

Getting Started with the Apache Axis Project

This chapter focuses on getting you started with the Axis Project. You will not be digging into anything with depth (saving the depth for later chapters); rather, you are going to focus solely on the basic components of an Axis Web Service and how these components are assembled in a basic web service.

What is the Axis Project?

At its core, the Axis Project is a third generation Simple Object Access Protocol (SOAP) engine. At the highest level, it is a complete framework for constructing and deploying interoperable XML transactions using SOAP. The Axis Project is an open-source Java implementation of SOAP v1.1.

What is SOAP?

According to the W3C (the tenants of the SOAP specifications), “SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment.” In other words, SOAP is a relatively easy-to-use method of communicating over different Internet protocols, using XML as the message layer. It’s a wire protocol that leverages HTTP as its transport layer and XML as its data layer to execute remote methods, known as SOAP/Web services.

Historically, SOAP was introduced by companies like Microsoft, IBM, and several others. The current specification is SOAP v1.1. However, as of this writing, the W3C has released a working draft of SOAP 1.2, which demystifies some of the more bewildering areas of the 1.1 specification.

The Axis implementation of SOAP provides four methods of invoking SOAP services:

- Remote Procedure Call (RPC)
- Document
- Wrapped
- Message

Remote Procedure Call (RPC) Services

The RPC method is a synchronous technique using a client server model to execute remote SOAP service. This model is defined by following sequence:

1. A client application builds an XML document containing the URI of the server that will service the request, the name of the method to execute on the server, and the parameters associated with the method.
2. The targeted server receives and unwinds the XML document. It then executes the named method.
3. After the named method has returned its results, Axis maps them into a response XML document and then sends them back to the calling client.
4. The client application receives the response and unwinds the results, which contain the response of the invoked method.

Note: You will be spending the majority of your time on the RPC flavor of Axis SOAP services.

Document and Wrapped Services

Document and Wrapped services are very much alike and therefore can really be discussed within a single context. They both differ from an RPC service in that neither uses SOAP-style bindings in their data-layer representation; their associated data is packaged as simple XML schema data. Therefore, when you look at a Document or Wrapped message coming across the wire, you would see a “non-SOAPified” XML message.

The Document and Wrapped services differ from each other only in the way they are processed by the actual implemented service. A Document service that has a schema similar to the following:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.sourcebeat.com/Person">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="first" type="xsd:string"/>
        <xs:element name="last" type="xsd:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A Document service would invoke a service method with the following signature:

```
public void methodName(Person person);
```

A Wrapped service would invoke a service method with the following “unwrapped” signature:

```
public void methodName(String first, String last);
```

Note: The names of the two previous methods are not important to this example. It is only important that you see the parameter lists.

Message Services

The final SOAP service method provided by the Axis project is the Message service. Use the Message service when you want to provide a service with direct access to the XML data, as opposed to letting Axis map your message into Java objects. As I stated earlier, you’re not going to see much about the type of service, but if you’re interested in seeing how a Message service is constructed, you can find a sample in the Axis archive.

```
<AXIS_HOME>/samples/message/MessageService.java
```

Assembling an Axis Web Service

Now that you have a basic understanding of SOAP and its services, look at how the Axis Project implements a service. Several components make up an Axis service. The goal of this section is to show you how to assemble and deploy a simple Axis web service. The steps that you will use to assemble your web service include:

1. Create the service implementation and its supporting classes.
2. Create a client application to execute your web service.

Creating and Deploying the Web Service

This section demonstrates the actual web service implementation. This is the code that will be executed on the server-side of your SOAP applications. Creating an RPC-based SOAP service is a very simple process that can be broken down into two steps. The following two sections describe these steps.

Creating the Service Implementation

Creating a SOAP service is the simplest step of the entire "SOAPifying" process. A SOAP service can be just about any Java class that exposes public methods for invocation. The class does not need to know anything about SOAP or even that it is being executed as a SOAP service. The only restriction is that the method parameters must be serializable. *Chapter 4, Axis Inputs and Outputs*, describes method parameters in greater detail.

