConfigurationException;
import
org.w3c.dom.Document;
import
org.xml.sax.SAXException;

public class
ValidateWithSchema
{

    public static
void main(String
args[]) {

    DocumentBuilderFactory
dbf =
DocumentBuilderFactory.newInstance();
dbf.setValidating(true);
```

```
dbf.setAttribute(
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
    "http://www.w3.org/2001/XMLSchema");
dbf.setAttribute(
    "http://java.sun.com/xml/jaxp/properties/schemaSource",
    "classifieds.xsd");

    Document
doc = null;
    try{
DocumentBuilder
parser =
dbf.newDocumentBuilder();
        doc =
parser.parse("classifieds.xml");
    } catch
(Exception e){
e.printStackTrace();
    }
}
```

The actual method of parsing is not tremendously important; in this case, Christine builds the application using a DOM parser. First, build a factory, and then specify that any parsers it creates should be "validating parsers". (Not all parsers are validating. Some simply verify that the document is "well-formed", without verifying their actual structure.)

Next, specify the schema language to use, and the schema document itself. In this case, Christine is using a local file, but you can use any file referenceable as a URL.

Christine saves the file and compiles it, and after running it, she gets the following output, shown in Listing 5.

Listing 5. Running the empty schema

```
Warning:
validation was
turned on but an
org.xml.sax.ErrorHandler
was not set,
which is probably
not what is
desired. Parser
will
use a default
ErrorHandler to
print the first
10 errors.
Please call
the
'setErrorHandler'
method to fix
this.

Error:
URI=file:///E:/WSDLFiles/classifieds.xml
Line=2:
cvc-elt.1:
Cannot find the
```

```
declaration of
element
'ClassifiedList'.
```

The `Warning` is not significant; Christine only wants to know whether the file is valid or not, so she doesn't need any particular error handling capabilities. The `Error` is significant, because it means that the parser is in fact looking to the schema and validating the instance document.

Now she needs to add the actual definitions.

Creating a simple element

It's easy to define the first few elements. The `content`, `endDate` and `startDate` elements consist of simple strings, as shown in Listing 6.

Listing 6. Adding simple elements

```
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element
type="xsd:string"
name="content" />
  <xsd:element
type="xsd:string"
name="endDate" />
  <xsd:element
type="xsd:string"
name="startDate"
/>

</xsd:schema>
```

In this case, Christine has defined three elements that have only text (as opposed to other elements) as content. Also, none of them have attributes. The XML Schema Recommendation defines a number of different types that you can use to define your content. For example, you can specify that the `endDate` and `startDate` values must be `datetime` values.

Creating a more complex element

Of course, if all elements were that simple, you probably wouldn't need a schema in the first place. Christine goes on to define the `ClassifiedList` element (see Listing 7).

Listing 7. Defining the `ClassifiedList` element

```
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element
name="ClassifiedList">
<xsd:complexType>

<xsd:sequence>
<xsd:element
name="ClassifiedAd"
maxOccurs="unbounded"
type="ClassifiedAdType"/>
</xsd:sequence>

</xsd:complexType>
</xsd:element>

<xsd:complexType
name="ClassifiedAdType">
<xsd:sequence>
<xsd:element
type="xsd:string"
name="content" />
<xsd:element
type="xsd:string"
name="endDate" />
<xsd:element
type="xsd:string"
name="startDate"
/>
</xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

Starting at the bottom, Christine creates the `ClassifiedAdType` `complexType`. It's a type of element that contains, in order, a `content` element, an `endDate` element, and a `startDate` element. Moving up to the top, she defines the `ClassifiedList` element has an element that also contains a sequence of elements. In this case, however, it contains zero or more elements of type `ClassifiedAdType`, all named `ClassifiedAd`.

So far, with just these two definitions, she's covered most of the structure of the document. Now all she needs to do is add a definition for the `adId` attribute.

Adding attributes

An attribute gets defined with in a `complexType` element, as shown in Listing 8.

Listing 8. Adding an attribute

```
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element
name="ClassifiedList">
```

```
<xsd:complexType>
<xsd:sequence>
<xsd:element
name="ClassifiedAd"
maxOccurs="unbounded"
type="ClassifiedAdType"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:complexType
name="ClassifiedAdType">
  <xsd:sequence>
<xsd:element
type="xsd:string"
name="content" />
<xsd:element
type="xsd:string"
name="endDate" />
<xsd:element
type="xsd:string"
name="startDate"
/>
</xsd:sequence>
  <xsd:attribute
name="adId"
type="xsd:integer"
/>
</xsd:complexType>

</xsd:schema>
```

In this case, Christine has limited the content of the `adId` attribute to integers. She could, however, create a more restrictive type.

Using simpleTypes

XML Schema provides a great deal of power when it comes to defining the actual content your document may contain. For example, Christine can specify that the `adId` attribute must contain only integers greater than or equal to 1000 (see Listing 9).

Listing 9. Restricting values

```
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
<xsd:complexType
name="ClassifiedAdType">
  <xsd:sequence>
<xsd:element
type="xsd:string"
name="content" />
<xsd:element
type="xsd:string"
name="endDate" />
<xsd:element
```

```
type="xsd:string"
name="startDate"
/>
</xsd:sequence>
  <xsd:attribute
name="adId"
type="thousandOrGreater"
/>
</xsd:complexType>

<xsd:simpleType
name="thousandOrGreater">
<xsd:restriction
base="xsd:integer">
<xsd:minInclusive
value="1000"/>
</xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

This definition doesn't involve any elements, so Christine uses a `simpleType` rather than a `complexType`. She's creating a `restriction` -- a limit on the type of data -- with the base type being integers. She then adds a "facet" for that restriction, specifying that the minimum value is 1000. XML Schema defines a great number of these facets for your use. Once she's created the type, she can reference it in the attribute definition. Now that she's defined the data, Christine passes the schema on to Larry so he can build the actual WSDL.

Section 4. Creating a WSDL document

While Christine is in charge of the actual data, Larry is responsible for the actual message that moves back and forth between the service and its clients. From those messages, he creates the WSDL document.

The messages

Larry's first steps are to decide what functions the service will actually perform, and then to define the messages for the services. After consultation with the rest of the department, he comes up with a list of functions and their corresponding messages, and in the remainder of this section you will look at these functions.

createNewAd

This function takes in the content of the ad at its end date, and gets back the idea for the newly created ad (see Listing 10 and Listing 11).

Listing 10. createNewAdRequest (Input)

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/SOAP/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:createNewAdRequest
xmlns:req="http://daily-moon.com/classifieds/">
  <req:content>Vintage
1963
T-Bird...</req:content>
<req:endDate>4/30/07</req:endDate>
</req:createNewAdRequest>
  </env:Body>
```

Listing 11. createNewAdResponse (Output)

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/SOAP/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <res:createNewAdResponse
xmlns:res="http://daily-moon.com/classifieds/">
  <res:newAdId>1138</res:newAdId>
</res:createNewAdResponse>
  </env:Body>
```

editExistingAd

This function takes an existing ad and replaces its content in the database. It returns a Boolean value stating whether the operation was successful (see Listing 12 and Listing 13).

Listing 12. editExistingAdRequest (Input)

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/SOAP/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:editExistingAdRequest
xmlns:req="http://daily-moon.com/classifieds/">
  <req:ClassifiedAd>
  <req:id>1138</req:id>
  <req:content>Vintage
1963
T-Bird...</req:content>
  <req:startDate>4/1/2007</req:startDate>
  <req:endDate>4/30/2007</req:endDate>
  <req:ClassifiedAd>
</req:editExistingAdRequest
  >
  </env:Body>
```

Listing 13. editExistingAdResponse (Output)

```

<env:Envelope
  xmlns:env="http://schemas.xmlsoap.org/SOAP/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <res:editExistingAdResponse
      xmlns:res="http://daily-moon.com/classifieds/">
      <res:isOK>true</res:isOK>
    </res:editExistingAdResponse>
  </env:Body>

```

getExistingAds

This function returns a list of existing ClassifiedAds (see Listing 14 and Listing 15).