For this example, create a web service that will act as a somewhat stupid inventory manager. It will contain three classes: the first is the web service itself (Listing 2.1), and the second and third classes contain the business logic used to support the service (Listings 2.2 and 2.3).

```
import ch02.InventoryItem;

public class InventoryService {

    public int addInventory(String sku, int quantity) {

        InventoryItem item = new InventoryItem(sku, quantity);

        System.out.println(item);

        return (ch02.Inventory.addInventory(item)).getQuantity();
    }

    public int reduceInventory(String sku, int quantity) {

        return (ch02.Inventory.reduceInventory(sku,
            quantity)).getQuantity();
    }

    public int getInventoryQuantity(String sku) {

        InventoryItem item = ch02.Inventory.getInventoryItem(sku);
        return item.getQuantity();
    }
}
```

Listing 2.1: InventoryService.java – The InventoryService Implementation

This InventoryService is a simple Java class that contains three public methods: addInventory(), reduceInventory(), and getInventoryQuantity(). Each of these methods takes simple Java types in their associated method signatures. The InventoryItem used in this service is shown below in Listing 2.2:

```
package ch02;

import java.io.Serializable;

public class InventoryItem implements Serializable {

    public String sku = null;
    public int quantity = 0;

    public InventoryItem() {

    }

    public InventoryItem(String sku, int quantity) {

        this.sku = sku;
        this.quantity = quantity;
    }

    public String getSku() {

        return sku;
    }

    public void setSku(String sku) {

        this.sku = sku;
    }

    public int getQuantity() {

        return quantity;
    }

    public void setQuantity(int quantity) {

        this.quantity = quantity;
    }

    public String toString() {

        return ("SKU = " + sku + " QUANTITY = " + quantity);
    }
}
```

Listing 2.2: InventoryItem.java – The InventoryItem Bean

This is a simple bean that represents a single piece of inventory.

Note: Why pass Integers and Strings back and forth as opposed to using the InventoryItem directly? Axis requires additional configuration to pass non-native objects. Chapter 4 discusses how to pass user-defined objects in detail.

The final server-side class is a bit more complex, but hardly shocking. It actually encapsulates the business logic for your web service. The `ch02.Inventory` class contains a `Hashtable` of tag/value pairs representing an inventory SKU and `InventoryItem`, respectively, and the methods used to add, modify, and retrieve those values.

```
package ch02;

import java.util.Hashtable;

public class Inventory {

    private static Hashtable inventory = new Hashtable();

    public static InventoryItem
        addInventory(InventoryItem item) {

        System.out.println("ADDING : " + item);

        if ( inventory.containsKey(item.getSku()) ) {

            InventoryItem currentItem =
                (InventoryItem)inventory.get(item.getSku());
            currentItem.setQuantity(currentItem.getQuantity() +
                item.getQuantity());
        }
        else {

            inventory.put(item.getSku(), item);
        }
        return (InventoryItem)inventory.get(item.getSku());
    }

    public static InventoryItem
        reduceInventory(InventoryItem item) {

        InventoryItem currentItem = null;

        if ( inventory.containsKey(item.getSku()) ) {

            currentItem =
                (InventoryItem)inventory.get(item.getSku());

            currentItem.setQuantity(currentItem.getQuantity() -
                item.getQuantity());
        }
        else {
```

```
        // this is an error and will be handled in Chapter 5
    }
    return currentItem;
}

public static InventoryItem getInventoryItem(String sku) {

    InventoryItem item = null;

    if ( inventory.containsKey(sku) ) {

        item = (InventoryItem)inventory.get(sku);
    }
    else {

        // this is an error and will be discussed in Chapter 5
    }
    return item;
}
}
```

Listing 2.3: Inventory.java – The Inventory Business Object Implementation

After you have had the opportunity to look over these classes, compile them so that you can move on to deployment.

Deploying the Web Service

This section describes the web service deployment process—with a caveat. You are actually going to look at an Axis deployment descriptor and then not use it.

The standard method of deployment in Axis is to use something known as a Web Service Deployment Descriptor, or WSDD file. While this sounds like a complex file, it is actually a very simple XML file that is used to describe your web service and some of its associated properties. Here's a sample that could be used to deploy a web service:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="InventoryService" provider="java:RPC">
    <parameter name="className" value="ch02.InventoryService"/>
    <parameter name="allowedMethods" value="*/>
  </service>

</deployment>
```

As you can see, the file is not very complex. It contains a couple of namespaces, which can be ignored for now, and then a `<service>` element and two `<parameter>` sub-elements used to describe the web service and how it is to be deployed. Don't focus on this file for more than a couple of seconds; you are not going to use it for now. I just wanted you to see what the normal process is. You will be looking at an entire chapter on web service deployment and undeployment later in this text.

For your current purposes, you're going to use the simplest form of Axis deployment called automatic deployment. It is a simple process that can be broken into three steps:

Note: This method of deploying web services is limited and not recommended. I only cover it to make these first examples simple and less confusing. The only time that you may want to use this deployment method is during development.

1. Change the extension of the service implementation source file from `*.java` to `*.jws`.