Listing 14. getExistingAdsRequest (Input)

```

<env:Envelope
  xmlns:env="http://schemas.xmlsoap.org/SOAP/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:getExistingAdsRequest
      xmlns:req="http://daily-moon.com/classifieds/">
    </req:getExistingAdsRequest>
  </env:Body>

```

Listing 15. getExistingAdsResponse (Output)

```

<env:Envelope
  xmlns:env="http://schemas.xmlsoap.org/SOAP/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <res:getExistingAdsResponse
      xmlns:res="http://daily-moon.com/classifieds/">
      <res:ClassifiedList>
        <res:ClassifiedAd>
          <res:id>1138</res:id>
          <res:content>Vintage
            1963
            T-Bird...</res:content>
          <res:startDate>4/1/2007</res:startDate>
          <res:endDate>4/30/2007</res:endDate>
        </res:ClassifiedAd>
        <res:ClassifiedAd>
          <res:id>2883</res:id>
          <res:content>Championship
            playoff
            tickets...</res:content>
          <res:startDate>4/1/2007</res:startDate>
          <res:endDate>4/30/2007</res:endDate>
        </res:ClassifiedAd>
      </res:ClassifiedList>
    </res:getExistingAdsResponse>
  >
</env:Body>

```

finalizeIssue

This function is an "in only" function, taking an issue date to finalize and returning nothing (see Listing 16).

Listing 16. finalizeIssueRequest (Input)

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:finalizeIssueRequest
xmlns:req="http://daily-moon.com/classifieds/">
      <req:issueDate>4/10/06</req:issueDate>
    </req:FinalizeIssueRequest
  >
</env:Body>
```

Armed with these messages, Larry sets out to create the actual WSDL document.

Create the basic document

Larry starts with the basic framework of a WSDL document (see Listing 17).

Listing 17. The basic WSDL document

```
<wsdl:definitions
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://ws.apache.org/axis2">

  <wsdl:types>
  </wsdl:types>

  <wsdl:message
name="createNewAdRequestMessage">
  </wsdl:message>

  <wsdl:portType
name="ClassifiedServicePortType">
  </wsdl:portType>

  <wsdl:binding
name="ClassifiedServiceBinding">
  </wsdl:binding>

  <wsdl:service
name="ClassifiedService">
  </wsdl:service>

</wsdl:definitions>
```

Like the SOAP messages it defines, WSDL is made up of XML. The definitions

reside within the `definitions` element, as you can see here. Starting at the bottom, you define the `service`. The `service` uses a particular `binding`, which is an implementation of a `portType`. The `portType` defines operations, which are made up of messages. The messages consist of XML defined in the `types` section.

The `definitions` element defines two namespaces. The first, which uses the prefix `wSDL:`, provides a namespace for the actual elements that make up the WSDL. The second, the `targetNamespace`, defines the namespace to which the items defined by the WSDL belong.

Larry starts by defining the types.

Define the types

Larry takes the definitions that Christine gave him and drops them into the `types` element, creating a new schema within the document, as shown in Listing 18.

Listing 18. Defining the types

```
<wSDL:definitions
xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  xmlns:ns1="http://org.apache.axis2/xsd"
targetNamespace="http://ws.apache.org/axis2">

<wSDL:types>
  <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://org.apache.axis2/xsd"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">

    <xs:element
type="ns1:ClassifiedAd"
name="ClassifiedAd"
/>
    <xs:complexType
name="ClassifiedAd">
      <xs:sequence>
        <xs:element
type="xs:int"
name="id" />
        <xs:element
type="xs:string"
name="content" />
        <xs:element
type="xs:string"
name="endDate" />
        <xs:element
type="xs:string"
name="startDate"
/>
      </xs:sequence>
    </xs:complexType>

    <xs:element
type="ns1:ClassifiedList"
name="ClassifiedList"
```

```
    />
    <xs:complexType
      name="ClassifiedList">
      <xs:sequence>
        <xs:element
          minOccurs="0"
          type="ns1:ClassifiedAd"
          name="ClassifiedAd"
          maxOccurs="unbounded"
        />
      </xs:sequence>
    </xs:complexType>

    <xs:element
      name="createNewAdRequest">
      <xs:complexType>
        <xs:sequence>
          <xs:element
            type="xs:string"
            name="content" />
          <xs:element
            type="xs:string"
            name="endDate" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element
      name="createNewAdResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element
            type="xs:int"
            name="newAdId" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element
      name="editExistingAdRequest">
      <xs:complexType>
        <xs:sequence>
          <xs:element
            type="ns1:ClassifiedAd"
            name="ClassifiedAd"
          />
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element
      name="editExistingAdResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element
            type="xs:boolean"
            name="isOK" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element
      name="getExistingAdsRequest">
      <xs:complexType
    />
    </xs:element>

    <xs:element
      name="getExistingAdsResponse">
```

```

<xs:complexType>
<xs:sequence>
<xs:element
type="ns1:ClassifiedList"
name="ClassifiedList"
/>
</xs:sequence>
</xs:complexType>
  </xs:element>

  <xs:element
name="finalizeIssueRequest">
<xs:complexType>
<xs:sequence>
<xs:element
type="xs:string"
name="issueDate"
/>
</xs:sequence>
</xs:complexType>
  </xs:element>

</xs:schema>
</wsdl:types>
</wsdl:definitions>

```

Starting at the top, notice that we have two sections of namespaces. The first is within the `schema` element itself. Here Larry defines two namespaces. The first, prefixed by `xs:`, is the XML Schema namespace. The second, `targetNamespace`, defines the namespace to which the definitions created by the schema belong. In other words, when the second definition creates a `complexType` called `ClassifiedAd`, that definition belongs to the namespace `http://org.apache.axis2/xsd`. However, in order to refer to that namespace, Larry needs to create another alias. You can see that alias on the definitions element, prefixed by `ns1:`. (Larry could've just as easily put that definition on the schema element, but he'll need it outside the schema element later.) The last two attributes, `elementFormDefault` and `attributeFormDefault`, refer to whether or not elements and attributes are expected to have namespace prefixes.

The first four definitions are from the schema that Christine gave Larry, but the rest define the messages Larry created earlier.

A note about namespaces

In XML, as in many programming languages, it is frequently necessary to specify a "namespace" for various elements and attributes. This gives you a means for distinguishing between elements that have the same name, but different purposes, and perhaps different origins. XML references a namespace by a URI. For example, the XML schema namespace is `http://www.w3.org/2001/XMLSchema`. For convenience, however, it is also assigned an alias, or prefix. For example, here the schema namespace has a prefix of `xs:`. Remember that the alias is just that: an alias. It is the URI that matters. Therefore, elements or attributes that are part of the

ns1: namespace are also part of the schema's targetNamespace.

Create the messages

With the types in place, Larry can define the messages that will pass back and forth in the SOAP envelope (see Listing 19).

Listing 19. Creating the messages

```
<wsdl:definitions
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:ns1="http://org.apache.axis2/xsd"
targetNamespace="http://ws.apache.org/axis2">

<wsdl:types>
  <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://org.apache.axis2/xsd"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
    ...
    <xs:element
name="createNewAdRequest">
<xs:complexType>
<xs:sequence>
<xs:element
type="xs:string"
name="content" />
<xs:element
type="xs:string"
name="endDate" />
</xs:sequence>
</xs:complexType>
  </xs:element>

    <xs:element
name="createNewAdResponse">
<xs:complexType>
<xs:sequence>
<xs:element
type="xs:int"
name="newAdId" />
</xs:sequence>
</xs:complexType>
  </xs:element>

    ...
  </xs:schema>
</wsdl:types>

<wsdl:message
name="createNewAdRequestMessage">
  <wsdl:part
name="part1"
element="ns1:createNewAdRequest"
/>
</wsdl:message>

<wsdl:message
name="createNewAdResponseMessage">
  <wsdl:part
```

```
name="part1"
element="ns1:createNewAdResponse"
/>
</wsdl:message>

<wsdl:message
name="getExistingAdsResponseMessage">
  <wsdl:part
name="part1"
element="ns1:getExistingAdsResponse"
/>
</wsdl:message>

<wsdl:message
name="editExistingAdRequestMessage">
  <wsdl:part
name="part1"
element="ns1:editExistingAdRequest"
/>
</wsdl:message>

<wsdl:message
name="getExistingAdsRequestMessage">
  <wsdl:part
name="part1"
element="ns1:getExistingAdsRequest"
/>
</wsdl:message>

<wsdl:message
name="editExistingAdResponseMessage">
  <wsdl:part
name="part1"
element="ns1:editExistingAdResponse"
/>
</wsdl:message>

<wsdl:message
name="finalizeIssueRequestMessage">
  <wsdl:part
name="part1"
element="ns1:finalizeIssueRequest"
/>
</wsdl:message>

<wsdl:portType
name="ClassifiedServicePortType">
</wsdl:portType>

<wsdl:binding
name="ClassifiedServiceBinding">
</wsdl:binding>

<wsdl:service
name="ClassifiedService">
</wsdl:service>

</wsdl:definitions>
```

Each message has a name so you can reference it later. Within each message, you define one or more parts. Note that WSDL 2.0 allows only one part per message, so Larry sticks with that convention here.

Each part has a name and the name of the element that makes up that part. The element name refers back to the types defined in the types element. Notice that the

name of the element is prefixed with `ns1:`, the prefix for the namespace that matches the `targetNamespace` for the schema. In other words, when Larry created the definition for `createNewAdResponse`, that definition went into the `http://org.apache.axis2/xsd` namespace, and because the prefix `ns1:` also refers to that namespace, you can reference it now using that prefix.

Define the interface (portType)

Messages don't do any good on their own, so Larry needs to associate them with specific operations. These operations are part of the `portType`. The `portType` contains only definitions, and not implementations. In that respect, it is much like an interface. In fact, in WSDL 2.0, the name of the `portType` has been changed to `interface`. Larry creates an operation for each of the functions (see Listing 20).

Listing 20. Adding operations

```
<wsdl:definitions
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://ws.apache.org/axis2"
  xmlns:ns1="http://org.apache.axis2/xsd"
  targetNamespace="http://ws.apache.org/axis2">

  <wsdl:types>
    ...
  </wsdl:types>

  <wsdl:message
    name="createNewAdRequestMessage">
    <wsdl:part
      name="part1"
      element="ns1:createNewAdRequest"
    />
  </wsdl:message>

  <wsdl:message
    name="createNewAdResponseMessage">
    <wsdl:part
      name="part1"
      element="ns1:createNewAdResponse"
    />
  </wsdl:message>
  ...
  <wsdl:message
    name="finalizeIssueRequestMessage">
    <wsdl:part
      name="part1"
      element="ns1:finalizeIssueRequest"
    />
  </wsdl:message>

  <wsdl:portType
    name="ClassifiedServicePortType">

    <wsdl:operation
      name="finalizeIssue">
      <wsdl:input
        message="tns:finalizeIssueRequestMessage"
      />
    </wsdl:operation>
```

```

    <wsdl:operation
name="createNewAd">
    <wsdl:input
message="tns:createNewAdRequestMessage"
/>
    <wsdl:output
message="tns:createNewAdResponseMessage"
/>
</wsdl:operation>

    <wsdl:operation
name="editExistingAd">
    <wsdl:input
message="tns:editExistingAdRequestMessage"
/>
    <wsdl:output
message="tns:editExistingAdResponseMessage"
/>
</wsdl:operation>

    <wsdl:operation
name="getExistingAds">
    <wsdl:input
message="tns:getExistingAdsRequestMessage"
/>
    <wsdl:output
message="tns:getExistingAdsResponseMessage"
/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding
name="ClassifiedServiceBinding">
</wsdl:binding>

<wsdl:service
name="ClassifiedService">
</wsdl:service>
</wsdl:definitions>

```

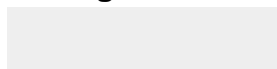
Larry creates two types of messages. One, such as `finalizeIssue`, is an "in-only" operation, in that it has only an input message. That message, `finalizeIssueRequest`, was defined previously, and like before, we created a new prefix, `tns:`, to refer to the namespace to which it belongs.

The second type of operation is a request/response operation, such as `createNewAd`. In this case, Larry defines both the input and output messages.

Define bindings

If the `portType` is like an interface, the `binding` is the implementation of that interface (see Listing 21).

Listing 21. Creating the binding



```
<wsdl:definitions
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://ws.apache.org/axis2"
xmlns:ns1="http://org.apache.axis2/xsd"
targetNamespace="http://ws.apache.org/axis2">

<wsdl:types>
</wsdl:types>

<wsdl:message
name="createNewAdRequestMessage">
  <wsdl:part
name="part1"
element="ns1:createNewAdRequest"
/>
</wsdl:message>
...
<wsdl:portType
name="ClassifiedServicePortType">

  <wsdl:operation
name="finalizeIssue">
    <wsdl:input
message="tns:finalizeIssueRequestMessage"
/>
  </wsdl:operation>

  <wsdl:operation
name="createNewAd">
    <wsdl:input
message="tns:createNewAdRequestMessage"
/>
    <wsdl:output
message="tns:createNewAdResponseMessage"
/>
  </wsdl:operation>

  <wsdl:operation
name="editExistingAd">
    <wsdl:input
message="tns:editExistingAdRequestMessage"
/>
    <wsdl:output
message="tns:editExistingAdResponseMessage"
/>
  </wsdl:operation>

  <wsdl:operation
name="getExistingAds">
    <wsdl:input
message="tns:getExistingAdsRequestMessage"
/>
    <wsdl:output
message="tns:getExistingAdsResponseMessage"
/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding
name="ClassifiedServiceBinding"
type="tns:ClassifiedServicePortType">

  <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document"
/>
```

```
<wsdl:operation
name="createNewAd">
<soap:operation
soapAction="createNewAd"
style="document"
/>
  <wsdl:input>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:input>
  <wsdl:output>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:output>
</wsdl:operation>

  <wsdl:operation
name="finalizeIssue">
<soap:operation
soapAction="finalizeIssue"
style="document"
/>
  <wsdl:input>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:input>
</wsdl:operation>

  <wsdl:operation
name="editExistingAd">
<soap:operation
soapAction="editExistingAd"
style="document"
/>
  <wsdl:input>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:input>
  <wsdl:output>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:output>
</wsdl:operation>

  <wsdl:operation
name="getExistingAds">
<soap:operation
soapAction="getExistingAds"
style="document"
/>
  <wsdl:input>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:input>
  <wsdl:output>
    <soap:body
use="literal"
/>
```

```
namespace="http://daily-moon.com/classifieds"
/>
</wsdl:output>
</wsdl:operation>

</wsdl:binding>

<wsdl:service
name="ClassifiedService">
</wsdl:service>

</wsdl:definitions>
```

First, notice that the binding type references the `ClassifiedServicePortType` Larry has already created. Second, note the addition of the `soap: namespace`. This binding is for SOAP messages over HTTP, as you can see in the `soap:binding` element. Notice the `transport` attribute. (Don't worry about the style yet; we will look at that in the next section.)

For each operation, which shares the name it had in the `portType`, Larry now adds a `soap:operation` element. This element performs two actions. One, it specifies that this is, in fact, a SOAP operation. The second is to specify a `soapAction`, an HTTP header that gets sent ahead of the actual SOAP message. The server can use this header to route the request to the appropriate operation.

Note that the `soapAction` has always been somewhat problematic, and while it is optional in WSDL 1.1, it has been removed from WSDL 2.0 completely.

Each operation also defines an `input` and an `output` message (or, in the case of in-only messages, just an `input` message), but now Larry also adds specific SOAP information. Again, don't worry about the `use` attribute; we'll look at that in a moment as well. This element also defines the namespace for the content of the payload.

Define the service

With both the interface and the binding defined, it's time to define the service (see Listing 22).

Listing 22. Define the service

```
...
<wsdl:binding
name="ClassifiedServiceBinding"
type="tns:ClassifiedServicePortType">
...
</wsdl:binding>

<wsdl:service
name="ClassifiedService">
  <wsdl:port
name="ClassifiedServicePort"
```