```
InventoryService.java to InventoryService.jws
```

Note: You only compiled this file earlier to make sure that it could be compiled. It really had nothing to do with this process.

2. Copy the newly renamed service file to the following directory of your hosting web application.

```
<TOMCAT_HOME>/webapps/axis/ch02
```

3. Compile and move any dependant class files to the `classes/` directory of your hosting web application. For your example, the classes are `ch02/InventoryItem.class` and `ch02/Inventory.class`, and the directory is:

```
<TOMCAT_HOME>/webapps/axis/WEB-INF/classes/ch02
```

Now you have a complete web service that is implicitly deployed and can be executed by any web service client that is aware of it. Next, look at how this service is executed.

Note: It may appear that the `InventoryService` is in the public package, while its supporting class, `Inventory`, is in the package `ch02`. However, that is not exactly true; by placing a `*.jws` file in a directory matching the `ch02` package, you have placed the service into the same package. If you were to place the package statement in the class file, it would receive a `java.io.FileNotFoundException`.

Creating the Client

Now write a client that will execute the service's methods. The Axis Project provides a client-side API that makes it relatively simple to create SOAP clients. An example client, which is used to execute the previously deployed methods, can be found in Listing 2.4.

```
package ch02;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;

import javax.xml.rpc.ParameterMode;

public class InventoryClient {

    public Integer addInventory(String endpoint)
        throws Exception {

        String method = "addInventory";

        String sku = new String("SKU456");
        Integer quantity = new Integer(8);

        Service service = new Service();
        Call call = (Call) service.createCall();

        call.setTargetEndpointAddress(new java.net.URL(endpoint));
        call.setOperationName(method);
        call.addParameter("sku", XMLType.XSD_STRING,
            ParameterMode.IN);
        call.addParameter("quantity", XMLType.XSD_INT,
            ParameterMode.IN);
        call.setReturnType(XMLType.XSD_INT);

        Object parameters[] = new Object[] {sku, quantity};
        return (Integer)call.invoke(parameters);
    }

    public Integer reduceInventory(String endpoint)
        throws Exception {

        String method = "reduceInventory";

        String sku = new String("SKU456");
        Integer quantity = new Integer(2);

        Service service = new Service();
```

```
Call call = (Call) service.createCall();

call.setTargetEndpointAddress(new java.net.URL(endpoint));
call.setOperationName(method);
call.addParameter("sku", XMLType.XSD_STRING,
    ParameterMode.IN);
call.addParameter("quantity", XMLType.XSD_INT,
    ParameterMode.IN);
call.setReturnType(XMLType.XSD_INT);

Object parameters[] = new Object[] {sku, quantity};

return (Integer)call.invoke(parameters);
}

public Integer getInventoryQuantity(String endpoint)
    throws Exception {

    String method = "getInventoryQuantity";

    String sku = new String("SKU456");

    Service service = new Service();
    Call call = (Call) service.createCall();

    call.setTargetEndpointAddress(new java.net.URL(endpoint));
    call.setOperationName(method);
    call.addParameter("sku", XMLType.XSD_STRING,
        ParameterMode.IN);
    call.setReturnType(XMLType.XSD_INT);

    Object parameters[] = new Object[] {sku};
    return (Integer)call.invoke(parameters);
}

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

    String endpoint =
        "http://localhost:8080/axis/ch02/InventoryService.jws";

    InventoryClient client = new InventoryClient();

    Integer ret = client.addInventory(endpoint);
    System.out.println("Adding Got result : " + ret);

    ret = client.reduceInventory(endpoint);
    System.out.println("Removing Got result : " + ret);

    ret = client.getInventoryQuantity(endpoint);
    System.out.println("Getting Got result : " + ret);
}
```