```

binding="tns:ClassifiedServiceBinding">
  <soap:address
location=
"http://127.0.0.1:8080/axis2/services/ClassifiedService"
/>
  </wsdl:port>
</wsdl:service>

</wsdl:definitions>

```

A service may have more than one endpoint, with each one defined by its own `port` element. The `port` element corresponds to a particular binding, and includes information on how to access it. In this case, Larry specifies that the port can be accessed via SOAP at

`http://127.0.0.1:8080/axis2/services/ClassifiedService`.

Documenting the service

With the service defined, there is one more step. It's optional, but you really should do it. WSDL enables you to add a documentation element to any element in the WSDL document. Listing 23 provides an example.

Listing 23. Documenting the service

```

...
<wsdl:portType
name="ClassifiedServicePortType">

  <wsdl:document>The
finalizeIssue
operation is an
"in-only"
operation that
tells the system
not to accept any
more ads
for a particular
date.</wsdl:document>

  <wsdl:operation
name="finalizeIssue">
  <wsdl:input
message="tns:finalizeIssueRequestMessage"
/>
</wsdl:operation>
...

```

There's a lot more to WSDL, but those are the basics. Before we move on, however, let's take a quick look at a couple of the more advanced situations you might run into.

Bindings other than SOAP over HTTP

Although the most common use of WSDL is to define SOAP over HTTP, it is by no means the only way to specify a web service. For example, Larry can create a binding that uses SOAP over e-mail, and enable it for only the `finalizeIssue` operation (see Listing 24).

Listing 24. Adding an additional binding

```
...
<wsdl:binding
name="ClassifiedServiceEmailBinding"
type="tns:ClassifiedServicePortType">

  <soap:binding
style="document"
transport="http://example.com/smtp"/>

<wsdl:operation
name="finalizeIssue">
  <wsdl:input
message="tns:finalizeIssueRequestMessage">
    <soap:body
parts="body"
use="literal"/>
  </wsdl:input>
</wsdl:operation>

</wsdl:binding>

<wsdl:service
name="ClassifiedService">
  <wsdl:port
name="ClassifiedServicePort"
binding="tns:ClassifiedServiceBinding">
    <soap:address
location=
"http://127.0.0.1:8080/axis2/services/ClassifiedService"
/>
  </wsdl:port>

  <wsdl:port
name="ClassifiedServiceEmailPort"
binding="tns:ClassifiedServiceEmailBinding">
    <soap:address
location="mailto:finalizeIssue@daily-moon.com"
/>
  </wsdl:port>
</wsdl:service>

</wsdl:definitions>
```

Here Larry has added a second binding, `ClassifiedServiceEmailBinding`. Like the `ClassifiedServiceBinding` binding, it is based on the `ClassifiedServicePortType` portType. In this case, however, he uses SMTP (the protocol used to send e-mail) rather than HTTP. He can then add a second port to the service, basing it on the new binding and using an e-mail URL as the location rather than a HTTP URL. You can create as many ports as you like for a single service.

SOAP and Mime

Another topic for which the solution is not immediately obvious is that of using SOAP with attachments. Because the message is more than just an XML document, how would you specify it using WSDL? The answer involves the fact that the message is sent using "multipart MIME". Fortunately, you can specify that in the WSDL document. For example, Larry wants to specify an operation in which users that edit an existing ad get back a PDF proof of the ad (see Listing 25).

Listing 25. Specifying a multipart message

```
...
  <wsdl:operation
name="editExistingAd">
  <soap:operation
soapAction="editExistingAd"
style="document"
/>
    <wsdl:input>
      <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
    </wsdl:input>
    <wsdl:output>
      <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
      <mime:multipartRelated>
        <mime:part>
          <soap:body
parts="body"
use="literal"/>
        </mime:part>
        <mime:part>
          <mime:content
part="docs"
type="text/html"/>
        </mime:part>
        <mime:part>
          <mime:content
part="proof"
type="image/gif"/>
        <mime:content
part="proof"
type="image/jpeg"/>
        </mime:part>
      </mime:multipartRelated>
    </wsdl:output>
  ...
```

In this case, the return message includes SOAP, an HTML document, and two image files.

Coming in WSDL 2.0

Throughout this section, I've tried to point out some of the places to in which WSDL 2.0 will be different from WSDL 1.1, such as the change from potentially multiple `portTypes` to a single `interface`, and the fact that `messages` can no longer have more than one part. Other changes are on the horizon as well.

Several factors motivate these changes, but mostly they are made for the purposes of interoperability -- constructs that are not legal under WS-I's Basic Profile are generally forbidden -- or to make it easier to use WSDL with extended SOAP specifications. For example, WSDL 2.0 uses more of a "component" model, which is better suited to integrating with the demands of specifications such as WS-Choreography.

Another change involves the formal specification of "message exchange patterns". Rather than simply counting on users to look at whether there is an input and output or just an input message, WSDL 2.0 enables you to specifically state what pattern you're using (see Listing 26).

Listing 26. Specifying message exchange patterns

```
...
  <wsdl:operation
name="finalizeIssue"
    pattern=http://www.w3.org/2006/01/wsdl/in-only">
    <wsdl:input
message="tns:finalizeIssueRequestMessage"
/>
  </wsdl:operation>

  <wsdl:operation
name="createNewAd"
    pattern="http://www.w3.org/2006/01/wsdl/in-out">
    <wsdl:input
message="tns:createNewAdRequestMessage"
/>
    <wsdl:output
message="tns:createNewAdResponseMessage"
/>
  </wsdl:operation>
...
```

Now let's get to the bottom of WSDL styles.

Section 5. WSDL styles

One of the most confusing tasks for developers just learning WSDL is choosing a style for a document. Let's look at the different choices and how they affect both the WSDL and the SOAP document.

Programming styles and encoding

In the XML world, there are generally two types of people: those who think above XML as a data format, and those who think of XML as document markup. In general, never the twain shall meet. This split continues, somewhat, into web services. There are those who consider it an XML based form of Remote Procedure Call, and those that consider it a means of getting XML information from one place to another.

In terms of WSDL, this plays out in the choice of the "style" of the message. When you create the binding, you have a choice of the document style, as Larry chose for the Classifieds service, or the RPC style. Neither style is inherently "right" or "wrong". But both have their advantages and disadvantages.

When using the RPC style, the name of the method to be executed is also the name of the root element of the payload. But wait, you say. Isn't that how the Classifieds WSDL is structured? Well, yes and no. Yes, the name of the root element is the same as the name of the method we want the service to perform. However, this is, in some sense, a coincidence; Larry has specifically structured the service that way.

Let's look at the different choices and how they play out in terms of WSDL and SOAP.

Document/Literal

The document/literal style means that the payload contains only the actual data to be passed to the service. Any routing necessary to get the message to its destination needs to be accomplished in some other way, such as through the `soapAction` header or a specific URL for the service.

The message is simple and straightforward (see Listing 27).

Listing 27. A document/literal message

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:content>Vintage
1963
T-Bird...</req:content>
<req:endDate>4/30/07</req:endDate>
  </env:Body>
</env:Envelope>
```

Notice that there is no root element for the payload. In the WSDL file, you define the elements directly, and add them to the message (see Listing 28).

Listing 28. Document/literal WSDL

```

...
<wsdl:types>
  <xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://org.apache.axis2/xsd"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
    ...
    <xs:element
      type="xs:string"
      name="content" />
    <xs:element
      type="xs:string"
      name="endDate" />
    ...
  </xs:schema>
</wsdl:types>

<wsdl:message
  name="createNewAdRequestMessage">
  <wsdl:part
    name="part1"
    element="ns1:content"
  />
  <wsdl:part
    name="part2"
    element="ns1:endDate"
  />
</wsdl:message>
...
<wsdl:portType
  name="ClassifiedServicePortType">
  ...
  <wsdl:operation
    name="createNewAd">
    <wsdl:input
      message="tns:createNewAdRequestMessage"
    />
    <wsdl:output
      message="tns:createNewAdResponseMessage"
    />
  </wsdl:operation>
  ...
</wsdl:portType>
...
<wsdl:binding
  name="ClassifiedServiceBinding"
  type="tns:ClassifiedServicePortType">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http"
    style="document"
  />
  <wsdl:operation
    name="createNewAd">
    <soap:operation
      soapAction="createNewAd"
      style="document"
    />
    <wsdl:input>
      <soap:body

```

```

use="literal" />
  </wsdl:input>
  <wsdl:output>
    <soap:body
use="literal" />
  </wsdl:output>
</wsdl:operation>

...
</wsdl:binding>
...

```

Notice that in the document style, all elements that are part of the payload also have definitions in the schema. Also, notice that the message has two distinct parts, each of which reference a specific element.

Document/Literal/wrapped

This tutorial uses the document/literal/wrapped style. It is similar to the document/literal style, except that the payload has a root element. This has the advantage of including the name of the method to execute (though this is not a requirement), as well as compliance with WS-I Basic Profile requirements. As a reminder, here is the simple message (see Listing 29).

Listing 29. A document/literal/wrapped message

```

<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:createNewAdRequest
xmlns:req="http://daily-moon.com/classifieds/">
      <req:content>Vintage
1963
T-Bird...</req:content>
      <req:endDate>4/30/07</req:endDate>
    </req:createNewAdRequest>
  </env:Body>
</env:Envelope>

```

For completeness, here's the relevant WSDL (see Listing 30).

Listing 30. Document/literal/wrapped WSDL

```

...
<wsdl:types>
  <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://org.apache.axis2/xsd"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
    ...
    <xs:element
name="createNewAdRequest">

```

```

<xs:complexType>
<xs:sequence>
<xs:element
type="xs:string"
name="content" />
<xs:element
type="xs:string"
name="endDate" />
</xs:sequence>
</xs:complexType>
  </xs:element>
  ...
</xs:schema>
</wsdl:types>

<wsdl:message
name="createNewAdRequestMessage">
  <wsdl:part
name="part1"
element="ns1:createNewAdRequest"
/>
</wsdl:message>
...
<wsdl:portType
name="ClassifiedServicePortType">

  <wsdl:operation
name="createNewAd">
    <wsdl:input
message="tns:createNewAdRequestMessage"
/>
    <wsdl:output
message="tns:createNewAdResponseMessage"
/>
  </wsdl:operation>
  ...
</wsdl:portType>
...
<wsdl:binding
name="ClassifiedServiceBinding"
type="tns:ClassifiedServicePortType">

  <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document"
/>

  <wsdl:operation
name="createNewAd">
    <soap:operation
soapAction="createNewAd"
style="document"
/>
    <wsdl:input>
      <soap:body
use="literal"
namespace="http://ws.apache.org/axis2"
/>
    </wsdl:input>
    <wsdl:output>
      <soap:body
use="literal"
namespace="http://ws.apache.org/axis2"
/>
    </wsdl:output>
  </wsdl:operation>
  ...
</wsdl:binding>

```

...

Again, all elements in the payload are defined in the schema.

RPC/Literal

The RPC style handles things a little differently. It is the WSDL proper that defines what goes into the message, and not the schema. For example, consider this SOAP message (see Listing 31).

Listing 31. An RPC/literal message

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>

    <req:createNewAdRequest
xmlns:req="http://daily-moon.com/classifieds/">
      <req:content>Vintage
1963
T-Bird...</req:content>
      <req:endDate>4/30/07</req:endDate>
    </req:createNewAdRequest>
  </env:Body>
</env:Envelope>
```

The message itself is identical to the document/literal/wrapped style, but the WSDL is very different (see Listing 32).

Listing 32. The WSDL for a RPC/literal message

```
...
<wsdl:types>
  <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://org.apache.axis2/xsd"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
    ...
  </xs:schema>
</wsdl:types>

<wsdl:message
name="createNewAdRequest">
  <wsdl:part
name="content"
element="xsd:string"
/>
  <wsdl:part
name="endDate"
element="xsd:string"
/>
</wsdl:message>
```

```

...
<wsdl:portType
name="ClassifiedServicePortType">

  <wsdl:operation
name="createNewAd">
  <wsdl:input
message="tns:createNewAdRequest"
/>
  <wsdl:output
message="tns:createNewAdResponseMessage"
/>
</wsdl:operation>
...
</wsdl:portType>
...
<wsdl:binding
name="ClassifiedServiceBinding"
type="tns:ClassifiedServicePortType">

  <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document"
/>

  <wsdl:operation
name="createNewAd">
  <soap:operation
soapAction="createNewAd"
style="rpc" />
  <wsdl:input>
  <soap:body
use="literal"
namespace="http://ws.apache.org/axis2"
/>
  </wsdl:input>
  <wsdl:output>
  <soap:body
use="literal"
namespace="http://ws.apache.org/axis2"
/>
</wsdl:output>
</wsdl:operation>
...
</wsdl:binding>
...

```

First, notice that nothing has actually been defined in the schema. Instead, the message carries the name of the method to be executed, and the message parts directly define each element. Notice also that in the RPC style, the name of the message part is significant; it is the name of the element within the payload. Message types are defined directly. (This does, of course, mean that you cannot have complex elements as part of your payload, but since this style is meant to emulate a remote procedure call, that's not a problem.)

In the portType, when you specify the message, you directly reference the message as created with those elements. Then, in the binding, by specifying the RPC style, it becomes clear how all of this translates into the SOAP message.

RPC/Encoded

The last style to look at is RPC/encoded. This style is similar to RPC/literal, except that the SOAP message defines the actual type information (see Listing 33).

Listing 33. An RPC/encoded SOAP message

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <req:createNewAdRequest
xmlns:req="http://daily-moon.com/classifieds/">
      <req:content
xsi:type="xs:string">Vintage
1963
T-Bird...</req:content>
      <req:endDate
xsi:type="xs:string">4/30/07</req:endDate>
    </req:createNewAdRequest>
  </env:Body>
</env:Envelope>
```

The WSDL for defining the message is the same as RPC/literal, but additional encoding information gets added to the binding (see Listing 34).

Listing 34. The RPC/encoded WSDL

```
<wsdl:definitions
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://ws.apache.org/axis2"

xmlns:axis2="http://ws.apache.org/axis2"
xmlns:ns1="http://org.apache.axis2/xsd"

targetNamespace="http://ws.apache.org/axis2">

  <wsdl:types>
    <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://org.apache.axis2/xsd"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">
      ...
    </xs:schema>
  </wsdl:types>

  <wsdl:message
name="createNewAdRequest">
    <wsdl:part
name="content"
element="xsd:string"
/>
    <wsdl:part
name="endDate"
```

```
    element="xsd:string"
  />
</wsdl:message>
...
<wsdl:portType
name="ClassifiedServicePortType">
  <wsdl:operation
name="createNewAd">
    <wsdl:input
message="tns:createNewAdRequest"
/>
    <wsdl:output
message="tns:createNewAdResponse"
/>
  </wsdl:operation>
  ...
</wsdl:portType>

<wsdl:binding
name="ClassifiedServiceBinding"
type="tns:ClassifiedServicePortType">

  <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="rpc" />

  <wsdl:operation
name="createNewAd">
<soap:operation
soapAction="createNewAd"
style="document"
/>
    <wsdl:input>
      <soap:body
use="encoded"
encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
namespace="http://ws.apache.org/axis2"
/>
    </wsdl:input>
    <wsdl:output>
      <soap:body
use="literal"
namespace=
"http://ws.apache.org/axis2"
/>
    </wsdl:output>
  </wsdl:operation>

  ...
</wsdl:binding>
...
```

When you create a WSDL file by hand, you have to handle all of this yourself. Fortunately, you don't always have to create it by hand.

Section 6. Generating code using WSDL

Gene and Francis are the programmers of the group at the newspaper, brought in

from the IT department whenever the Classifieds department can pry them away for one of their own projects. They're going to work on generating the WSDL code by approaching it in two ways; Gene will focus on generating code from Java to WSDL, and Francis will generate the code from WSDL to Java.

How code generation works

In the early days of WSDL, two of the first applications to appear were Java2WSDL and WSDL2Java. After all, what good is an automated format if you can't use it to automate anything? Of course, back then your options were somewhat limited. Code was oriented towards the RPC style, and it was difficult to automatically generate a system with complex payloads.