```
}  
}
```

Listing 2.4: InventoryClient.java – The Inventory Client

The best place to begin the examination of this client is in the main() method. The first thing to notice is the String named endpoint. This String represents the location of your web service.

The value of this String is a URL that points to the localhost on port 8080 and makes a request for the file InventoryService.jws located in the axis web application. You could ignore the localhost:8080 (it just points to your instance of Tomcat), but you do need to take a look at what it is requesting. This is the file that was renamed and moved into the Axis web application; it contains your web service.

When the Axis web application receives a request for a file with a .jws extension, it recognizes it as a special extension because of the following <servlet-mapping> element and passes the request to the AxisServlet for processing.

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

Next, move on to one of the three methods defined by the InventoryClient. The method signatures of all the methods are the same; they all take a single parameter, a URL, to your targeted .jws. Now look at the previously listed InventoryClient.addInventory() method.

The first line of the addInventory() method is simple; it creates a String set to the value of the addInventory. If you remember, this is the name of one of the methods found in the InventoryService.

```
String method = "addInventory";
```

Then it creates the two objects that you will pass to the web service--sku and quantity--which all map to the key value pairs of your Inventory object.

```
String sku = new String("SKU456");  
Integer quantity = new Integer(8);
```

This is where you start using some of the Axis APIs. First, an instance of `org.apache.axis.client.Service` is created. The `Service` class is the factory that creates the `org.apache.axis.client.Call`, which is the object that actually invokes the web service.

```
Service service = new Service();
Call call = (Call) service.createCall();
```

Once you have a `Call` instance, start setting the properties needed to invoke your web service. The first property set is the target address of the web service, which in this case is the `String` passed into this method, `http://localhost:8080/axis/InventoryService.jws`.

```
call.setTargetEndpointAddress(new java.net.URL(endpoint));
```

The next property sets the name of the method that will be invoked, `addInventory`.

```
call.setOperationName(method);
```

The next section of code is a bit interesting:

```
call.addParameter("sku",
    XMLType.XSD_STRING, ParameterMode.IN);

call.addParameter("quantity",
    XMLType.XSD_INT, ParameterMode.IN);
```

Upon first glance, it appears that you're adding the parameter names and values used by the web service, but in reality, you are telling Axis to expect two parameters, `sku` and `quantity`. Their values will be of type `String` and `Integer`, respectively. The real parameter values that will be bound to this request are not actually set until the `Call.invoke()` method is invoked, which you will see in just a moment.

The final property set is the expected return type, which for this example is `Integer`.

```
call.setReturnType(XMLType.XSD_INT);
```

Note: In this example, the properties of the call object are set using each property's explicit method. For instance, the `call.setOperationName()` method is used to set the name of the web service method that's going to be invoked. The class `org.apache.axis.client.Service` provides alternative versions of the `createCall()` method that allows you to pass most of this information in the `createCall()` parameter list. I have chosen to use these methods to explicitly show each step of the process and describe each property being set.

Now all of the properties are set, and it's time to invoke your web service. As discussed earlier, Axis has been told what to expect, but no parameter values have actually been passed to the Call instance. To do this, add all of the parameters to an array of objects and then pass the array to the Call.invoke() method, which is done below.

```
Object parameters[] = new Object[] {sku, quantity};  
return (Integer) call.invoke(parameters);
```

When this method is called, it will invoke the addInventory() method of the Inventory web service and pass it the two values for sku and quantity (SKU456 and 8). When it completes, it will return the Integer value result of the addInventory() operation.

Note: As for the rest of this client, you don't need to go over the remaining methods, remove() and getInventory(). They are very similar to the add() method and are just provided to round out the example.

Executing the Client

It is now time to see the results of your labors. To execute this client, complete the following steps:

1. Make sure that your CLASSPATH was set according to the directions provided in *Chapter 1, Installation and Configuration*.
2. Start Tomcat.
3. Test the results of your web service deployment by opening your browser to the following URL:

```
http://localhost:8080/axis/ch02/InventoryService.jws?wsdl
```

If the previous section's deployment was performed correctly, you should see an image similar to Figure 2.1. For now, you can ignore the generated WSDL, but make sure that you see something similar to the following image.

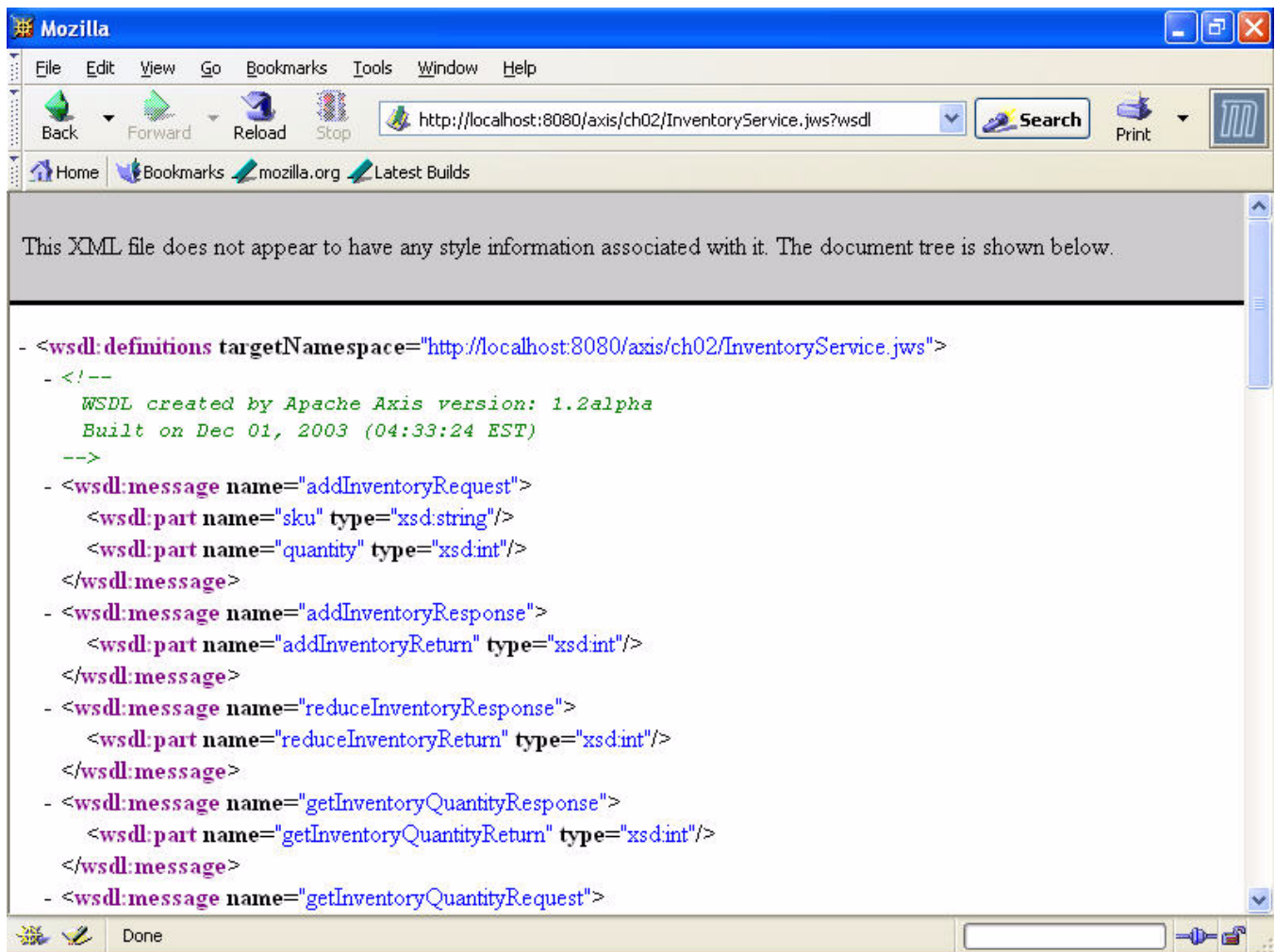


Figure 2.1: The WSDL Defining Your Inventory Web Service

4. Compile the InventoryClient.
5. Execute the compiled InventoryClient. If everything went well, you should see output similar to the following:

```
Adding Got result : 8  
Removing Got result : 6  
Getting Got result : 6
```

Summary

This chapter covered quite a bit of material. You began with a brief look at what Axis is and how it implements some of the SOAP services. You then moved on to actually create an Axis web service, deploy that web service, and then execute the deployed web service.

The goal of this chapter was to provide you with an overview of what Axis is and how you can use it. At this point, you should feel comfortable with the basic components of an Axis project and how it is assembled. In the next chapter, you'll see the Axis Web Services Deployment Descriptor (WSDD) and how it's used to customize the deployment of your web services.