Fast forward to today, and these problems have been pretty well licked. Axis2 can generate Java code from virtually any WSDL document, and WSDL from a Java class. It accomplishes this feat by using data binding, in which an XML structure gets converted to a Java object, and vice versa. The generation process creates the code, which you can then alter, tweak, compile, and run.

First, Gene starts with a Java class and uses it to generate a WSDL document. Francis then takes that document and uses it to generate both the service and client. For the service, the generation process creates a skeleton into which you can add your own code to perform the actions you wish the service to perform. For the client, it creates a stub you can use to call a web service method as though it were a Java method.

Getting ready

The first step is to make sure that your environment is ready. Download version 0.95 of Apache Axis2, unpack it, and make sure all of the *.jar files in the lib directory are on the CLASSPATH.

To run the web service, install Apache Geronimo (if you haven't already) and start it. (See Part 1 for instructions.) Download the Axis2 v0.95 War distribution and copy it to the <GERONIMO_HOME>/deploy directory. Geronimo will deploy Axis2 automatically.

The Java classes

Gene starts with the `ClassifiedService` class, which he intends to use as the maid service, and also as a means for testing to make sure everything works the way he expects (see Listing 35).

Listing 35. ClassifiedService.java

```
package
org.dailymoon.classifieds;

public class
ClassifiedService
{
    public static
    int
    createNewAd(String
content, String
endDate){

ClassifiedAd
newAd = new
ClassifiedAd();
newAd.setEnd(endDate);
newAd.setContent(content);
newAd.save();

        return 1;
    }

    public static
    boolean
    editExistingAd(ClassifiedAd
adToEdit){

        //Do stuff
with the ad here
        return
true;
    }

    public static
    ClassifiedList
    getExistingAds(){

ClassifiedAd[]
listOfAds = {new
ClassifiedAd(),
new
ClassifiedAd(),
new
ClassifiedAd()};
ClassifiedList
listToReturn =
new
ClassifiedList(listOfAds);
        return
listToReturn;
    }

    public static
    void
    finalizeIssue(String
dateToFinalize){
        //Don't
return anything.
System.out.println(dateToFinalize
+ " finalized.");
    }
}
```

```

    public static
    void main (String
    args[]){

    ClassifiedService.createNewAd(
    "Eclipse experts
    needed. Contact
    Nick for
    details.",
    "4/21/2006");

    ClassifiedAd
    adToEdit = new
    ClassifiedAd();
    adToEdit.setId(1);
    adToEdit.setStart("4/8/2006");
    adToEdit.setEnd("4/30/2006");
    adToEdit.setContent(
    "Geronimo experts
    needed. Contact
    Nick for
    details.");

    ClassifiedService.editExistingAd(adToEdit);

    ClassifiedList
    adList =
    ClassifiedService.getExistingAds();
    System.out.println(adList.toString());

    }

}

```

The application itself is fairly straightforward. It provides an example of creating a new ad, editing an existing ad, and listing all of the existing ads. It provides basic limitations for the four methods Gene wants to expose, `createNewAd`, `editExistingAd`, `getExistingAds`, and `finalizeIssue`.

(Make sure to comment out the main method before generating the WSDL. It won't hurt anything, but it will generate extra, unnecessary code.)

The class also refers to two other classes, `ClassifiedAd`, and `ClassifiedList`. In order for the generation process to understand how to structure these objects as XML, Gene creates them as separate classes (see Listing 36).

Listing 36. ClassifiedAd.java

```

package
org.dailymoon.classifieds;

public class
ClassifiedAd {

    private int
    id;
    private String
    startDate;

```

```
        private String
        endDate;
        private String
        content;

        public void
        setId(int newId){
            id = newId;
        }

        public void
        setStartDate(String
        newStart){
            startDate =
            newStart;
        }

        public void
        setEndDate(String
        newEnd){
            endDate =
            newEnd;
        }

        public void
        setContent(String
        newContent){
            content =
            newContent;
        }

        public void
        save(){
            //Save
            data here
            System.out.println("Ad
            saved.");
        }
    }
}
```

Again, the class itself is not complete, but the structure is in place (see Listing 37).

Listing 37. ClassifiedList.java

```
package
org.dailymoon.classifieds;

public class
ClassifiedList {

    public
    ClassifiedAd[]
    listOfAds;

    public
    ClassifiedList(ClassifiedAd[]
    newListOfAds){
        listOfAds
    = newListOfAds;
    }

    public
    ClassifiedAd[]
    getRawAds(){
```

```

        return
        listOfAds;
    }

    public String
    toString(){
        return
        "This is a string
        of results.";
    }
}

```

Here Gene specifies that the `ClassifiedList` consists of an array of `ClassifiedAd` objects.

With all the classes in place, he can generate the WSDL.

Generate and massage the WSDL

Generating the WSDL is a straightforward process. From the command line, Gene issues the command, as in Listing 38:

Listing 38. Command to generate the WSDL

```

java
org.apache.axis2.wsdl.Java2WSDL
-cn
org.dailymoon.classifieds.ClassifiedService
-o

```

(Note that this should all appear on one line.)

The `-cn` switch specifies the class that forms the basis for the service. The `-o` switch specifies the output directory. Assuming there are no problems, the class quietly executes, leaving the `ClassifiedService.wsdl` file in the output directory. This file is very similar to the one Larry generated earlier -- by design, as they're all working on the same service -- but some small changes need to be made to accommodate items that were generically named by the generation process. Specifically, parameters do not always translate well, and will likely have to be renamed.

Here is the generated WSDL file, with tweaks in bold (see Listing 39).

Listing 39. The WSDL file

```

<wsdl:definitions
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

```

```

xmlns:tns="http://ws.apache.org/axis2"
  xmlns:ns1="http://org.apache.axis2/xsd"
targetNamespace="http://ws.apache.org/axis2">

<wsdl:types>
  <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://org.apache.axis2/xsd"
elementFormDefault="unqualified"
attributeFormDefault="unqualified">

    <xs:element
type="ns1:ClassifiedAd"
name="ClassifiedAd"
/>
  <xs:complexType
name="ClassifiedAd">
  <xs:sequence>
  <xs:element
type="xs:int"
name="id" />
  <xs:element
type="xs:string"
name="content" />
  <xs:element
type="xs:string"
name="endDate" />
  <xs:element
type="xs:string"
name="startDate"
/>
  </xs:sequence>
</xs:complexType>

    <xs:element
type="ns1:ClassifiedList"
name="ClassifiedList"
/>
  <xs:complexType
name="ClassifiedList">
  <xs:sequence>
  <xs:element
minOccurs="0"
type="ns1:ClassifiedAd"
name="ClassifiedAd"
maxOccurs="unbounded"
/>
  </xs:sequence>
</xs:complexType>

    <xs:element
name="createNewAdRequest">
  <xs:complexType>
  <xs:sequence>
  <xs:element
type="xs:string"
name="content" />
  <xs:element
type="xs:string"
name="endDate" />
  </xs:sequence>
</xs:complexType>
  </xs:element>

    <xs:element
name="createNewAdResponse">
  <xs:complexType>
  <xs:sequence>
  <xs:element

```

```

    type="xs:int"
    name="newAdId" />
  </xs:sequence>
</xs:complexType>
</xs:element>

  <xs:element
name="editExistingAdRequest">
  <xs:complexType>
  <xs:sequence>
  <xs:element
type="nsl:ClassifiedAd"
name="existingAd"
/>
  </xs:sequence>
  </xs:complexType>
  </xs:element>

  <xs:element
name="editExistingAdResponse">
  <xs:complexType>
  <xs:sequence>
  <xs:element
type="xs:boolean"
name="
wasSuccessful" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>

  <xs:element
name="getExistingAdsRequest">
  <xs:complexType>
  />
  </xs:element>

  <xs:element
name="getExistingAdsResponse">
  <xs:complexType>
  <xs:sequence>
  <xs:element
type="nsl:ClassifiedList"
name="ClassifiedList"
/>
  </xs:sequence>
  </xs:complexType>
  </xs:element>

  <xs:element
name="finalizeIssueRequest">
  <xs:complexType>
  <xs:sequence>
  <xs:element
type="xs:string"
name="issueToFinalize"
/>
  </xs:sequence>
  </xs:complexType>
  </xs:element>

</xs:schema>

</wsdl:types>

<wsdl:message
name="createNewAdRequestMessage">
  <wsdl:part
name="part1"
element="nsl:createNewAdRequest"

```

```
    />
  </wsdl:message>

  <wsdl:message
  name="createNewAdResponseMessage">
    <wsdl:part
    name="part1"
    element="ns1:createNewAdResponse"
    />
  </wsdl:message>

  <wsdl:message
  name="getExistingAdsResponseMessage">
    <wsdl:part
    name="part1"
    element="ns1:getExistingAdsResponse"
    />
  </wsdl:message>

  <wsdl:message
  name="editExistingAdRequestMessage">
    <wsdl:part
    name="part1"
    element="ns1:editExistingAdRequest"
    />
  </wsdl:message>

  <wsdl:message
  name="getExistingAdsRequestMessage">
    <wsdl:part
    name="part1"
    element="ns1:getExistingAdsRequest"
    />
  </wsdl:message>

  <wsdl:message
  name="editExistingAdResponseMessage">
    <wsdl:part
    name="part1"
    element="ns1:editExistingAdResponse"
    />
  </wsdl:message>

  <wsdl:message
  name="finalizeIssueRequestMessage">
    <wsdl:part
    name="part1"
    element="ns1:finalizeIssueRequest"
    />
  </wsdl:message>

  <wsdl:portType
  name="ClassifiedServicePortType">

    <wsdl:operation
    name="finalizeIssue">
      <wsdl:input
      message="tns:finalizeIssueRequestMessage"
      />
    </wsdl:operation>

    <wsdl:operation
    name="createNewAd">
      <wsdl:input
      message="tns:createNewAdRequestMessage"
      />
      <wsdl:output
      message="tns:createNewAdResponseMessage"
      />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:binding>
</wsdl:service>
</wsdl:definitions>
```

```
</wsdl:operation>
  <wsdl:operation
name="editExistingAd">
  <wsdl:input
message="tns:editExistingAdRequestMessage"
/>
  <wsdl:output
message="tns:editExistingAdResponseMessage"
/>
</wsdl:operation>

  <wsdl:operation
name="getExistingAds">
  <wsdl:input
message="tns:getExistingAdsRequestMessage"
/>
  <wsdl:output
message="tns:getExistingAdsResponseMessage"
/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding
name="ClassifiedServiceBinding"
type="tns:ClassifiedServicePortType">

  <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
style="document"
/>

  <wsdl:operation
name="createNewAd">
  <soap:operation
soapAction="createNewAd"
style="document"
/>
    <wsdl:input>
      <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
    </wsdl:input>
    <wsdl:output>
      <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
    </wsdl:output>
  </wsdl:operation>

  <wsdl:operation
name="finalizeIssue">
  <soap:operation
soapAction="finalizeIssue"
style="document"
/>
    <wsdl:input>
      <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
    </wsdl:input>
  </wsdl:operation>

  <wsdl:operation
name="editExistingAd">
```

```
<soap:operation
soapAction="editExistingAd"
style="document"
/>
  <wsdl:input>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:input>
  <wsdl:output>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:output>
</wsdl:operation>

  <wsdl:operation
name="getExistingAds">
  <soap:operation
soapAction="getExistingAds"
style="document"
/>
  <wsdl:input>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:input>
  <wsdl:output>
    <soap:body
use="literal"
namespace="http://daily-moon.com/classifieds"
/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service
name="ClassifiedService">
  <wsdl:port
name="ClassifiedServicePort"

  binding="tns:ClassifiedServiceBinding">
    <soap:address
location=
"http://127.0.0.1:8080/axis2/services/ClassifiedService"
/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Most of these changes are just convenience, as well as usability; it's much easier to remember "content" than "param0". Two of these changes -- the namespace at the top and the namespace prefix at the bottom -- are due to slight bugs in the generation process as it exists in Axis2 version 0.95, and by the time you read this, they may no longer be necessary.

Generate the service from the WSDL

Once the WSDL file exists, Francis can use it to generate the service and the client. (Actually, Francis could just as easily have used the version generated by Larry.)

Francis starts by generating the server-side code, as in Listing 40:

Listing 40. The server-side code

```
java
org.apache.axis2.wsdl.WSDL2Java
-uri
ClassifiedService.wsdl
-ss -sd -p
org.dailymoon.classifieds
-d xmlbeans -o
service
```

(Again, enter this command on a single line.)

The first parameter is the URL for the WSDL file. Yes, you can access a remote file using this application. The second switch, `-ss`, tells the application to generate the service, as opposed to the client. The `-sd` switch tells the application to generate the XML service descriptor, making it easier for you to deploy the service once you've generated it. The next parameter is, of course, the package, followed by the data binding method. Available methods are `adb`, `xmlbeans`, and `jaxme`. Finally, to keep things clean, Francis generates the service into a new directory called `source`.

The result is literally hundreds of files. Fortunately, you only need to deal with one of them.

Implement the service

Although in this case the service has been generated from a WSDL file that was itself generated from a Java class, there is no actual logic in the generated code. Only the structure appears. To get the service to actually do something, you need to edit the skeleton file.

In this structure, the file in question is shown in Listing 41:

Listing 41. File generated from a Java class

```
service\src\org\dailymoon\classifieds\ClassifiedServicePortTypeSkeleton.
java
```

The code is heavily commented, which is useful when you try to figure it out for yourself, but can be very distracting when you're trying to explain it. Here is the cleaned up version, along with the added code to implement part of the service (see Listing 42).

Listing 42. ClassifiedServicePortTypeSkeleton.java

```
package
org.dailymoon.classifieds;

public class
ClassifiedServicePortTypeSkeleton
{
    public
axis2.apache.org.xsd.CreateNewAdResponseDocument
createNewAd
(axis2.apache.org.xsd.CreateNewAdRequestDocument
param0 )
throws Exception
{
    //Todo fill
this with the
necessary
business logic
    //throw new
java.lang.UnsupportedOperationException();

    System.out.println("New
ad requested, to
end on " +
param0.getCreateNewAdRequest().getEndDate());
System.out.println(
param0.getCreateNewAdRequest().getContent());

axis2.apache.org.xsd.CreateNewAdResponseDocument
responseDoc =
axis2.apache.org.xsd.CreateNewAdResponseDocument
.Factory.newInstance();

axis2.apache.org.xsd.CreateNewAdResponseDocument
.CreateNewAdResponse
response =
responseDoc.addNewCreateNewAdResponse();

response.setNewAdId(1138);
return
responseDoc;
}

public void
finalizeIssue
(axis2.apache.org.xsd.FinalizeIssueRequestDocument
param2)
throws Exception
{
    //Todo
fill this with
the necessary
business logic
}

public
axis2.apache.org.xsd.EditExistingAdResponseDocument
editExistingAd
(axis2.apache.org.xsd.EditExistingAdRequestDocument
param3)
throws Exception
```

```
{
    //Todo fill
    this with the
    necessary
    business logic
    throw new
    java.lang.UnsupportedOperationException();
}

public
axis2.apache.org.xsd.GetExistingAdsResponseDocument
getExistingAds
(axis2.apache.org.xsd.GetExistingAdsRequestDocument
param5)
throws Exception
{
    //Todo fill
    this with the
    necessary
    business logic
    throw new
    java.lang.UnsupportedOperationException();
}
}
```

Each method starts life throwing an `UnsupportedOperationException`, until you actually implement the method. To get at the data submitted to the service, start with the parameter and get the request itself. From there, you can extract individual members using getter methods.

Obviously, in a real service, you'd want to do more than simply output text, but Francis is just interested in making sure it's working. To create the response, start with the appropriate response document, acquiring an instance through the class's `Factory`. (The classes themselves are quite complex, containing a number of inner classes, but it's worth having a look to see how they're structured.) Once you have the document, create the actual response itself and add it to that document.

Using setter methods, you can directly set values on the response. Simply return of the response document, and the support classes will handle sending it back to the requester.

Deploy the service

To deploy the service, you will need to compile it and turn it into an Axis2 archive file. Start by compiling and packaging the service, as shown in Listing 43.

Listing 43. Package the service

```
set
```

```
ANT_HOME=e:\apache-ant-1.6.5
PATH=%PATH%;%ANT_HOME%\bin;
set
AXIS2_HOME=e:\axis2
cd service
ant jar.service
```

Adjust your syntax appropriately for non-Windows installations, and be sure to use your actual file locations.

This Ant task compiles all of the appropriate files, and creates two archive files, ClassifiedService.aar, and XBeans-packaged.jar, both in the build/lib directory.

To deploy the service, make sure Geronimo is running and point your browser as shown in Listing 44:

Listing 44. Deploying the service

```
http://localhost:8080/axis2/Login.jsp
```

Log in with the credentials admin/axis2 and click **Upload Service>Browse**. Navigate to the ClassifiedService.aar file and click OK. Click Upload to complete the process.

If you click View Services, you should see the new service listed.

Generate the client stub from the WSDL

All that's left now is to generate the client to access the new service. To do that, execute the following command from the command line:

Listing 45. Command to generate the client

```
java
org.apache.axis2.wsdl.WSDL2Java
-uri
ClassifiedService.wsdl
-p
org.dailymoon.classifieds
-d xmlbeans -o
client
```

As usual, this is a single command, meant for a single line. The parameters are virtually identical to those used for the server-side generation, except that you don't need a service descriptor. Also, to keep things clean, Francis puts the new files in a separate, client, directory.

This class should execute quietly, leaving hundreds of files in its wake, but you don't need to deal with any of them directly.

Create the client

The code generation process doesn't actually create a client, but it does create a class you can use to easily create a client. To simplify compilation, Francis creates a new class file called `Client.java` in the `client\src\org\dailymoon\classifieds` directory. This way, Ant will pick up the `.java` file and compile it with the rest of the source.

Francis adds the code in Listing 46.

Listing 46. The client

```
package
org.dailymoon.classifieds;

import
axis2.apache.org.xsd.*;

public class
Client{

    public static
    void
    main(java.lang.String
    args[]){

        try{
        ClassifiedServicePortTypeStub
        stub =

            new
            ClassifiedServicePortTypeStub(null,
            "http://localhost:8080/axis2/services/ClassifiedService");

        CreateNewAdRequestDocument
        cnaDoc =
        CreateNewAdRequestDocument.Factory.newInstance();

        CreateNewAdRequestDocument.CreateNewAdRequest
        cnaReq =
        cnaDoc.addNewCreateNewAdRequest();
        cnaReq.setContent("Vintage
        1963 T-Bird...");
        cnaReq.setEndDate("7/4/06");

        CreateNewAdResponseDocument
        cnaResDoc =
        stub.createNewAd(cnaDoc);
        System.out.println("New
        ad ID number: "+
        cnaResDoc.getCreateNewAdResponse().getNewAdId());

        }
        catch(Exception
        e){
        e.printStackTrace();
        }
    }
}
```

The `ClassifiedServicePortTypeStub` class represents the actual service, and you instantiate it with the `AXIS_HOME` (here left to the default) and the location of the actual service. Next, create the request document, once again by referencing its `Factory`, and use it to create a new `CreateNewAdRequest`, adding it to the request document in the process. Just as in the service itself, you can then set attributes directly using setter methods.

To get the response, use the stub to execute the `createNewAd()` method, passing it the request document as a parameter. Once you have the response document, or rather the `CreateNewAtResponseDocument`, you can use it to extract the response itself, and an attribute of that response.

Now let's run it.

Run the client

To run a client, Francis first needs to compile it. Execute the following steps (see Listing 47).

Listing 47. Compiling the client

```
>>set
ANT_HOME=e:\apache-ant-1.6.5
>>PATH=%PATH%;%ANT_HOME%\bin;
>>set
AXIS2_HOME=e:\axis2
>>cd client
>>ant jar.client
Buildfile:
build.xml

init:

pre.compile.test:
 [echo] Stax
Availability=
true
 [echo] Axis2
Availability=
true

compile.src:

compile.test:

jar.client:

BUILD SUCCESSFUL
Total time: 2
seconds
```

First, make sure the environment has the appropriate variables. (This assumes you have already added all of the AXIS2_HOME\lib jar files.) Next, change to the client directory (or whatever directory uses the output for the generation process) and run Ant against the jar.client target. You should see results similar to those shown in italics (see Listing 47). To run the client, first amend the CLASSPATH to include the resources directory and the directory that houses all of the classes created by the data binding process (see Listing 48).

Listing 48. Running the client

```
>>set
CLASSPATH=E:\WSDLFiles\client\resources\;E:\WSDLFiles\client\build\classes\axis2\apache\org\xsd\;%CLASSPATH%
>>cd
build\classes
>>java
org.dailymoon.classifieds.Client
```

You should see results like that shown in Listing 49:

Listing 49. New ad ID number

```
New ad ID number :
1138
```

That, as they say, is all there is to it.

Section 7. Summary

In Part 1 of this series, the Classifieds Department of the Daily Moon newspaper learned how to use SOAP web services to connect with the newspaper's Content Management System. In this tutorial, the staff learned to use Web Services Description Language (WSDL) to describe their own services so that others could use them. The tutorial covered the basics of XML schema, along with the structure of a WSDL file so that it can be built by hand. It also discussed the differences between the various styles and encodings available. The tutorial also explained how to use the Java2WSDL and WSDL2Java tools that come with Apache Axis2 to automatically generate WSDL from Java files, and vice versa.

Part 3 of this series will look at building web service registries using UDDI.

Downloads

Description	Name	Size	Download method
Sample Perl scripts	ws-understand-web-services2-WSDL10216	102KB	HTTP

[Information about download methods](#)

Resources

Learn

- In Part 1 of this series, [Understanding web Services specifications, Part 1: SOAP](#) you'll learn more about web services in general and SOAP web services in particular.
- Access the [WSDL 1.1 specification](#).
- The WSDL 2.0 (Candidate) Recommendation consists of three documents: [Part 0: Primer](#), [Part 1: Core Language](#), and [Part 2: Adjuncts](#). You can also check out two related documents, [SOAP 1.1 Binding](#) and [RDF Mapping](#).
- Check out the [Apache Axis2 web site](#) for links to the documentation for your particular version.
- The [Apache Geronimo web site](#) has details on Apache Geronimo, the J2EE server project of the Apache Software Foundation.
- The developerWorks tutorial, "[Introduction to XML tutorial](#)" (August, 2002) offers a good grounding in XML.
- For a better understanding of dealing with XML as an object, read [Understanding DOM](#).
- [Which style of WSDL should I use?](#) is an excellent article explaining document vs RPC and encoded vs literal, including the pros and cons of each.
- [Supercharging WSDL with RDF](#) explains how to get more out of your WSDL by incorporating the Resource Description Framework, the World Wide Web Consortium's official format for defining the metadata of XML objects.
- You can add additional power and flexibility to your WSDL by importing and including data, but it can be tricky. [Web services Programming Tips and Tricks: WSDL file imports](#) explains how to do it correctly.
- Axis2 isn't the only game in town when it comes to generating code to and from WSDL. [Top-down web service development: Build a WSDL file to generate a web service using WebSphere Studio](#) provides another view.
- [Processing WSDL documents with XSLT](#) presents approaches that can be used to extract a deep understanding of WSDL documents in an XSL stylesheet as well as common mistakes you should avoid.
- You can find a wealth of information on the [developerWorks SOA and web services zone](#).
- At the [developerWorks Open Source zone](#) you can learn all about Apache Geronimo.

- Read [Online banking with Apache Geronimo and Axis2 tutorial series](#) on developerWorks for an example of Axis2 at work.
- Check out the [developerWorks Java zone](#) to learn more about programming in Java.
- Learn all about XML at the [developerWorks XML zone](#).

Get products and technologies

- Get [Apache Geronimo](#).
- Get [Apache Axis2](#). This tutorial uses version 0.95, but later versions should work.
- Get the [J2SE SDK](#).
- Get [Apache Ant](#).

About the author

Nicholas Chase



Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the Chief Technology Officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams).